MLDesign Technologies, Inc.
SOFTWARE LICENSE AGREEMENT

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS OF THIS SOFTWARE LICENSE AGREEMENT AND ANY PROVIDED SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "SOFTWARE LICENSE AGREEMENT") BEFORE USING THIS LICENSED PROGRAM, THE USE OF WHICH IS LICENSED BY MLDesign Technologies, Inc. (MLD) TO ITS CUSTOMERS FOR THEIR USE AS SET FORTH BELOW. OPENING THE SOFTWARE PACKAGE INDICATES THAT YOU ACCEPT THE TERMS OF THIS SOFTWARE LICENSE AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS SOFTWARE LICENSE AGREEMENT, RETURN THE ENTIRE PACKAGE AND ITS CONTENTS TO MLDesign Technologies, Inc. OR TO THE LOCATION WHERE YOU OBTAINED IT.

DEFINITIONS USED IN THIS SOFTWARE LICENSE AGREEMENT

Licensed Program means each executable software program and any updated, improved or otherwise modified versions thereof, together with documentation and error corrections, furnished by MLD pursuant to a Product Quotation or an order from Customer solely for Customer’s internal purposes only; it may include software licensed by MLD from third parties.

Design means the representation of Systems, functions, or architectures, derived from or created by Customer through the use of the Licensed Program in their various formats, including but not limited to, equations, associated parameters, schematic diagrams, textual descriptions, system description languages, and netlists.

Commercial License means a paid annual subscription license to use the Licensed Program whereby the Licensed Program is activated for use via a license key that will automatically deactivate the Licensed Program at the end of the subscription period. You acknowledge that the Licensed Program is enabled with a license key that will automatically deactivate the Licensed Program at the end of the Subscription Period.

Evaluation License means a short-term free-of-charge license to test and evaluate the Licensed Program for a period of 28 days. You acknowledge that the Licensed Program is enabled with a license key that will automatically deactivate the Licensed Program at the end of the Evaluation Period.

Intellectual Property Rights means all rights protected by patents, patent rights, copyrights, trade secrets, service marks and trademarks, and all applications of those rights, in all countries in the world.

Known 3rd Party Confidential Information means any information known by the user of this Software Product to have been designated as confidential by a party other than the user or the user’s employer.

Licensed Use means copying, running, or otherwise executing any portion of the Licensed Pro-
gram including loading data into or displaying, viewing, or extracting output results therefrom for the purpose of Design development, analysis, verification and other related activities. Licensed Use excludes employing or executing any portion of the Licensed Program for the purpose of violating Intellectual Property Rights or improperly discovering Known 3rd Party Confidential Information.

1. LICENSE TO USE. MLDesign Technologies, Inc. (MLD) grants you a non-exclusive and non-transferable license for the internal Licensed Use only of the accompanying Licensed Program, under the terms of an Evaluation License or by the number of users and the class of computer hardware for which the corresponding Commercial License fee has been paid.

2. RESTRICTIONS. Licensed Program is confidential and copyrighted. Title to Licensed Program and all associated intellectual property rights is retained by MLD and/or its licensors. Except as specifically authorized in any Supplemental License Terms, you may not make copies of Licensed Program, other than a single copy of Licensed Program for archive purposes. Unless enforcement is prohibited by applicable law, you may not modify, decompile, or reverse engineer Licensed Program. You acknowledge that Licensed Program is not licensed or intended for violating Intellectual Property Rights or improperly discovering Known 3rd Party Confidential Information. You acknowledge that Licensed Program is not designed, licensed or intended for use in the design, construction, operation or maintenance of any nuclear facility. MLD disclaims any express or implied warranty of fitness for such uses. No right, title or interest in or to any trademark, service mark, logo or trade name of MLD or its licensors is granted under this Agreement.

3. LIMITED WARRANTY. MLD warrants to you that for a period of ninety (90) days from the date of purchase, as evidenced by a copy of the receipt, the media on which Licensed Program is furnished (if any) will be free of defects, in materials and workmanship under normal use and that Licensed Program will substantially conform to the specifications incorporated in this Software License Agreement or in the documentation provided as part of the version of Licensed Program provided to Customer upon purchase of a Commercial License. Except for the foregoing, Licensed Program is provided "AS IS". Your exclusive remedy and MLD’s entire liability under this limited warranty will be at MLD’s option to replace Licensed Program media or refund the fee paid for Licensed Program.

4. DISCLAIMER OF WARRANTY. UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

5. LIMITATION OF LIABILITY. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL MLD OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED, REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE LICENSED PROGRAM, EVEN IF MLD HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. In no event will MLD’s liability to you, whether in contract, tort (including negligence), or other-
wise, exceed the amount paid by you for Licensed Program under this Agreement. The foregoing limitations will apply even if the above stated warranty fails of its essential purpose.

MLDesign Technologies’ MLDesigner Licensed Program incorporates software code and documentation derived from the “Classic” Ptolemy software which is Copyright (c) 1990-1998 by The Regents of the University of California. Such derivative use is permitted with the condition that the following two paragraphs be included with each copy. These paragraphs are hereby incorporated into this MLD Software License Agreement.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

The following paragraphs are included in accordance with the Apache Software license agreement Copyright © 1999 The Apache Software Foundation. All rights reserved.

This product includes software developed by the Apache Software Foundation
http://www.apache.org/

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

6. Termination. This Agreement is effective until terminated. You may terminate this Agreement at any time by destroying all copies of Licensed Program. This Agreement will terminate immediately without notice from MLD if you fail to comply with any provision of this Agreement. Upon Termination, you must destroy all copies of Licensed Program.

7. Export Regulations. All Licensed Program and technical data delivered under this Agreement
are subject to US export control laws and may be subject to export or import regulations in other
countries. You agree to comply strictly with all such laws and regulations and acknowledge that
you have the responsibility to obtain such licenses to export, re-export, or import as may be re-
quired after delivery to you.

8. U.S. Government Restricted Rights. If Licensed Program is being acquired by or on behalf
of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier),
then the Government’s rights in Licensed Program and accompanying documentation will be only
as set forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4
(for U.S. Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for
non-DOD acquisitions).

9. Governing Law. Any action related to this Agreement will be governed by the laws of the
State of California and accompanying regulations. No choice of law rules of any jurisdiction will
apply.

10. Severability. If any provision of this Agreement is held to be unenforceable, this Agreement
will remain in effect with the provision omitted, unless omission would frustrate the intent of the
parties, in which case this Agreement will immediately terminate.

11. Integration. This Agreement is the entire agreement between you and MLD relating to its
subject matter. It supersedes all prior or contemporaneous oral or written communications, pro-
posals, representations and warranties and prevails over any conflicting or additional terms of any
quote, order, acknowledgment, or other communication between the parties relating to its subject
matter during the term of this Agreement. No modification of this Agreement will be binding,
unless in writing and signed by an authorized representative of each party.

Installation of this license file signifies agreement with the MLDesign Technologies Software Li-
cense Agreement provided with this file and with the Licensed Product.

For inquiries please contact:
MLDesign Technologies, Inc.
2130 Hanover St.
Palo Alto, CA 94306

February 2002
Mission Level Design Concept

What is MLDesigner?

MLDesigner is a software system for the design of missions, systems, products, and chips. Its multi-domain simulator for the first time permits the seamless integration of the design flow from mission/operational level to implementation handoff and test of complex designs. The design flow may include,

- Mission/Operational level design tradeoff
  - Modeling and simulation of dynamic Use Cases
  - Mission environment (e.g., terrain, channel models)
  - Operation modeling (e.g., operational models, human input devices, human output devices)
  - Test of complex designs at mission/operational level

- System level design tradeoff
  - System design
  - Computer architecture and distributed processing system design
  - Communication network design
  - Embedded systems design
  - Performance tradeoffs of designs

- Functional level design tradeoff
  - Algorithm design
  - Implementation tradeoff
  - Hardware/Software partitioning

You can use MLDesigner for design and analysis of a broad range of applications, from complex systems like mobile/fixed communication networks, satellite communication/navigation/observation system design, performance and architecture tradeoff of electronic systems and chips, and automotive navigation/communication system design to simple logic design.

MLDesigner consists of

- Multi-document graphical Editor, including Parameter Editor to create, edit and store graph-
ical models of your system

- PTCL command environment to define complex systems that are too large to be defined by a graphical block diagram editor and for parameter tradeoffs

- Animation and plotting system to analyze simulation results and performance measures

- Integrated multi-domain simulators, including debugging animators
  - Discrete Event (DE) simulator
  - Finite State Machine (FSM) simulator
  - Continuous Time/Discrete Event (CTDE) simulator
  - Synchronous Data Flow (SDF) simulator
  - Boolean Data Flow (BDF) simulator
  - Dynamic Data Flow (DDF) simulator
  - High Order Function (HOF) simulator

- more than 1400 core library modules

- more than 260 example systems

The MLDesigner-SatLab interface gives MLDesigner access to SatLab’s mission simulations capabilities of mobile communication nodes and satellites, SatLab’s terrain data base system and terrain based channel models and the SatLab Command Language (SCL) (similar to Ctrl-C and Matlab) and it’s more than 450 functions (You got SatLab with MLDesigner).

You can construct a system model through the graphical editor or the PTcl command language. You may specify the functionality of your modules by a hierarchical block diagram, a finite state machine, a module defined in the C/C++ like PL language, or by a PTcl module definition. Multi-domain simulation modules can be combined and simulated together. The simulation results may be viewed through an animation view while the simulation is running and/or by the post-processing graphical plots.

About this Document

The documentation department are continuously adapting this document in an attempt to keep you up to date with the latest developments in MLDesigner. An updated copy of this document is distributed with each new release of MLDesigner and a copy is posted to our FTP server.

Note: While every effort has been made to ensure the accuracy of this document, MLDesign Technologies Inc. does not make any warranties with respect to this document. Any liability for incidental or consequential damages as a result of availability or use of this document is disclaimed.

If you have upgraded from an earlier version of MLDesigner please be sure to read the what-snewx.x.pdf document found on the cd or on our web server at

Structure of Document

The document is divided into three sections

- Modeling guide - Introduces you to the graphical user interface and walks you through creating primitives, modules, and systems. You are also introduced to the domains available in MLDesigner.
- Programming guide - For advanced users. A knowledge of C++ and Tcl/Tk is assumed.
- Domains - Detailed look at advanced domain issues.

Sections of this document are taken directly from the Ptolemy User's manual. These sections relate to domain issues and Ptolemy language constructs. Please read the Copyright © information in the Software License Agreement. It should also be noted that MLDesign Technologies, Inc. does not necessarily offer support for all domains described in this document. Refer to ch. 6.3 for information about unsupported domains.
I  Modeling Guide  -1-3

1  First Steps with MLDesigner  1-1
  1.1  Basic Terms  1-1
  1.2  Graphical User Interface  1-3
  1.3  Run a Demo Simulation  1-6
    1.3.1  Choose a System  1-6
    1.3.2  Create a Library  1-6
    1.3.3  Save Demo in the Created Library  1-8
    1.3.4  Explore and Run the Demo  1-8
  1.4  Build a Simple Model  1-10
    1.4.1  Create a Sub-library  1-10
    1.4.2  Build a Module  1-11
    1.4.3  Build a System  1-15
    1.4.4  Multiple Iterations and Parameter Sets  1-18
    1.4.5  Xgraph Configuration  1-19
    1.4.6  Build a Primitive  1-19

2  Modeling with MLDesigner  2-1
  2.1  Understanding Environment Variables  2-1
  2.2  Graphical User Interface  2-2
    2.2.1  User Interface Structure  2-2
    2.2.2  Settings  2-2
    2.2.3  Graphical User Interface Filters  2-11
    2.2.4  Workspace  2-11
    2.2.5  Tree View  2-11
    2.2.6  Property Editor  2-19
    2.2.7  Console Window  2-20
    2.2.8  Using Menu Bar  2-21
    2.2.9  Using Toolbars  2-31
  2.3  Handling Models  2-37
    2.3.1  Creating New Models  2-37
    2.3.2  Copying Existing Models  2-40
    2.3.3  Creating Special Primitives  2-41
    2.3.4  Create Model from Source  2-44
2.3.5 Open Existing Models .................................................. 2-45
2.3.6 Model Update ............................................................ 2-46
2.3.7 Deleting Models ........................................................ 2-46
2.3.8 Printing Models .......................................................... 2-47
2.3.9 Exporting EPS ............................................................ 2-50
2.4 Shared Libraries ............................................................ 2-50
2.4.1 Exporting a Top Level Library to Shared Libraries .......... 2-51
2.4.2 Export Libraries ......................................................... 2-51
2.4.3 Import Libraries .......................................................... 2-52
2.4.4 Environment Variables and Dynamic Referencing Mechanism .. 2-53
2.4.5 Set User Environment Variables ..................................... 2-54

3 Developing Models .......................................................... 3-1
3.1 Introduction ................................................................. 3-1
3.2 Steps to Develop Models ................................................. 3-2
3.3 Modifying Model Properties ............................................. 3-3
3.4 Modeling Input/Output Ports .......................................... 3-5
  3.4.1 Introduction ............................................................ 3-6
  3.4.2 Creating Ports ......................................................... 3-8
  3.4.3 Changing Port Properties ............................................ 3-8
3.5 Definition of Parameters ................................................ 3-11
  3.5.1 Introduction ............................................................ 3-11
  3.5.2 Creating Parameters ................................................. 3-11
  3.5.3 Deleting Parameters ................................................. 3-13
3.6 Adding Model Component Instances .................................. 3-14
  3.6.1 Add Model Instance .................................................. 3-14
  3.6.2 Setting Text Label .................................................... 3-15
  3.6.3 Placement of model instances ...................................... 3-16
3.7 Setting Parameters ....................................................... 3-17
  3.7.1 Changing Parameter Values ........................................ 3-18
  3.7.2 Parameter Expressions .............................................. 3-18
  3.7.3 Complex-valued parameters ....................................... 3-19
  3.7.4 Fixed-point parameters ............................................ 3-19
  3.7.5 Linking Parameters .................................................. 3-20
  3.7.6 Inserting Comments in Parameters ............................... 3-23
  3.7.7 Using Tcl Expressions in Parameters ............................. 3-23
  3.7.8 Using MATLAB and MATHEMATICA to Compute Parameters .. 3-26
  3.7.9 Array parameters .................................................... 3-26
  3.7.10 String Parameters .................................................. 3-27
3.8 Connecting Blocks ....................................................... 3-28
3.9 Auto-Forking ............................................................... 3-29
  3.9.1 Relations without formal ports .................................... 3-29
3.10 Using Buses and Delays ............................................... 3-32
  3.10.1 Buses ................................................................. 3-32
  3.10.2 Delays ............................................................... 3-33
3.11 Using Labels for Annotation ........................................... 3-34
## Contents

3.12 Color Settings .......................................................... 3-35
3.13 Using Shared Elements .................................................. 3-36
  3.13.1 Creating Shared Elements ........................................ 3-36
  3.13.2 Setting Shared Model Elements ................................... 3-36
  3.13.3 Exporting Shared Model Elements ............................... 3-36
3.14 Dynamic Instances .................................................... 3-38
  3.14.1 Create a Dynamic Instance ....................................... 3-38
  3.14.2 Example Tutorial ................................................ 3-38
3.15 Dynamic Linking .......................................................... 3-39
  3.15.1 Linked Objects and External Simulations ....................... 3-42
  3.15.2 Permanently Linking Objects to MLDesigner at Startup .... 3-42
3.16 Model Documentation .................................................... 3-43
  3.16.1 Creating Documentation ......................................... 3-43
  3.16.2 Browsing Documentation ......................................... 3-43
3.17 Source Code Editors ..................................................... 3-45

### 4 Debugging and Analyzing Systems

4.1 Breakpoints .............................................................. 4-1
  4.1.1 The Breakpoints Console .......................................... 4-1
  4.1.2 Unconditional Breakpoints ....................................... 4-2
  4.1.3 Module Breakpoints ................................................ 4-3
  4.1.4 Breakpoints in Dynamic Instances ............................... 4-4
  4.1.5 FSM Breakpoints .................................................. 4-5
4.2 Probes ................................................................. 4-6
  4.2.1 Probe Properties .................................................. 4-6
  4.2.2 Probe Primitives .................................................. 4-8
  4.2.3 Port Probes ........................................................ 4-8
  4.2.4 Probes on Memories and Events .................................. 4-8
  4.2.5 Creating User Defined Probes .................................... 4-9
  4.2.6 The DataNew Flag .................................................. 4-9
  4.2.7 Probes on Dynamic Instances .................................... 4-10
4.3 Argument Dependency Highlighting .................................. 4-10
4.4 Compile with Debug Option ............................................ 4-10
4.5 Debugging With External Debugger .................................... 4-11

### 5 MLDesigner Kernel

5.1 Models of Computation .................................................. 5-2
5.2 Mixing Models of Computation ....................................... 5-2
5.3 Simulation Domains ..................................................... 5-3
5.4 Code Generation Domains .............................................. 5-5

### 6 Introduction to MLDesigner Domains

6.1 Foreword to the domain concept ..................................... 6-1
6.2 Supported domains ..................................................... 6-2
  6.2.1 Synchronous Data Flow (SDF) .................................... 6-2
  6.2.2 Higher-Order Functions (HOF) .................................... 6-2
6.2.3 Dynamic Data Flow (DDF) ........................................ 6-2
6.2.4 Boolean Data Flow (BDF) ...................................... 6-3
6.2.5 Discrete Event (DE) ............................................... 6-3
6.2.6 FSM Domain ....................................................... 6-4
6.2.7 CTDE Domain ...................................................... 6-4
6.3 Unsupported domains .................................................. 6-5
6.3.1 Synchronous Reactive (SR) ..................................... 6-5
6.3.2 Multidimensional Synchronous Data Flow (MDSDF) ........ 6-5
6.3.3 Code generation (CG) ............................................. 6-6
6.3.4 Code generation in C (CGC) ..................................... 6-6
6.3.5 Code generation for the Motorola DSP56000 (CG56) ....... 6-6
6.3.6 Code generation in VHDL (VHDL, VHDLB) .................. 6-6
6.4 Summary of various domains ......................................... 6-7
6.5 Targets ............................................................... 6-11

7 Simulation with MLDesigner ............................................ 7-1
7.1 Generate Extern ..................................................... 7-2
7.1.1 Generate C++ .................................................... 7-3
7.1.2 Generate PTcl Extern ............................................ 7-3
7.1.3 Execute on other Platforms ..................................... 7-6
7.1.4 Environment Variables .......................................... 7-6
7.2 Generate & Run Extern ............................................... 7-6
7.2.1 External Parameters ............................................ 7-7
7.2.2 Example .......................................................... 7-7
7.3 Debug Mode .......................................................... 7-8
7.3.1 Place a Breakpoint .............................................. 7-10
7.3.2 Unconditional Breakpoints ..................................... 7-10
7.3.3 Module Breakpoints ............................................. 7-10
7.4 Simulation with Parameter Sets ..................................... 7-11
7.5 Saving Simulation Results ........................................... 7-14
7.5.1 Write Simulation Results to the Console ....................... 7-14
7.5.2 Write Simulation Results to File ................................ 7-14
7.6 Distributed External Simulations .................................... 7-15
7.7 Simulation Statistics ................................................ 7-16

8 Plots, Graphs and Animation ......................................... 8-1
8.1 Animation Using Tk Primitives ..................................... 8-1
8.2 Visualization Using 2D Plotting System ......................... 8-4
8.3 Xgraph Configuration ............................................... 8-4

9 Modeling Using PTCL - The Ptolemy TCL Interpreter ............. 9-1
9.1 Introduction ........................................................ 9-1
9.2 Global information ................................................ 9-1
9.3 Commands for Defining Simulation ................................ 9-2
9.3.1 Creating and deleting Systems ................................. 9-6
9.3.2 Setting the domain ............................................ 9-7
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3.3</td>
<td>Creating instances of primitives and modules</td>
<td>9-7</td>
</tr>
<tr>
<td>9.3.4</td>
<td>Connecting primitives and modules</td>
<td>9-7</td>
</tr>
<tr>
<td>9.3.5</td>
<td>Connecting internal primitives and modules to the outside</td>
<td>9-8</td>
</tr>
<tr>
<td>9.3.6</td>
<td>Setting the value of parameters</td>
<td>9-9</td>
</tr>
<tr>
<td>9.3.7</td>
<td>Setting the number of ports in a primitive</td>
<td>9-10</td>
</tr>
<tr>
<td>9.3.8</td>
<td>Defining new modules</td>
<td>9-10</td>
</tr>
<tr>
<td>9.4</td>
<td>Showing the Current Status</td>
<td>9-11</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Displaying the known classes</td>
<td>9-11</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Displaying information on a the current module or other class</td>
<td>9-12</td>
</tr>
<tr>
<td>9.5</td>
<td>Running the Simulation</td>
<td>9-12</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Creating a schedule</td>
<td>9-12</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Run Length</td>
<td>9-13</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Continuing a simulation</td>
<td>9-13</td>
</tr>
<tr>
<td>9.6</td>
<td>Undo Commands</td>
<td>9-14</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Resetting the interpreter</td>
<td>9-14</td>
</tr>
<tr>
<td>9.6.2</td>
<td>Removing a primitive</td>
<td>9-14</td>
</tr>
<tr>
<td>9.6.3</td>
<td>Removing a connection</td>
<td>9-15</td>
</tr>
<tr>
<td>9.6.4</td>
<td>Removing a node</td>
<td>9-15</td>
</tr>
<tr>
<td>9.7</td>
<td>Targets</td>
<td>9-15</td>
</tr>
<tr>
<td>9.7.1</td>
<td>Available targets</td>
<td>9-15</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Changing the target</td>
<td>9-15</td>
</tr>
<tr>
<td>9.7.3</td>
<td>Changing target parameters</td>
<td>9-15</td>
</tr>
<tr>
<td>9.7.4</td>
<td>Pragmas</td>
<td>9-16</td>
</tr>
<tr>
<td>9.8</td>
<td>Miscellaneous Commands</td>
<td>9-16</td>
</tr>
<tr>
<td>9.8.1</td>
<td>Loading commands from a file</td>
<td>9-16</td>
</tr>
<tr>
<td>9.8.2</td>
<td>Changing the seed of random number generation</td>
<td>9-16</td>
</tr>
<tr>
<td>9.8.3</td>
<td>Changing the current directory</td>
<td>9-17</td>
</tr>
<tr>
<td>9.8.4</td>
<td>Dynamically linking new primitives</td>
<td>9-17</td>
</tr>
<tr>
<td>9.8.5</td>
<td>Top-level blocks</td>
<td>9-18</td>
</tr>
<tr>
<td>9.8.6</td>
<td>Examining parameters</td>
<td>9-18</td>
</tr>
<tr>
<td>9.8.7</td>
<td>Quitting the Interpreter</td>
<td>9-18</td>
</tr>
<tr>
<td>9.8.8</td>
<td>Getting help</td>
<td>9-18</td>
</tr>
<tr>
<td>9.8.9</td>
<td>Registering actions</td>
<td>9-19</td>
</tr>
<tr>
<td>9.9</td>
<td>The Interface to MATLAB and MATHEMATICA</td>
<td>9-19</td>
</tr>
<tr>
<td>9.10</td>
<td>Definition of Shared Elements</td>
<td>9-22</td>
</tr>
<tr>
<td>9.10.1</td>
<td>Defining Memories</td>
<td>9-22</td>
</tr>
<tr>
<td>9.10.2</td>
<td>Defining Events</td>
<td>9-23</td>
</tr>
<tr>
<td>9.10.3</td>
<td>Defining Resources</td>
<td>9-23</td>
</tr>
<tr>
<td>9.11</td>
<td>Definition of Data Structure Types</td>
<td>9-24</td>
</tr>
<tr>
<td>9.11.1</td>
<td>Defining Composite Data Structures</td>
<td>9-24</td>
</tr>
<tr>
<td>9.11.2</td>
<td>Defining Enumerations</td>
<td>9-25</td>
</tr>
<tr>
<td>9.11.3</td>
<td>Handling Data Structures</td>
<td>9-26</td>
</tr>
<tr>
<td>9.12</td>
<td>A Wormhole Example</td>
<td>9-26</td>
</tr>
<tr>
<td>9.13</td>
<td>PTCL as Simulation Control Language</td>
<td>9-30</td>
</tr>
<tr>
<td>9.13.1</td>
<td>Creation of PTCL Scripts</td>
<td>9-30</td>
</tr>
<tr>
<td>9.13.2</td>
<td>Execute the Simulation</td>
<td>9-34</td>
</tr>
</tbody>
</table>
## 10 Shared Model Elements

10.1 Introduction ................................................. 10-1
10.2 Memories ....................................................... 10-1
   10.2.1 Memory Modules ..................................... 10-2
   10.2.2 Local Memory ........................................ 10-2
   10.2.3 Shared Memory ........................................ 10-2
   10.2.4 Global Memory ......................................... 10-3
10.3 Events ......................................................... 10-3
10.4 Resources ...................................................... 10-4
   10.4.1 Introduction .......................................... 10-4
   10.4.2 Quantity Resources .................................... 10-8
   10.4.3 Server Resources ....................................... 10-13

## 11 Import/Conversion of Models

11.1 Converting OCT Models ....................................... 11-1
   11.1.1 Supported Oct types .................................. 11-1
   11.1.2 How to start conversion ................................ 11-1
   11.1.3 Estimated time vs. estimated number of models .... 11-1
   11.1.4 Models that will be converted ......................... 11-1
   11.1.5 Converting or not ....................................... 11-2
   11.1.6 Layout of converted models ............................ 11-2
   11.1.7 Changes .................................................. 11-2
   11.1.8 Parameter list .......................................... 11-3
   11.1.9 Inconsistencies in Oct ................................. 11-3
   11.1.10 Missing interface facets of modules ................ 11-3
   11.1.11 New library structure in MLDesigner ............... 11-3
11.2 Converting BONeS Models .................................... 11-5
   11.2.1 Conversion Conventions ................................ 11-5
   11.2.2 Conversion Conventions For Models ................... 11-5
   11.2.3 Conversion Conventions For Data Structures .......... 11-6
   11.2.4 BONeS Conversion Assistant ............................ 11-6
   11.2.5 Troubleshooting ........................................ 11-7
   11.2.6 Error Messages ......................................... 11-8
   11.2.7 BONeS Categories ....................................... 11-16
   11.2.8 BONeS Primitives ........................................ 11-37
11.3 COSSAP Conversion Tool ...................................... 11-53
   11.3.1 Prerequisites and Limitations ......................... 11-53
   11.3.2 How to Convert COSSAP Project Libraries ............. 11-53
   11.3.3 The User Mapper File ................................... 11-54
   11.3.4 Prefer User Mapper Entries and Overwrite Existing Files 11-54
   11.3.5 Reading Process ....................................... 11-55
   11.3.6 Conversion Process ..................................... 11-57
   11.3.7 Conversion of Schematics. .............................. 11-57
   11.3.8 Model Definition File ................................... 11-59
   11.3.9 History .................................................. 11-61
   11.3.10 Declarations ........................................... 11-61
11.3.11 Functional code ........................................ 11-61
11.3.12 Dataset Handling Library ................................ 11-61
11.3.13 Dataset Parameters ..................................... 11-61
11.3.14 EXIT_FLAG ............................................ 11-62
11.3.15 Unsupported Features .................................. 11-63
11.3.16 Parsing Model Definition File ........................... 11-63
11.3.17 Conversion of Primitive Models ......................... 11-64
11.3.18 Limitations with COSSAP Project Conversion .......... 11-64
11.3.19 Input Dataset File Formats ............................. 11-65
11.3.20 Troubleshooting Guide for Cossap Model Converter .. 11-67

12 Data Structure Management ................................. 12-1
12.1 Managing Data Structures .................................. 12-1
  12.1.1 Creating Data Structures .............................. 12-2
  12.1.2 Adding Composite Members ............................ 12-3
  12.1.3 Editing Composite Members ............................ 12-4
  12.1.4 Deleting Composite Members ........................... 12-4
12.2 Managing Enumeration Elements ........................... 12-5
  12.2.1 Adding Enumeration Elements ......................... 12-5
  12.2.2 Editing Enumeration Elements ......................... 12-5
  12.2.3 Deleting Enumeration Elements ......................... 12-5
12.3 Data Structure Handling Mechanism ....................... 12-5
  12.3.1 Overview of Data Structures .......................... 12-6
  12.3.2 Creating an Enumeration .............................. 12-7
  12.3.3 Creating a Vector .................................... 12-8
  12.3.4 Editing Existing Data Structures ..................... 12-8
  12.3.5 Import Libraries ..................................... 12-9
  12.3.6 Data Structure string representation .................. 12-10
  12.3.7 Data Structure Types ................................ 12-11
12.4 Data Structure Libraries .................................. 12-13
  12.4.1 DSHandling Library ................................ 12-14
  12.4.2 EnumOperations Library ............................... 12-15
  12.4.3 VectorOperations Library ............................. 12-15

II Programming Guide ........................................ 12-1

13 Designing Primitives ....................................... 13-1
  13.1 Introduction ............................................ 13-1
  13.2 Definition of Primitive Interfaces ....................... 13-2
    13.2.1 Model Property Definitions ......................... 13-3
    13.2.2 Load Mode ......................................... 13-3
    13.2.3 Input/Output Port Definitions ..................... 13-4
    13.2.4 Parameter Definitions ............................. 13-4
    13.2.5 Annotations ....................................... 13-5
  13.3 Primitive Functionality Definition ...................... 13-5
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.4 Ptolemy Language Description</td>
<td>13-6</td>
</tr>
<tr>
<td>13.4.1 Compiling Primitives</td>
<td>13-6</td>
</tr>
<tr>
<td>13.4.2 Example</td>
<td>13-7</td>
</tr>
<tr>
<td>13.5 Primitive Language Constructs</td>
<td>13-8</td>
</tr>
<tr>
<td>13.5.1 Keywords in detail</td>
<td>13-9</td>
</tr>
<tr>
<td>13.5.2 Writing C++ Code for Primitives</td>
<td>13-22</td>
</tr>
<tr>
<td>13.5.3 Reading Inputs and Writing Outputs</td>
<td>13-22</td>
</tr>
<tr>
<td>13.5.4 Parameters</td>
<td>13-27</td>
</tr>
<tr>
<td>13.5.5 Array Parameter</td>
<td>13-28</td>
</tr>
<tr>
<td>13.5.6 Programming Examples</td>
<td>13-31</td>
</tr>
<tr>
<td>13.5.7 Preventing Memory Leaks in C++ Code</td>
<td>13-37</td>
</tr>
<tr>
<td>13.6 Infrastructure for Primitive Definition</td>
<td>13-39</td>
</tr>
<tr>
<td>13.6.1 Handling Errors</td>
<td>13-39</td>
</tr>
<tr>
<td>13.6.2 I/O Classes</td>
<td>13-40</td>
</tr>
<tr>
<td>13.6.3 String Functions and Classes</td>
<td>13-45</td>
</tr>
<tr>
<td>13.6.4 List Classes</td>
<td>13-49</td>
</tr>
<tr>
<td>13.6.5 Hash Tables</td>
<td>13-51</td>
</tr>
<tr>
<td>13.6.6 Using Random Numbers</td>
<td>13-53</td>
</tr>
<tr>
<td>14 Using Data Types</td>
<td>14-1</td>
</tr>
<tr>
<td>14.1 Scalar Numeric Types</td>
<td>14-1</td>
</tr>
<tr>
<td>14.1.1 Complex Data Type</td>
<td>14-2</td>
</tr>
<tr>
<td>14.1.2 Fixed-point Data Type</td>
<td>14-4</td>
</tr>
<tr>
<td>14.2 Defining New Data Types</td>
<td>14-15</td>
</tr>
<tr>
<td>14.2.1 Defining a New Message Class</td>
<td>14-16</td>
</tr>
<tr>
<td>14.2.2 Use of the Envelope Class</td>
<td>14-18</td>
</tr>
<tr>
<td>14.2.3 Use of the MessageParticle Class</td>
<td>14-19</td>
</tr>
<tr>
<td>14.2.4 Use of Messages in Primitives</td>
<td>14-20</td>
</tr>
<tr>
<td>14.3 Matrix Data Types</td>
<td>14-23</td>
</tr>
<tr>
<td>14.3.1 Design philosophy</td>
<td>14-23</td>
</tr>
<tr>
<td>14.3.2 PtMatrix Class</td>
<td>14-24</td>
</tr>
<tr>
<td>14.3.3 Public Functions and Operators for the PtMatrix Class</td>
<td>14-24</td>
</tr>
<tr>
<td>14.3.4 Writing Primitives Using the PtMatrix Class</td>
<td>14-32</td>
</tr>
<tr>
<td>14.3.5 Future Extensions</td>
<td>14-36</td>
</tr>
<tr>
<td>14.4 File and String Types</td>
<td>14-36</td>
</tr>
<tr>
<td>14.4.1 File Type</td>
<td>14-36</td>
</tr>
<tr>
<td>14.4.2 String Type</td>
<td>14-37</td>
</tr>
<tr>
<td>14.5 Manipulating Particles of Type \textit{anytype}</td>
<td>14-38</td>
</tr>
<tr>
<td>14.6 Unsupported Types</td>
<td>14-40</td>
</tr>
<tr>
<td>14.6.1 Sub-Matrices</td>
<td>14-40</td>
</tr>
<tr>
<td>14.6.2 Image Particles</td>
<td>14-44</td>
</tr>
<tr>
<td>15 Programming Using Data Structures</td>
<td>15-1</td>
</tr>
<tr>
<td>15.1 Initializing Data Structures</td>
<td>15-1</td>
</tr>
<tr>
<td>15.2 Using Data Structures</td>
<td>15-3</td>
</tr>
<tr>
<td>15.2.1 Generic Type Operations</td>
<td>15-3</td>
</tr>
</tbody>
</table>
15.2.2 Type Specific Interfaces ................................. 15-3
15.3 When to Clone/Release Data Structures. ....................... 15-8
15.4 When is a data structure released? .............................. 15-9
15.5 Compatibility Problems ............................... 15-10
15.6 Known problems ................................ 15-10

16 Using Tcl/Tk in Primitives ................................. 16-1
16.1 Writing Tcl/Tk Scripts for the TclScript Primitive .......... 16-1
   16.1.1 Create a New TclScript Special Primitive .......... 16-3
   16.1.2 The Tcl Script Explained ............................... 16-5
16.2 Tcl Utilities Available to the Programmer .................... 16-8
16.3 Creating Primitives Derived from TclScript ............... 16-13
16.4 Writing Tcl Primitives for DE Domain ...................... 16-14

17 Domain Related Issues ........................................ 17-1
17.1 SDF Domain .............................................. 17-1
17.2 DDF Domain .............................................. 17-2
17.3 BDF Domain .............................................. 17-6
17.4 DE Domain .............................................. 17-7
   17.4.1 Programming Primitives in the DE Domain .......... 17-7
   17.4.2 Programming Examples ................................ 17-18

III Domains .................................................. 17-21

18 SDF Domain ................................................ 18-1
18.1 Introduction ............................................ 18-1
18.2 Basic Data Flow Terminology ................................ 18-1
18.3 Balancing production and consumption of tokens .......... 18-2
18.4 Iterations in SDF ........................................ 18-3
18.5 Inconsistency ............................................ 18-3
18.6 Delays .................................................. 18-4
18.7 Targets ................................................ 18-4
   18.7.1 Default SDF target .................................... 18-4
   18.7.2 The loop-SDF target ................................... 18-6
   18.7.3 SDF to PTCL target ................................... 18-7
18.8 An overview of SDF Primitives ................................ 18-7
18.9 Source primitives ..................................... 18-9
   18.9.1 Floating Point Sources ................................. 18-9
   18.9.2 Fixed-point sources .................................. 18-11
   18.9.3 Complex sources .................................... 18-11
   18.9.4 Matrix Sources ..................................... 18-12
18.10 Sink primitives ..................................... 18-12
   18.10.1 Batch Plotting Facilities .............................. 18-12
18.11 Arithmetic primitives .................................. 18-13
18.12 Nonlinear primitives .................................. 18-14
18.12.1 Quantizers ................................................ 18-14
18.12.2 Math Functions ........................................... 18-15
18.12.3 Other Nonlinear Functions ............................... 18-16
18.13 Logic primitives .............................................. 18-17
18.14 Control primitives ............................................ 18-18
  18.14.1 Single-Rate Operations ................................. 18-18
  18.14.2 Multirate Operations ................................... 18-18
  18.14.3 Other Operations ........................................ 18-19
18.15 Conversion primitives ....................................... 18-19
  18.15.1 Complex data type formats ............................. 18-20
  18.15.2 Other data type formats ................................. 18-20
  18.15.3 Matrix Conversion Primitives ......................... 18-21
18.16 Matrix primitives ........................................... 18-22
  18.16.1 Matrix-Vector Conversion ............................... 18-22
  18.16.2 Matrix operations ....................................... 18-23
  18.16.3 Miscellaneous ........................................... 18-24
18.17 Matlab primitives ............................................ 18-24
18.18 Signal processing (DSP) primitives ....................... 18-26
  18.18.1 Filters ..................................................... 18-26
  18.18.2 Adaptive Filters ......................................... 18-28
  18.18.3 Block Filters ............................................. 18-28
  18.18.4 Vector Quantization ..................................... 18-29
18.19 Spectral analysis ............................................ 18-30
  18.19.1 Miscellaneous signal processing blocks ............... 18-31
18.20 Communication primitives .................................. 18-32
  18.20.1 Sources and Pulse Shapers ............................. 18-32
  18.20.2 Transmitter Functions ................................... 18-32
  18.20.3 Receiver functions ...................................... 18-33
  18.20.4 Channel Models ......................................... 18-33
18.21 Telecomm ..................................................... 18-34
  18.21.1 Conversion, Signal Sources, and Signal Tests ....... 18-34
  18.21.2 Touch tone Decoders ..................................... 18-34
  18.21.3 Channel Models ......................................... 18-35
  18.21.4 PCM and ADPCM ......................................... 18-36
18.22 Spatial Array Processing .................................... 18-36
  18.22.1 Data Models .............................................. 18-36
  18.22.2 Sensor and Antenna Models ............................. 18-36
  18.22.3 Doppler Effects .......................................... 18-37
  18.22.4 Beamforming Methods ................................... 18-37
18.23 Image Processing Primitives ................................. 18-37
  18.23.1 Displaying images ....................................... 18-37
  18.23.2 Reading images .......................................... 18-38
  18.23.3 Color conversions ....................................... 18-39
  18.23.4 Image and video coding ................................. 18-39
  18.23.5 Miscellaneous image blocks ............................. 18-40
18.24 Neural Networks ............................................. 18-41
## DDF Domain

19 DDF Domain

19.1 Introduction .......................................... 19-1
19.2 The DDF Schedulers ....................................... 19-2
   19.2.1 DDF Backward Scheduler .......................... 19-3
   19.2.2 The default scheduler ............................. 19-3
   19.2.3 The clustering scheduler .......................... 19-4
   19.2.4 The fast scheduler ................................. 19-5
   19.2.5 Lazy evaluation .................................... 19-6
19.3 Inconsistency in DDF ..................................... 19-7
19.4 The default-DDF target .................................. 19-7
19.5 An overview of DDF primitives ......................... 19-8
19.6 An overview of DDF demos .............................. 19-9
19.7 Mixing DDF with other domains ......................... 19-10

## BDF Domain

20 BDF Domain .............................................. 20-1

20.1 Introduction ............................................. 20-1
20.2 The default-BDF target .................................. 20-2
20.3 An overview of BDF primitives ......................... 20-2
20.4 An overview of BDF demos .............................. 20-3

## HOF Domain

21 HOF Domain .............................................. 21-1

21.1 Introduction ............................................. 21-1
21.2 Using the HOF domain .................................... 21-2
21.3 The Map primitive and its variants .................... 21-2
   21.3.1 Example ............................................. 21-3
   21.3.2 MapGR and SrcGR primitive ....................... 21-6
   21.3.3 Setting parameter values .......................... 21-6
   21.3.4 Number of replacement blocks .................... 21-9
   21.3.5 How the inputs and outputs are connected ........ 21-9
   21.3.6 A note about data types ........................... 21-9
### 23 CTDE Domain

23.1 Purpose of the domain .................................. 23-1
23.2 Introduction to the CTDE domain .......................... 23-1
23.3 Continuous-Time Computation Models ....................... 23-1
   23.3.1 Computation model .......................... 23-1
   23.3.2 Signal Form ................................ 23-1
   23.3.3 Example: Spring-Mass system .................. 23-2
   23.3.4 Modeling .................................. 23-3
   23.3.5 Simulation .......................... 23-3
   23.3.6 Limitations of purely continuous-time models .... 23-5
   23.4.1 The CTDE Computational Model ............... 23-5
   23.4.2 Model Structure .......................... 23-7
23.5 Modeling in the CTDE domain .......................... 23-8
   23.5.1 Vectorial continuous signals ............... 23-8
   23.5.2 Simulation Algorithm .................. 23-8
23.6 Example: Bouncing Ball-Model ....................... 23-9
23.7 User-adjustable parameters .......................... 23-10
23.8 The ODE solver ................................ 23-10
   23.8.1 Solver parameters .................. 23-10
23.9 The CTDE domain in mixed-signal simulations ........ 23-11
23.10 Current limitations .......................... 23-12

### 24 FSM Domain

24.1 What is a Finite State Machine? ....................... 24-1
24.2 The MLDesigner FSM Domain ........................ 24-1
24.3 MLDesigner FSM Semantic .......................... 24-2
   24.3.1 Basic FSM Elements .................. 24-2
   24.3.2 FSM Action Language .................. 24-8
24.4 FSM Execution Semantics .......................... 24-15
   24.4.1 Initialization .......................... 24-16
   24.4.2 Execution Steps .................. 24-16
24.5 Elevator Example ................................ 24-17
   24.5.1 Interface .......................... 24-17
   24.5.2 Execution .......................... 24-17
24.6 The FSM Model ................................ 24-18
   24.6.1 FSM Model Interface .................. 24-18
   24.6.2 FSM Model Design .................. 24-19
   24.6.3 Current State Data Structure ............. 24-19
   24.6.4 Current State Memory .................. 24-19
   24.6.5 CurrentStateDS Property ............. 24-19
   24.6.6 Internal Event Property ............. 24-20
   24.6.7 Additional Code Property .......... 24-20
## Contents

24.7 FSM Model Editor ........................................... 24-21
24.8 FSM Design Objects ........................................ 24-22
  24.8.1 States .................................................. 24-22
  24.8.2 Transitions ............................................. 24-23
  24.8.3 Default Entrance Transitions ......................... 24-23
  24.8.4 Transition Labels .................................... 24-23
  24.8.5 Default Entrances .................................... 24-23
  24.8.6 Histories .............................................. 24-23
24.9 FSM Dialogs ................................................. 24-23
  24.9.1 Action Dialog ......................................... 24-23
  24.9.2 Event Expression Dialog ............................. 24-24
  24.9.3 Slave Model Dialog .................................. 24-25
24.10 FSM Design Check .......................................... 24-25
  24.10.1 States ................................................ 24-25
  24.10.2 Default Entrances .................................. 24-26
  24.10.3 Histories .............................................. 24-26
24.11 FSM and Concurrency Domains ............................ 24-26
  24.11.1 FSM and DE ......................................... 24-26
  24.11.2 FSM inside DE ....................................... 24-26
  24.11.3 FSM outside DE ..................................... 24-27
  24.11.4 FSM and SDF ........................................ 24-27
  24.11.5 FSM inside SDF ...................................... 24-27
  24.11.6 FSM outside SDF .................................... 24-27
  24.11.7 FSM inside FSM ..................................... 24-28
24.12 Creating an FSM ........................................... 24-28
  24.12.1 System Description ................................ 24-28
  24.12.2 Example .............................................. 24-29
24.13 Backward Compatibility .................................. 24-35
24.14 ANSI C Code Synthesis ................................... 24-36
  24.14.1 Overview ............................................. 24-36
  24.14.2 Generator Input ..................................... 24-36
  24.14.3 Generator Output ................................... 24-37
  24.14.4 Limitations .......................................... 24-37
  24.14.5 Code Generation Process ........................... 24-41
  24.14.6 Run-Time Environment ............................... 24-43
  24.14.7 Output Source Files ................................. 24-45
  24.14.8 Code Customization ................................. 24-47
  24.14.9 Code Debugging ..................................... 24-49
  24.14.10 Example ............................................. 24-50

25 NS2 Domain .................................................. 25-1
  25.1 Introduction ............................................. 25-1
  25.2 MLDesigner and NS2 ..................................... 25-1
    25.2.1 Modeling Networks with MLDesigner ............. 25-1
    25.2.2 About NS2 .......................................... 25-2
    25.2.3 Linking MLDesigner and NS2 ....................... 25-3
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.3</td>
<td>Working with the MLDesigner NS2 Domain</td>
<td>25-5</td>
</tr>
<tr>
<td>25.3.1</td>
<td>Getting Started</td>
<td>25-5</td>
</tr>
<tr>
<td>25.3.2</td>
<td>Assembling NS2 Models to Build Simulations</td>
<td>25-6</td>
</tr>
<tr>
<td>25.3.3</td>
<td>Writing New NS2 Primitives</td>
<td>25-14</td>
</tr>
<tr>
<td>26</td>
<td>Unsupported Domains</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1</td>
<td>SR Domain</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1.1</td>
<td>Introduction</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1.2</td>
<td>SR concepts</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1.3</td>
<td>SR compared to other domains</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1.4</td>
<td>The semantics of SR</td>
<td>26-2</td>
</tr>
<tr>
<td>26.1.5</td>
<td>Overview of SR primitives</td>
<td>26-2</td>
</tr>
<tr>
<td>27</td>
<td>Code Generation Domains - unsupported</td>
<td>27-1</td>
</tr>
<tr>
<td>27.1</td>
<td>VHDL Domain</td>
<td>27-1</td>
</tr>
<tr>
<td>27.1.1</td>
<td>Introduction</td>
<td>27-1</td>
</tr>
<tr>
<td>27.1.2</td>
<td>VHDL Targets</td>
<td>27-3</td>
</tr>
<tr>
<td>27.1.3</td>
<td>An Overview of VHDL Primitives</td>
<td>27-7</td>
</tr>
<tr>
<td>27.1.4</td>
<td>An Overview of VHDL Demos</td>
<td>27-8</td>
</tr>
<tr>
<td>27.2</td>
<td>CG Domain</td>
<td>27-9</td>
</tr>
<tr>
<td>27.2.1</td>
<td>Introduction</td>
<td>27-9</td>
</tr>
<tr>
<td>27.2.2</td>
<td>Targets</td>
<td>27-10</td>
</tr>
<tr>
<td>27.2.3</td>
<td>Schedulers</td>
<td>27-14</td>
</tr>
<tr>
<td>27.2.4</td>
<td>Interfacing Issues</td>
<td>27-17</td>
</tr>
<tr>
<td>27.2.5</td>
<td>Dynamic constructs in CG domain</td>
<td>27-18</td>
</tr>
<tr>
<td>27.2.6</td>
<td>Primitives</td>
<td>27-21</td>
</tr>
<tr>
<td>27.2.7</td>
<td>Demos</td>
<td>27-23</td>
</tr>
<tr>
<td>27.3</td>
<td>CGC Domain</td>
<td>27-24</td>
</tr>
<tr>
<td>27.3.1</td>
<td>Introduction</td>
<td>27-24</td>
</tr>
<tr>
<td>27.3.2</td>
<td>CGC Targets</td>
<td>27-24</td>
</tr>
<tr>
<td>27.3.3</td>
<td>An Overview of CGC Primitives</td>
<td>27-28</td>
</tr>
<tr>
<td>27.3.4</td>
<td>An Overview of CGC Demos</td>
<td>27-31</td>
</tr>
<tr>
<td>27.4</td>
<td>CG56 Domain</td>
<td>27-35</td>
</tr>
<tr>
<td>27.4.1</td>
<td>Introduction</td>
<td>27-35</td>
</tr>
<tr>
<td>27.4.2</td>
<td>An overview of CG56 primitives</td>
<td>27-35</td>
</tr>
<tr>
<td>27.4.3</td>
<td>An overview of CG56 Demos</td>
<td>27-43</td>
</tr>
<tr>
<td>27.4.4</td>
<td>Targets</td>
<td>27-46</td>
</tr>
<tr>
<td>27.5</td>
<td>C50 Domain</td>
<td>27-48</td>
</tr>
<tr>
<td>27.5.1</td>
<td>Introduction</td>
<td>27-48</td>
</tr>
<tr>
<td>27.5.2</td>
<td>An overview of C50 primitives</td>
<td>27-48</td>
</tr>
<tr>
<td>27.5.3</td>
<td>Source primitives</td>
<td>27-49</td>
</tr>
<tr>
<td>27.5.4</td>
<td>An overview of C50 Demos</td>
<td>27-55</td>
</tr>
<tr>
<td>27.5.5</td>
<td>Targets</td>
<td>27-56</td>
</tr>
</tbody>
</table>
IV Appendix

A General
A.1 System Requirements ............................................. A-1
A.2 Environment Variables ........................................... A-3
A.3 Valid File Names .................................................. A-3
A.4 Uninstall MLDesigner ............................................ A-4
A.5 Version Update Warning ..................................... A-4

B Support
B.1 How to Contact Us ................................................. B-1
B.2 Reporting Problems/Bugs ...................................... B-1
B.3 Viewing the Online Documentation ..................... B-1

C Frequently Asked Questions
C.1 General Questions ................................................ C-1
C.2 Error Messages and Their Most Common Causes ............ C-1
C.3 Segmentation Faults .............................................. C-1
C.4 Data Structures .................................................... C-1
C.5 Load Mode .......................................................... C-2
C.6 Plotting Systems ................................................... C-2
C.7 Setting Environment Variables ............................... C-2
C.8 ddd debugger and Red Hat ................................... C-2
C.9 Linked Objects ..................................................... C-2
C.10 Shared Libraries ................................................ C-2

D Answers to FAQ's
D.1 Answers for the General Questions ......................... D-1
D.2 Error Messages and Their Causes ............................ D-1
D.3 Segmentation Faults / System Crashes ...................... D-2
D.4 Data Structures .................................................... D-2
D.5 Load Mode .......................................................... D-2
D.6 Plotting Systems ................................................... D-3
D.7 Setting Environment Variables ............................... D-3
D.8 Using ddd Debugger under Red Hat ....................... D-5
D.9 Linked Objects ..................................................... D-6
D.10 Shared Libraries ................................................ D-6

E Troubleshooting
E.1 Closing complex models becomes slower and slower after simulations ...... E-1
E.2 DHCP Client/License problem ................................ E-1
E.3 Waiting for Users Lock .......................................... E-1
E.4 Distributed Simulation Timeout ................................ E-2
E.5 ddd Debugger Problems ....................................... E-2
E.6 Compile Errors ..................................................... E-3
E.7 Preserving the Order of Multiple Outputs in Priority Free Scheduler .......... E-3
E.8 MLDesigner Does Not Start After the Splash Screen Disappears ............ E-3
Contents

E.9 Red Hat Linux 9 ............................................. E-4
E.10 Red Hat Enterprise Linux 4 32 bit ....................... E-5
E.12 VMWare Player .......................................... E-6
E.13 QClipboard::Unknown SelectionNotify ................. E-6
E.14 Value of MLD_USER variable is lost ................... E-7
E.15 The license manager fails on a system with multiple NICs ........ E-7
E.16 MySQL ................................................. E-7

F Abbreviations ................................................. F-1

G Glossary ....................................................... G-1

H Bibliography ................................................ H-1

I Index ......................................................... I-1
# List of Figures

1.1 Embedded Model Instance .................................................. 1-2
1.2 The MLDesigner Graphical User Interface. ............................. 1-4
1.3 testPacket System ............................................................ 1-7
1.4 The New Model dialog ...................................................... 1-7
1.5 Comparative Packetized/Non Packetized Output of testPacket ...... 1-9
1.6 New Model - Library Menu Item ......................................... 1-11
1.7 Complete Module MyAdderModule .................................... 1-15
1.8 Complete System MyAdderSystem .................................... 1-17
1.9 The Xgraph Plot with Cumulation=Paramsets ...................... 1-20
1.10 Simple Adder Primitive ................................................... 1-22

2.1 Settings Dialog ............................................................. 2-2
2.2 Search Dialog ............................................................... 2-12
2.3 Tree View Tabs .............................................................. 2-12
2.4 Recognized physical file types ......................................... 2-13
2.5 Recognized physical file types ......................................... 2-13
2.6 Logical Reference Icons in the Library View ....................... 2-14
2.7 Tree View filter toolbar ................................................... 2-16
2.8 Tree View Context Menu Examples ................................... 2-16
2.9 Property Editor example .................................................. 2-20
2.10 Simulation Properties Window ......................................... 2-21
2.11 File menu ................................................................. 2-22
2.12 Edit menu examples ...................................................... 2-24
2.13 Edit menu examples ...................................................... 2-24
2.14 View menu ................................................................. 2-30
2.15 Window menu .............................................................. 2-30
2.16 Standard toolbar .......................................................... 2-31
2.17 Tree View filter toolbar .................................................. 2-34
2.18 Editor toolbar .............................................................. 2-35
2.19 Creation Dialog for the Example ...................................... 2-38
2.20 “Save as” Including Specials ........................................... 2-43
2.21 Create Special Primitive Dialog ...................................... 2-43
2.22 Create Special Primitive - Parameter Logic ......................... 2-44
2.23 Update Model dialog ...................................................... 2-46
2.24 Printed model example ................................................... 2-48
2.25 Print setup dialog .......................................................... 2-48
2.26 Printer Configuration Dialog ........................................... 2-49
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.27</td>
<td>Export Top Level Library Dialog</td>
<td>2-52</td>
</tr>
<tr>
<td>2.28</td>
<td>Import Top Level Library Dialog</td>
<td>2-53</td>
</tr>
<tr>
<td>2.29</td>
<td>Library Structure and System</td>
<td>2-54</td>
</tr>
<tr>
<td>3.1</td>
<td>Model properties of the example</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2</td>
<td>Select Primitive to Derive From</td>
<td>3-5</td>
</tr>
<tr>
<td>3.3</td>
<td>Representation of port types in MLD</td>
<td>3-7</td>
</tr>
<tr>
<td>3.4</td>
<td>Port property editor plane</td>
<td>3-9</td>
</tr>
<tr>
<td>3.5</td>
<td>Example models after the creation of the ports</td>
<td>3-10</td>
</tr>
<tr>
<td>3.6</td>
<td>Property editor for defining new parameters</td>
<td>3-11</td>
</tr>
<tr>
<td>3.7</td>
<td>Context menus for creating formal parameters</td>
<td>3-12</td>
</tr>
<tr>
<td>3.8</td>
<td>Context menu for deleting parameters</td>
<td>3-13</td>
</tr>
<tr>
<td>3.9</td>
<td>Select model dialog</td>
<td>3-15</td>
</tr>
<tr>
<td>3.10</td>
<td>Example models after the creation of the model instances</td>
<td>3-17</td>
</tr>
<tr>
<td>3.11</td>
<td>Model instance properties example</td>
<td>3-18</td>
</tr>
<tr>
<td>3.12</td>
<td>Parameter Expression Dialog</td>
<td>3-19</td>
</tr>
<tr>
<td>3.13</td>
<td>Parameter linking in the singen module</td>
<td>3-20</td>
</tr>
<tr>
<td>3.14</td>
<td>Parameter settings for the example</td>
<td>3-21</td>
</tr>
<tr>
<td>3.15</td>
<td>Parameter settings for the example</td>
<td>3-22</td>
</tr>
<tr>
<td>3.16</td>
<td>Parameter settings for the example</td>
<td>3-24</td>
</tr>
<tr>
<td>3.17</td>
<td>Example models after connecting the model instances</td>
<td>3-29</td>
</tr>
<tr>
<td>3.18</td>
<td>Single to Multi and Multi to Single Port</td>
<td>3-30</td>
</tr>
<tr>
<td>3.19</td>
<td>Single to Many Multi Port</td>
<td>3-30</td>
</tr>
<tr>
<td>3.20</td>
<td>Many Single to Single/Multi Ports</td>
<td>3-30</td>
</tr>
<tr>
<td>3.21</td>
<td>Forks may Not be used in conjunction with Port Type <strong>imulti</strong></td>
<td>3-31</td>
</tr>
<tr>
<td>3.22</td>
<td>Example models after connecting the model instances</td>
<td>3-31</td>
</tr>
<tr>
<td>3.23</td>
<td>Example models after connecting the model instances</td>
<td>3-32</td>
</tr>
<tr>
<td>3.24</td>
<td>Add Text Label dialog</td>
<td>3-34</td>
</tr>
<tr>
<td>3.25</td>
<td>Color Selection Dialog</td>
<td>3-35</td>
</tr>
<tr>
<td>3.26</td>
<td>Model after colors are changed</td>
<td>3-35</td>
</tr>
<tr>
<td>3.27</td>
<td>Property context menu for shared element</td>
<td>3-37</td>
</tr>
<tr>
<td>3.28</td>
<td>Dynamic Instance of Xgraph in the DE domain</td>
<td>3-39</td>
</tr>
<tr>
<td>3.29</td>
<td>Linked Objects Dialog</td>
<td>3-41</td>
</tr>
<tr>
<td>3.30</td>
<td>Example of Dynamic Linking</td>
<td>3-42</td>
</tr>
<tr>
<td>3.31</td>
<td>Generated hypertext documentation</td>
<td>3-44</td>
</tr>
<tr>
<td>3.32</td>
<td>Generated hypertext documentation</td>
<td>3-44</td>
</tr>
<tr>
<td>4.1</td>
<td>Breakpoint Properties Window</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2</td>
<td>Context Menu in Breakpoints Console</td>
<td>4-2</td>
</tr>
<tr>
<td>4.3</td>
<td>The Select Source Module dialog for Breakpoints</td>
<td>4-3</td>
</tr>
<tr>
<td>4.4</td>
<td>sinMod with singen#1 Model Instance</td>
<td>4-4</td>
</tr>
<tr>
<td>4.5</td>
<td>Breakpoint Properties window and Select Source Models Dialog</td>
<td>4-5</td>
</tr>
<tr>
<td>4.6</td>
<td>Probe Properties Editor</td>
<td>4-8</td>
</tr>
<tr>
<td>5.1</td>
<td>Interface between internal and external domains</td>
<td>5-3</td>
</tr>
<tr>
<td>5.2</td>
<td>Hierarchical Model Structure</td>
<td>5-4</td>
</tr>
<tr>
<td>5.3</td>
<td>MLD designer domains</td>
<td>5-4</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.4</td>
<td>Hierarchical system model structure.</td>
<td>5-6</td>
</tr>
<tr>
<td>7.1</td>
<td>Simulation Icons on the Toolbar</td>
<td>7-1</td>
</tr>
<tr>
<td>7.2</td>
<td>Step Into Example</td>
<td>7-2</td>
</tr>
<tr>
<td>7.3</td>
<td>Step Over Example</td>
<td>7-2</td>
</tr>
<tr>
<td>7.4</td>
<td>Finish Example</td>
<td>7-3</td>
</tr>
<tr>
<td>7.5</td>
<td>Additional controls in TclScript panel</td>
<td>7-8</td>
</tr>
<tr>
<td>7.6</td>
<td>Simulation control window with Tk slider elements</td>
<td>7-9</td>
</tr>
<tr>
<td>7.7</td>
<td>Simulation control window with Tk slider elements</td>
<td>7-9</td>
</tr>
<tr>
<td>7.8</td>
<td>Breakpoint Properties Window</td>
<td>7-10</td>
</tr>
<tr>
<td>7.9</td>
<td>Data Type Hierarchy</td>
<td>7-12</td>
</tr>
<tr>
<td>7.10</td>
<td>Used Computers Dialog</td>
<td>7-17</td>
</tr>
<tr>
<td>8.1</td>
<td>Sink primitive model components for animation</td>
<td>8-1</td>
</tr>
<tr>
<td>8.2</td>
<td>Model</td>
<td>8-3</td>
</tr>
<tr>
<td>8.3</td>
<td>Animation</td>
<td>8-3</td>
</tr>
<tr>
<td>8.4</td>
<td>XMGraph Configuration Syntax</td>
<td>8-5</td>
</tr>
<tr>
<td>8.5</td>
<td>XMGraph example</td>
<td>8-7</td>
</tr>
<tr>
<td>9.1</td>
<td>Model of the wormhole example</td>
<td>9-29</td>
</tr>
<tr>
<td>9.2</td>
<td>Simulation result of the wormhole example</td>
<td>9-29</td>
</tr>
<tr>
<td>9.3</td>
<td>Model of the example</td>
<td>9-30</td>
</tr>
<tr>
<td>10.1</td>
<td>CPU Demo Showing Linking of Resource Elements</td>
<td>10-6</td>
</tr>
<tr>
<td>10.2</td>
<td>Allocate and Free blocks</td>
<td>10-9</td>
</tr>
<tr>
<td>10.3</td>
<td>Service block</td>
<td>10-14</td>
</tr>
<tr>
<td>10.4</td>
<td>Modify SRM block</td>
<td>10-19</td>
</tr>
<tr>
<td>11.1</td>
<td>Convert BONeS Model Dialog</td>
<td>11-7</td>
</tr>
<tr>
<td>11.2</td>
<td>Convert COSSAP Dialog</td>
<td>11-54</td>
</tr>
<tr>
<td>11.3</td>
<td>Log Entries after Reading a Project</td>
<td>11-55</td>
</tr>
<tr>
<td>11.4</td>
<td>Missing Libraries Dialog</td>
<td>11-56</td>
</tr>
<tr>
<td>12.1</td>
<td>Data Type Editor structure</td>
<td>12-2</td>
</tr>
<tr>
<td>12.2</td>
<td>Add Member dialog</td>
<td>12-3</td>
</tr>
<tr>
<td>12.3</td>
<td>Data structure member context menu</td>
<td>12-4</td>
</tr>
<tr>
<td>12.4</td>
<td>Data Structure Member editor</td>
<td>12-4</td>
</tr>
<tr>
<td>12.5</td>
<td>Data Type Selection for Input/Output Ports</td>
<td>12-6</td>
</tr>
<tr>
<td>12.6</td>
<td>Data Structure Selection for Input/Output Ports</td>
<td>12-7</td>
</tr>
<tr>
<td>12.7</td>
<td>Data Structure Editor dialog</td>
<td>12-8</td>
</tr>
<tr>
<td>12.8</td>
<td>Data Structure Hierarchy</td>
<td>12-10</td>
</tr>
<tr>
<td>12.9</td>
<td>Select Data Structure dialog</td>
<td>12-12</td>
</tr>
<tr>
<td>13.1</td>
<td>Toolbar for primitive interface models</td>
<td>13-2</td>
</tr>
<tr>
<td>13.2</td>
<td>Property Editor for primitive example</td>
<td>13-4</td>
</tr>
<tr>
<td>13.3</td>
<td>Primitive editor</td>
<td>13-6</td>
</tr>
<tr>
<td>13.4</td>
<td>Example of output of the plotting system using the XGraph class</td>
<td>13-41</td>
</tr>
<tr>
<td>13.5</td>
<td>Example of animated bar graph using the BarGraph class</td>
<td>13-43</td>
</tr>
</tbody>
</table>
List of Figures

13.6 Random Number Generator Properties ........................................... 13-54
16.1 Examples of TclScript icons ......................................................... 16-2
16.2 System model of TclScript demo .................................................. 16-2
16.3 Full path where TclScript.tcl is saved ......................................... 16-4
16.4 Additional controls in TclRunControl panel ................................. 16-4
16.5 System model of TclScript demo .................................................. 16-6
16.6 Simulation control window for the TclScript demo ....................... 16-11

18.1 A simple connection of SDF primitives, used to illustrate the use of balance equations in constructing a schedule. ......................... 18-2
18.2 System Illustrating Iterations in the SDF Domain ....................... 18-4
18.3 Logic primitives in the SDF library. ............................................. 18-17

19.1 A simple example used to illustrate the notion of an iteration .......... 19-3
19.2 (a) The EndCase primitive waits on the control port. (b) The primitive fires when data arrives on the control port (the value of the data is 0). (c) Now the primitive waits for input to arrive on input port 0. (d) The primitive fires again when data arrives on input port 0. (e) The data that arrived on input port 0 is transmitted by the output port of the EndCase primitive. ......................... 19-6

21.1 An example of the use of the Map primitive to plot three different cosine pulses .......................................................... 21-4
21.2 The plot that results from running the system ............................. 21-5
21.3 A block diagram equivalent to the demo above, but without higher-order functions ............................................................. 21-5
21.4 A block diagram equivalent to the two demos above, except that the number of instances can be specified by a parameter ......................... 21-5
21.5 A block diagram equivalent to the last demo, except that the replacement blocks are specified graphically ........................................ 21-6
21.6 A more complicated example using higher-order functions with the number of replacement blocks graphically defined .......................... 21-7
21.7 The plot created by running the system above ............................. 21-7
21.8 A recursive system, where the IfElse HOF primitive replaces itself with an instance of the same system until its condition parameter gets to zero ......................... 21-11
21.9 System with BusMerge instances ............................................... 21-12
21.10 The Nop primitive is used to create busses from individual connections, to break busses down into individual lines and to break out multiports into individual ports ....................................................... 21-13

22.1 When DE primitives are enabled by simultaneous events, the choice of which to fire is determined by priorities based on a topological sort. Thus if B and C both have events with identical time stamps, B will take priority over C. The delay on the path from C to A serves to break the topological sort. ......................... 22-3
22.2 When an SDF domain appears within a DE domain, events at the input to the SDF subsystem result in zero-delay events at the output of the SDF subsystem. Thus, the time stamps at the output are identically to the time stamps at the input ... 22-5
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.3</td>
<td>A typical DE subsystem intended for inclusion within an SDF subsystem. When a DE subsystem appears within an SDF system, the DE subsystem must ensure that the appropriate number of output events are produced in response to input events. This is typically accomplished with a “Sampler” primitive</td>
<td>22-6</td>
</tr>
<tr>
<td>23.1</td>
<td>A continuous-time signal</td>
<td>23-2</td>
</tr>
<tr>
<td>23.2</td>
<td>Example: Spring-Mass System</td>
<td>23-2</td>
</tr>
<tr>
<td>23.3</td>
<td>Spring-Mass Example: Simulation results</td>
<td>23-3</td>
</tr>
<tr>
<td>23.4</td>
<td>Example: CTDE Model of the Spring-Mass system</td>
<td>23-4</td>
</tr>
<tr>
<td>23.5</td>
<td>A Discrete-Event signal</td>
<td>23-6</td>
</tr>
<tr>
<td>23.6</td>
<td>Structure of a combined model</td>
<td>23-6</td>
</tr>
<tr>
<td>23.7</td>
<td>The LimitedIntegrator primitive as example of a general hybrid block</td>
<td>23-7</td>
</tr>
<tr>
<td>23.8</td>
<td>Spring-Mass model using vectorial signals and a StateSpace Block</td>
<td>23-8</td>
</tr>
<tr>
<td>23.9</td>
<td>Bouncing Ball System Model</td>
<td>23-9</td>
</tr>
<tr>
<td>23.10</td>
<td>Bouncing Ball output</td>
<td>23-9</td>
</tr>
<tr>
<td>23.11</td>
<td>Plant Process model</td>
<td>23-11</td>
</tr>
<tr>
<td>23.12</td>
<td>Control System model</td>
<td>23-12</td>
</tr>
<tr>
<td>23.13</td>
<td>Plant process output graph showing the effects of the control system</td>
<td>23-12</td>
</tr>
<tr>
<td>24.1</td>
<td>Transition with Label</td>
<td>24-4</td>
</tr>
<tr>
<td>24.2</td>
<td>An FSM Self Transition</td>
<td>24-4</td>
</tr>
<tr>
<td>24.3</td>
<td>Default Entrance</td>
<td>24-6</td>
</tr>
<tr>
<td>24.4</td>
<td>Recursive(*) and Static History symbols</td>
<td>24-7</td>
</tr>
<tr>
<td>24.5</td>
<td>FSM Example, Simple Elevator</td>
<td>24-17</td>
</tr>
<tr>
<td>24.6</td>
<td>Additional Icons for FSM models</td>
<td>24-22</td>
</tr>
<tr>
<td>24.7</td>
<td>FSM Design Objects</td>
<td>24-22</td>
</tr>
<tr>
<td>24.8</td>
<td>FSM Action Dialog with Sub-Diags</td>
<td>24-24</td>
</tr>
<tr>
<td>24.9</td>
<td>Event Expression Dialog</td>
<td>24-24</td>
</tr>
<tr>
<td>24.10</td>
<td>Slave Model Dialog</td>
<td>24-24</td>
</tr>
<tr>
<td>24.11</td>
<td>ReflexGame2 System in the DE domain</td>
<td>24-25</td>
</tr>
<tr>
<td>24.12</td>
<td>game2_FSM#1 FSM</td>
<td>24-29</td>
</tr>
<tr>
<td>24.13</td>
<td>the GameOn slave module, in DE domain</td>
<td>24-31</td>
</tr>
<tr>
<td>24.14</td>
<td>rule1_FSM for GameOn</td>
<td>24-33</td>
</tr>
<tr>
<td>24.15</td>
<td>rule2_FSM for GameOn</td>
<td>24-34</td>
</tr>
<tr>
<td>24.16</td>
<td>The count module, in SDF domain</td>
<td>24-34</td>
</tr>
<tr>
<td>24.17</td>
<td>FSM Module</td>
<td>24-35</td>
</tr>
<tr>
<td>24.18</td>
<td>Code Generator Configuration Dialog</td>
<td>24-36</td>
</tr>
<tr>
<td>24.19</td>
<td>RTOS Timelines</td>
<td>24-42</td>
</tr>
<tr>
<td>24.20</td>
<td>Dynamic Memory Partition</td>
<td>24-44</td>
</tr>
<tr>
<td>24.21</td>
<td>Code Generator Output Files</td>
<td>24-45</td>
</tr>
<tr>
<td>24.22</td>
<td>Hardware/Software Architecture - Customization Layers</td>
<td>24-46</td>
</tr>
<tr>
<td>24.23</td>
<td>LEGO® Mindstorms™ Block Sorter Robot</td>
<td>24-47</td>
</tr>
<tr>
<td>24.24</td>
<td>Block Sorter Modeling: Conveyor Belt FSM</td>
<td>24-51</td>
</tr>
<tr>
<td>24.25</td>
<td>Block Sorter Modeling: Sorter FSM</td>
<td>24-52</td>
</tr>
<tr>
<td>24.26</td>
<td>Block Sorter Modeling: Controller FSM Module</td>
<td>24-53</td>
</tr>
<tr>
<td>24.27</td>
<td>Block Sorter Simulation: System Model</td>
<td>24-53</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>24.28</td>
<td>Block Sorter Simulation: Control Panel</td>
<td>24-54</td>
</tr>
<tr>
<td>24.29</td>
<td>Block Sorter Simulation: 3D Visualization</td>
<td>24-54</td>
</tr>
<tr>
<td>25.1</td>
<td>Hello World Example</td>
<td>25-8</td>
</tr>
<tr>
<td>25.2</td>
<td>MLDesigner executing an NS2 Script</td>
<td>25-10</td>
</tr>
<tr>
<td>25.3</td>
<td>Nam analyzing the Script-System</td>
<td>25-11</td>
</tr>
<tr>
<td>25.4</td>
<td>The “Using Nam” system</td>
<td>25-12</td>
</tr>
<tr>
<td>25.5</td>
<td>Nam output for the demo system</td>
<td>25-12</td>
</tr>
<tr>
<td>25.6</td>
<td>DE Wormhole System</td>
<td>25-13</td>
</tr>
<tr>
<td>25.7</td>
<td>NS2 Module inside the DE Wormhole system</td>
<td>25-13</td>
</tr>
<tr>
<td>25.8</td>
<td>XGraph results for the DE Wormhole system</td>
<td>25-13</td>
</tr>
<tr>
<td>25.9</td>
<td>Architecture of MLDesigner and NS2</td>
<td>25-14</td>
</tr>
<tr>
<td>25.10</td>
<td>The Primitive Class Hierarchy</td>
<td>25-20</td>
</tr>
<tr>
<td>25.11</td>
<td>Architecture: sending primitive data to MLDesigner</td>
<td>25-24</td>
</tr>
<tr>
<td>25.12</td>
<td>System: Port data transfer using Tcl</td>
<td>25-25</td>
</tr>
<tr>
<td>25.13</td>
<td>Architecture: sending user-defined data to MLDesigner</td>
<td>25-29</td>
</tr>
<tr>
<td>25.14</td>
<td>System: GUI warnings, messages and errors</td>
<td>25-32</td>
</tr>
<tr>
<td>25.15</td>
<td>The commands window ’Log’ tab window after “GUI warnings, ...” was run</td>
<td>25-32</td>
</tr>
<tr>
<td>D.1</td>
<td>Generated hypertext documentation</td>
<td>D-4</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Default Settings Options ................................................. 2-10

3.1 Common properties for different model types ....................... 3-6
3.2 Possible data type combinations ........................................ 3-6
3.3 Possible data type combinations ........................................ 3-6
3.4 Parameter data types supported by MLDesigner ..................... 3-47
3.5 Colors defined for model instances ................................. 3-48
3.6 Domains containing Merge and Fork Primitives .................... 3-48

4.1 Probe Properties Editor .................................................. 4-7
4.2 Probe Methods .............................................................. 4-9

6.1 Summary of various domains ............................................. 6-10
6.2 SDF Targets and Target Parameters .................................... 6-13

7.1 generated files .............................................................. 7-4
7.2 files in extern directory .................................................. 7-4
7.3 Permutation of Simulation with Three Step Parameters and Three Start Value ........................................... 7-13
7.4 Permutation of Simulation with Two Parameter Sets and Three Start Value ........................................... 7-14
7.5 Host registry for Distributed Simulations ............................. 7-16

9.1 Summary of PTCL commands .............................................. 9-6
9.2 Commands for the MATLAB interface ................................. 9-20
9.3 Commands for the MATHEMATICA interface ......................... 9-20

10.1 The chain of the linked resources in CPU Demo ................... 10-5
10.2 Quantity Resource Attributes ........................................... 10-7
10.3 Server Resource Attributes ............................................. 10-8
10.4 Quantity Resource Attributes ........................................... 10-13
10.5 Server Resource Attributes ............................................. 10-20

11.1 Common Errors after BONeS Model Conversion .................... 11-15
11.2 Address Mapping .......................................................... 11-16
11.3 Arithmetic ................................................................. 11-16
11.4 Arithmetic> Integer ....................................................... 11-17
11.5 Arithmetic> Real .......................................................... 11-18
11.6 Bitwise Operations> Integer ............................................. 11-18
11.7 Comparison ................................................................. 11-19
11.8 Conversions ........................................... 11-19
11.9 Counters .............................................. 11-19
11.10 DS Access/Modify .................................... 11-20
11.11 DS TYPE Operations ................................. 11-20
11.12 Delays .................................................. 11-20
11.13 Execution Control .................................... 11-21
11.14 File Access ........................................... 11-22
11.15 GotoGroup ............................................. 11-23
11.16 GotoGroup ............................................. 11-23
11.17 Logical ............................................... 11-24
11.18 Loops ................................................... 11-24
11.19 Memory Access > Global Memory .................. 11-24
11.20 Memory Access > Linked Memory ................. 11-25
11.21 Memory Access > Local Memory .................. 11-25
11.22 Miscellaneous ........................................ 11-25
11.23 Number Generators .................................. 11-26
11.24 Number Generators > Random ..................... 11-27
11.25 Quantity-Shared Resource ......................... 11-27
11.26 Quantity-Shared Resource > *Internals* ......... 11-27
11.27 Queues ............................................... 11-28
11.28 Queues > Components ............................... 11-29
11.29 Queues & Servers .................................... 11-29
11.30 Queues & Servers > *Internals* .................... 11-29
11.31 SET Operations ...................................... 11-30
11.32 Server Resource ...................................... 11-30
11.33 Server Resource > *Internals* ...................... 11-30
11.34 Statistical > Batch ................................... 11-31
11.35 Statistical > General ................................ 11-31
11.36 Statistical > Histogram ............................. 11-31
11.37 Statistical > Histogram > *Internals* .......... 11-32
11.38 Statistical > Misc ..................................... 11-32
11.39 String Operations .................................... 11-32
11.40 Switches .............................................. 11-33
11.41 Timers ............................................... 11-34
11.42 Traffic Generators .................................... 11-34
11.43 Vector Operations > General ....................... 11-35
11.44 Vector Operations > Int Matrix ................... 11-35
11.45 Vector Operations > Int Vector ................... 11-36
11.46 Vector Operations > Real Matrix .................. 11-36
11.47 Vector Operations > Real Vector .................. 11-37
11.48 Table of BONeS primitives in alphabetical order 11-52
11.49 Possible Errors During COSSAP Model Conversion 11-69
11.50 Common Errors after COSSAP Model Conversion 11-70

13.1 Definition of models in MLDesigner .................. 13-2
13.2 Advantages of Modules, FSM models, and Primitives 13-3
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.3</td>
<td>Summary of most important primitive items</td>
<td>13-9</td>
</tr>
<tr>
<td>13.4</td>
<td>Summary of items that have subitems</td>
<td>13-17</td>
</tr>
<tr>
<td>13.5</td>
<td>Summary of methods of class Error</td>
<td>13-40</td>
</tr>
<tr>
<td>13.6</td>
<td>Summary of methods of class XGraph</td>
<td>13-42</td>
</tr>
<tr>
<td>13.7</td>
<td>Summary of methods of class BarGraph</td>
<td>13-43</td>
</tr>
<tr>
<td>13.8</td>
<td>Summary of methods of class Histogram</td>
<td>13-44</td>
</tr>
<tr>
<td>13.9</td>
<td>Summary of methods of class XHistogram</td>
<td>13-45</td>
</tr>
<tr>
<td>13.10</td>
<td>Summary of String functions</td>
<td>13-46</td>
</tr>
<tr>
<td>13.11</td>
<td>Summary of Path search functions</td>
<td>13-46</td>
</tr>
<tr>
<td>13.12</td>
<td>Summary of methods of class StringList</td>
<td>13-47</td>
</tr>
<tr>
<td>13.13</td>
<td>Summary of methods of class InfString</td>
<td>13-48</td>
</tr>
<tr>
<td>13.14</td>
<td>Summary of methods of class StringListIter</td>
<td>13-49</td>
</tr>
<tr>
<td>13.15</td>
<td>Summary of methods of class SequentialList</td>
<td>13-50</td>
</tr>
<tr>
<td>13.16</td>
<td>Summary of methods of class Queue</td>
<td>13-51</td>
</tr>
<tr>
<td>13.17</td>
<td>Summary of methods of class Stack</td>
<td>13-51</td>
</tr>
<tr>
<td>13.18</td>
<td>Summary of methods of class HashTable</td>
<td>13-52</td>
</tr>
<tr>
<td>13.19</td>
<td>Summary of methods of class TextTable</td>
<td>13-53</td>
</tr>
<tr>
<td>13.20</td>
<td>Random Number Generation and Seed Value</td>
<td>13-54</td>
</tr>
<tr>
<td>15.1</td>
<td>Data structure Class Methods</td>
<td>15-13</td>
</tr>
<tr>
<td>17.1</td>
<td>Summary of methods of class DEStar</td>
<td>17-8</td>
</tr>
<tr>
<td>17.2</td>
<td>Summary of methods of class InDEPort</td>
<td>17-13</td>
</tr>
<tr>
<td>17.3</td>
<td>Summary of methods of class OutDEPort</td>
<td>17-13</td>
</tr>
<tr>
<td>17.4</td>
<td>Summary of methods of class DERepeatStar</td>
<td>17-15</td>
</tr>
<tr>
<td>18.1</td>
<td>parameters for WaveForm primitive</td>
<td>18-10</td>
</tr>
<tr>
<td>22.1</td>
<td>WaveForm capabilities</td>
<td>22-9</td>
</tr>
<tr>
<td>23.1</td>
<td>ODE solvers</td>
<td>23-10</td>
</tr>
<tr>
<td>23.2</td>
<td>ODE solver parameters</td>
<td>23-11</td>
</tr>
<tr>
<td>24.1</td>
<td>FSM Semantic Limitations</td>
<td>24-38</td>
</tr>
<tr>
<td>24.2</td>
<td>FSM Action Built-In Function Limitations (1)</td>
<td>24-39</td>
</tr>
<tr>
<td>24.3</td>
<td>FSM Action Built-In Function Limitations (2)</td>
<td>24-40</td>
</tr>
<tr>
<td>24.4</td>
<td>Data Type Limitations</td>
<td>24-41</td>
</tr>
<tr>
<td>24.5</td>
<td>RTOS Porting Functions</td>
<td>24-48</td>
</tr>
<tr>
<td>24.6</td>
<td>General Interface Configuration Functions</td>
<td>24-49</td>
</tr>
<tr>
<td>A.1</td>
<td>General Requirements</td>
<td>A-1</td>
</tr>
<tr>
<td>A.2</td>
<td>Platform Dependent Requirements</td>
<td>A-2</td>
</tr>
<tr>
<td>E.1</td>
<td>Red Hat 9 Troubleshooting</td>
<td>E-5</td>
</tr>
</tbody>
</table>
Part I

Modeling Guide
Chapter 1

First Steps with MLDesigner

1.1 Basic Terms

**Model** - An MLDesigner model describes the structure and behavior of a system or a part of the system in the real world. A model can be realized either by a hierarchy of other models, see fig. 1.1, or by programming language code and can consist of:

- Model instances
- Primitive source code
- Ports
- Relations
- Arguments

In MLDesigner the following model types are known:

- Primitive
- Module
- System
- Library

**Model instance** - A model instance is an instantiation of a model.

**Primitive source code** - The primitive source code describes the behavior of a model. It is defined by Ptolemy language source code, a simplified C++.

**Port** - Input/output ports are used to connect model instances for exchanging data objects. A port is called **formal port**, if it is part of a model, and it is known as **actual port** in case of a model instance port.

**Relation** - A relation is the connection between ports.

**Argument** - An argument is either a parameter or a shared model element.

**Parameter** - Parameters are used to control the functionality of models. **Formal parameters** of modules or primitives define the interface on embedding into other modules or systems. The cor-
responding parameters that appear in instances of the model are called actual parameters.

**Shared model element** - So-called shared model elements are used to share information without exchanging data. Shared elements are memories, events and resources. The terms formal and actual are used accordingly to formal/actual parameters and ports.

**Primitive** - A primitive is the lowest level model in MLDesigner, with functionality defined using the Ptolemy language containing C++ code fragments. The model part of a primitive defines its external interface, whereas the primitive source code written in Ptolemy defines the behavior of the primitive. Primitives can have input/output ports and arguments for interfacing. **FSM primitives** are a special type of primitive model. In contrast to normal primitives, the functional model of an FSM primitive is described by the FSM model.

**Module** - A module is a model made up of model instances. Like primitives, modules can have input/output ports as well as arguments for interfacing. A module can be a simple structure with one level or a hierarchical structure with more levels of module or primitive instances.

**System** - A system is the top-level model and consists of a number of primitive or module instances. A system model does not have any input/output ports and cannot be instantiated in other models. Defined parameters are used to parameterize the system model. The system model defines the target used for the execution of the system as well as the target parameters. Target parameters are used to parameterize the target. In MLDesigner systems are models, which are executable.

**Library** - A library is not really a model, but the MLDesigner model mechanism is used to group different models. Since the model mechanism is used for the definition of libraries, they can be
handled in the same way as all the other models. A library that is not a sub-library of any other library is called top-level library and is placed in a model base. An independent library is a library that does not contain any model that requires resources like models or data structures from another library.

**Model base** - A model base is used in MLDesigner as base for a number of dependent and independent model libraries stored in a physical directory. In MLDesigner the following model bases are known:

- MLD Examples
- MLD Experimentals
- MLD Libraries
- MLD Addons
- My Libraries
- Shared Libraries

**Simulation** - A simulation is the execution of a model containing shared model elements, modules and primitives within a system. In MLDesigner only sytems are executable models. All elements of the system must be connected or linked before the simulation can be executed. The simulation produces results which are useful for analyzing the behavior of the model and all elements contained therein.

**Target** - A target is an object that manages the execution of a simulation or code generation process. Thus, for example, in code generation, the target would be responsible for compiling the generated code and spawning the process to execute that code, if desired.

**Data structure** - A data structure in MLDesigner is a container for values of base type, vector, enumeration, or composite data type.

### 1.2 Graphical User Interface

When MLDesigner opens you will see the MLDesigner’s graphical user interface (GUI) containing the following elements, refer fig. 1.2:

1. **Menu Bar** - with entries for the **File**, **Edit**, **View**, **Window**, and **Settings** menu.

2. **Toolbars** - The lower group of toolbars is extended when a model is opened in the Model Editor. Active toolbar icons depend on the type of model selected in the Model Editor and whether MLDesigner is in simulation mode or edit mode. In simulation mode run control icons are visible on the toolbar. All toolbars are explained in detail in sec. 2.2.

3. **Tree View History** - A combo-box containing a list of models last opened or used as instances in other models. The number of models displayed in the list is determined in the Settings Dialog under the **History** item of the Tree View category.

4. **Tree View** - The window with tabs to select the four views **File** (physical structure of model base), **Library** (logical structure of model base), **Model**, and **Search** (use the right mouse button to activate the context menu).
Figure 1.2: The MLDesigner Graphical User Interface.
1.2 Graphical User Interface

The Tree View is used to create, open, and maintain your libraries, primitives, files, etc. (For detailed information about these views see sec. 2.2).

Throughout this manual, the Library tab is used to navigate the libraries of primitives and demos unless otherwise indicated. To see the physical location of the respective model, hold the cursor over the entry in the Library tab and wait a second or so for the tool tip text to display. The Library tab of the Tree View is also called Library View whereas the file tab is usually called File View.

5. **Parameter Set Window** - Here you can create a new parameter set. You can run simulations using preset values from more than one parameter set.

   An example is described in sec. 7.4 in the Modeling Guide of MLDesigner Manual.

6. **Property Editor** - This is where you change or define the default properties of models and ports. [P]arameters and [T]argets are given values, and information about the model or model instance can be entered such as description and short description. The active tab depends on the type of model instance selected in the Model Editor. When in Simulation Mode with no system elements selected in the Model Editor, the Simulation Properties tab is active.

7. **Model Editor** - That is your main work area where you build primitives, modules and systems. Model Editor windows are arranged within the so-called modeling workspace area. You can maximize your Model Editor windows so that the whole modeling workspace area is occupied by exactly one Model Editor window.

8. **Data Structure Editor** - Here you can edit user defined data structures. Editable data structures have a green icon exactly the same shape as the red icon. You must open a system, module, primitive or library with write access before you can create a new data structure or edit members of a data structure.

9. **Data Structure Member Editor** - It is part of the Data Structure Editor for creating, deleting and modifying members of existing data structures.

10. **Console Window** - The Console Window has two tabs when you are in edit mode:

    - The Command Console where Tcl and PTcl commands can be entered.
    - The Log Console where a record of your actions is displayed and error messages or warnings are printed. You can save the contents of the Console to a file or clear the Console by selecting the appropriate menu option from the context menu.

Further two tabs appear when you are in simulation mode:

    - The Breakpoints Console where you see a list of all breakpoints. Breakpoints are useful for debugging systems and the list of breakpoints makes it easier for you to find and edit them. You can add and remove breakpoints while in simulation mode. To do so select Add Breakpoint or Remove Breakpoint from the context menu over the relevant entry in the Breakpoints Console.
    - The Progress Window where you can monitor the compile process and the progress of the simulation measured as a percentage value.

Another tab, the Animation tab, appears if the Textual Animation icon on the toolbar is checked. The port and model instance firings are printed to the console in this mode and simulation is a lot slower.
1.3 Run a Demo Simulation

MLDesigner is shipped with a large selection of demo systems. There are a number of links to demos in the Library View under MLD Examples. When you expand the + to the left of each library item, you see an entry for all the modules contained within the system as well as an entry for the system itself. Observe the different icons for the various elements i.e. Library, System, Module, and Primitive.

Many more demos can be found in MLD Libraries/Demos sorted into their categories according to domain. The following example demonstrates the procedure to follow if you want to run a simulation. Once you are familiar with the steps needed to open and run simulations from these libraries, you can explore the Demos.

NOTE: The File View shows the physical structure of MLDesigner’s libraries where the Library View shows the logical structure of these libraries. The logical structure is merely a structure of links to the physical location where the model element is saved. It is possible to have a System appear in the Library View in more than one location with the same logical name or in the same location with a different name. You may however not have models with the same physical name in the same location.

For demonstration purposes we will choose a system, copy it to a user library, run a simulation, change some parameters, perform another simulation and compare the results.

1.3.1 Choose a System

Proceed as follows:

1. Select the Library tab in the Tree View window. Expand MLD Libraries/Demos/DE Demo/ Basic/ by clicking the + to the left of each library in the Tree View window. The system testPacket shown in fig. 1.3 is an excellent example where we can see some of the features offered by MLDesigner.

2. Double click the system to open the XML model in the Model Editor Window.

You will notice the values in the System Properties in Property Editor window (bottom left) are not editable since the model is write-protected. When you click on a model instance in the Model Editor Window, the Instance Properties tab is activated in the Property Editor window with all parameter values being grayed out. It is possible to run the simulation here with the default settings. We want to change some parameters so you need to copy the system to a user library where you have write permission. The first step is to create a top level user library as a container to hold your work.

1.3.2 Create a Library

The standard libraries shipped with MLDesigner are write-protected, i.e., they are read-only and you can only open them, but it is not possible to extend them. You must create a container where
1.3 Run a Demo Simulation

**Figure 1.3: testPacket System**

you can save copies of existing primitives, modules or systems or alternatively create new models in a read/write environment.

**Figure 1.4: The New Model dialog**

To create a new library proceed as follows:

1. Click **File** item from the main menu bar and choose **New** or click the corresponding tool button **New Model**. A Create New Model dialog is displayed, see fig. 1.4.

2. Choose **Library** from the **Type of Model** drop-down menu. If the Create New Model dialog pops up the first time, library should be selected already as default.
3. Specify the name of your library by setting **Logical Name** to *First Steps*. The logical name can contain any character in contrast to the physical name.

4. Leave the **Library** field blank to create a top level library.

5. The **Physical Name** field is set automatically when you click in the next input field. If not given manually, the physical name is derived from logical name by replacing all characters different from letter, digit, or `.` with an `.`. You can give the library a different physical name to the logical name. The logical name appears in the Library View window and the physical name in the File View window.

6. The **Physical Location** field is set automatically to `$MLD_USER`.

7. Leave the **Group Permissions** and **World Permissions** as they are set by default. Using these check boxes, you can normally control the access to your models for members of your group and other users.

8. You have the opportunity to type a brief description for your library in the **Description** input field. This text appears in the tool tip text when the cursor is placed over the Library in the Tree View window and as an introduction for the online documentation which is automatically generated.

9. Click **OK** button to create the library.

### 1.3.3 Save Demo in the Created Library

The next step is to save the open demo system in the newly created library. Click on the grey background of the Model Editor with the open *testPacket* system. All menu options now apply to the active Model Editor window. Activate the **File** menu on the menu bar and proceed as follows:

1. Choose **Save As**. A **Save As New Model** dialog appears.

2. You need not change anything in this dialog. You must only select a library to save the system in. Click the icon next to the input field **Library**. Choose the library *First Steps* as a location to save the system. If you select a write protected location, an error message is displayed.

3. Click **OK** button to save the model in the selected library.

4. The title bar of the Model Editor window changes to show the new location as `$MLD_USER/First Steps/testPacket`.

### 1.3.4 Explore and Run the Demo

You will notice that a number of fields in the **System Properties** in Property Editor window (bottom left) are now editable, i.e., the text is no longer grey and italic. On the model level you can change a number of settings like common model properties, descriptions, linked objects, target parameters as well as set the default values for parameters.

Select the model instance *Ramp#1* in the Model Editor window with a single mouse-click. Notice the **Instance Properties** tab becomes active in Property Editor window showing the settable
1.3 Run a Demo Simulation

parameters value and step of the model instance Ramp#1. A double click on the model instance opens the primitive at its original location. The title bar of the primitive Model Editor window shows where the primitive is physically saved.

Notice that it is not possible to edit any property in the **Primitive Properties** tab of the built-in primitive. Click on the input port to see the port properties. The type of data consumed by the primitive is determined by the port type. You cannot change the characteristics of the built-in primitive, but only the way the primitive behaves when the simulation is run.

Close the primitive Model Editor window and click the background of the testPacket system in the Model Editor window. Click on the **Switch to Simulation Mode** button on the toolbar.

To run the simulation using the default settings click the **Go** button. The system is compiled and executed and an XMgraph titled “Packetized signal output” displays showing the original signal compared to the packetized signal, see fig. 1.5. The total Run Length is set to 50 with a Step Increment of 2. Click on each model instance and look at the parameters that influence the output of this system.

The model instance Packetize#1 waits until it has 5 events (maxLength) before firing. Notice that the model instance VarServer#1 with serviceTime set to 1.0, can process 50 samples in packages of 5 defined by the Packetize#1 model instance parameter maxLength.

![Packetized signal output](image)

**Figure 1.5:** Comparative Packetized/Non Packetized Output of testPacket

We now want to change some parameters and run the simulation again.

1. Click on the gray background of the system. Adjust the property Run Length to 100 in the **Simulation Properties** tab of the Property Editor (bottom left).
2. Click on the model instance Packetize#1 and set the parameter maxLength to 10 in the **Instance Properties** tab of the Property Editor.
3. Click the **Go** button on the upper toolbar. The demo will start to execute.

Notice the difference in server time as a proportion of the total simulation time of the original unpacketized output.
There are a few simulation control buttons on the toolbar that are not used because the simulation is very quick. These icons are useful for longer simulation runs.

1. To interrupt the demo click the **Pause** button. Restart by clicking **Continue**. In interactive mode the option to proceed one **Step** at a time is available. This is a useful feature, in conjunction with Graphical Animation, when debugging systems or primitives.

2. To stop a simulation running click the **Early End** button. This calls the *wrapup* method (see sec. 13.5.2) of all primitive models and all post processor results are displayed if enough information has been generated and is available to generate the graphs or files (sinks) that are present in the system. The **Abort** button ends the simulation and resets all default settings in the Run dialog.

### 1.4 Build a Simple Model

This section will show you a variety of methods to create a simple model and run a simulation. You should have created a library called *First Steps* which contains the copied demo system *testPacket*. If not please go to sec. 1.3.2 and create a Library.

The main purpose of this exercise is to familiarize you with the basic functions of the different elements of MLDesigner’s graphical user interface.

#### 1.4.1 Create a Sub-library

As mentioned above a library in MLDesigner is used to hold all kind of models like systems, modules, primitives, and sub-libraries. You can create a sub-library called *MySubLibrary* as follows:

1. Click **File** from the main menu bar and choose **New** or click the corresponding tool button **New Model**. A Create New Model dialog is displayed.
2. Choose **Library** from the **Type of Model** drop-down menu.
3. Specify the name for your library by setting **Logical Name** to *MySubLibrary*. The logical name can contain any character in contrast to the physical name.
4. Click the icon to the right of the **Library** field. Select *First Steps* from the selection dialog. The new library becomes a sub-library of the one selected.
5. The **Physical Name** field is set automatically when you click in the next input field. If not given manually, the physical name is derived from logical name by replacing all characters different from letter, digit, or ‘.’ with an ‘.’. You can give the sub-library a different physical name to the logical name. The logical name appears in the Library View window and the physical name in the File View window.
6. The **Physical Location** field is set automatically to $MLD_USER/First_Steps$.

**NOTE:** A sub-library is created by default as a sub-directory in within the directory of the library to which it belongs. That is, the physical hierarchy of libraries corresponds
1.4 Build a Simple Model

always to directory structure. Even though you can change the physical location, we do not recommend to change it.

7. You have the opportunity to type a brief description for your library in the **Description** input field. This text appears in the tool tip text when the cursor is placed over the Library in the tree view window and as an introduction for the online documentation which is automatically generated.

8. Click **OK** button to create the library.

![Figure 1.6: New Model - Library Menu Item](image)

Another possibility to create a sub-library is to right-click on a existing library item in the Tree View either in Library tab or File tab. Select the **New Model** context menu item. A sub-menu opens by which you can invoke the creation of a model of given type in the library, see fig. 1.6. Select the **Library** menu item to open the Create New Model dialog. The advantage of using the context menu on an item in the Tree View to create a new model within a library is that all of the relevant information is filled in the according input fields. You have only to specify the name of the new model.

### 1.4.2 Build a Module

In this example we create a module that adds two input signals and scales the result by a constant factor.

#### 1.4.2.1 Create a New Module

Proceed as follows:
1 First Steps with MLDesigner

1. Click File from the menu line and choose New or click the tool button New Model (fig. 1.4).
2. Choose Module from the Type of Model drop-down menu.
3. Specify the name for your module by setting Logical Name to MyAdderModule.
4. Click the icon to the right of the Library field. Select MySubLibrary from the selection dialog.
5. From the drop down menu Modeling Domain select SDF.
6. The Physical Name field is set automatically. You can change it if you want.
7. The Physical Location field is set automatically to $MLD_USER/First_Steps/MySubLibrary.

NOTE: A module is created by default as a sub-directory in within the directory of the library to which it belongs. Even though you can change the physical location, we do not recommend to change it.

8. You have the opportunity to type a brief description for your module in the Description input field. This text appears in the tool tip text when the cursor is placed over the module in the Tree View window and as an introduction for the online documentation which is automatically generated.
9. Click OK to save the module in your library. When creation of the model is finished, the new module will open automatically in a new Model Editor window as a grey square.

As for creating a new library, you can also create a module quickly by selecting the New Model - Module context menu item after right-clicking on the library item in the Tree View.

1.4.2.2 Add Ports

You can create an arbitrary number of input and output ports. To create ports either use the appropriate tool buttons Add Input Port and Add Output Port, or use the shortcuts I for input port and O for output port. Alternatively click Edit on the main menu bar and select the appropriate options.

For this example we need two input ports and one output port:

1. Select the Add Input Port button from the toolbar.

NOTE: Your pointer has switched to a different mode. Every time you click in the grey rectangle of your module in the Model Editor window which represents the bounding box of your model, a port will be generated.

2. Click on the side of the grey rectangle (bounding box) in the Model Editor window to create a new input port. It will be put near your cursor location. A second click creates a new input port. You may have to drag the ports to a suitable position if they are on top of each other for model instance. The ports Input1 and Input2 have been created.

3. Switch back to the default pointer mode by right mouse click or by clicking the Select Tool icon.
1. You can drag the newly created port and move it anywhere in your module view using the left mouse. The grey rectangle resizes automatically.

5. Select the Add Output Port button from the toolbar.

6. Click on the side of the grey rectangle (bounding box) in the Model Editor window to create a new output port.

Another method to create new ports is to press <Control> and then click on an existing port you wish to copy. You can now drag and drop the port you wish to duplicate.

When using the context menu option Add Input/Output Port your mouse pointer will not change and one new port will be placed in the Model Editor window. The Edit menu works in the same way as the context menu.

We now need to change the properties of each port.

1. Click on the first input port Input1. The Property Editor (in the lower left of your MLDesigner window) now displays the Port Properties of your port Input1.

2. Next to Data Type click on the anytype entry and change it to float.

3. You can change the name from Input1 if you like.

4. Repeat the procedure for Input2 and Output1 changing the Data Type to float for all ports.

1.4.2.3 Add Model Instances

The next step is to create the model instances of required primitives of the system by dragging them from the Tree View into the module. The module MyAdderModule should be open in the Model Editor window and appears as a grey square with two input ports and one output port.

Continue as follows:

1. Go to the Tree View window.

2. Switch to the Library tab and press the icon to the left of MLD Libraries. Expand the library SDF Domain and the sub-library Arithmetic. You should see now the primitive item Add.

3. Click on the Add primitive item with the left mouse and drag the primitive into the grey square of the module MyAdderModule.

4. A Select Special Primitive dialog appears. Here you can select how many input ports you want the Add primitive has. This is possible because the Add primitive in the SDF domain can have an arbitrary number of input ports represented by a so-called multi-port. Such a primitive can be specialized by specifying a certain number of input ports. Such
specializations are called **special primitives**. Please refer sec. 2.3.3 for detailed information about special primitives and how to create them.

The `Add` primitive has a number of such specializations, select the entry `Add.input=2` and click **OK**.

5. You now see the model instance `Add` with two input ports and one output port in the Model Editor Window. The label reads `Add.input=2#1` indicating this is model instance #1 of `Add.input=2`.

The next step is to add the primitive `Gain` to your system. Proceed as follows:

1. Click the `+` icon of the library `SDF/Arithmetic` library and select the `Gain` primitive in the list.

2. Drag and drop this primitive into the system Model Editor Window so that its input port is near the output port of the model instance `Add.input=2#1`.

You now have all the elements needed for the module. The next phase is to connect the model instances.

### 1.4.2.4 Connect Model Instances

You have two options when connecting ports. The first option is to double click the formal input port `Input1` of the module. The pointer changes to connection mode. Move the cursor towards the first actual input port of the model instance `Add.input=2#1`. A line is traced behind the cursor. You can go straight to the input port and complete the connection by a single mouse-click on the port or you can draw a line with 90 degree corners. Move the cursor in connection mode till a straight line appears and you are in the correct position where you would like to insert a 90 degree turn. Click the left mouse once and move the cursor in the direction you want the line to move. A single left click places an anchor or node where you can turn the corner. A single right click reverts the connection back by one node. A single click on a port completes the connection and the cursor returns to normal selection mode. If you want to revert back to normal selection mode before a connection is complete, right click the mouse until all nodes are deleted and the cursor image changes back to normal. Connect the formal input port `Input2` with the second input of the model instance `Add.input=2#1`, the output port of `Add.input=2#1` with the input port of model instance `Gain#1`, and the output of `Gain#1` with the formal output port `Output1`.

You can also click once on the **Connect** icon and proceed exactly as described with the first method. A single click of the mouse creates a node where you can draw a 90 degree corner and a single click on a port completes the connection. In this case however, the cursor is still in connection mode. To change back to the normal selection tool click the **Select Tool** icon in the toolbar or click on the model background with the right mouse button.

Your result should be similar to the one shown in fig. 1.7.

**NOTE:** Notice the drag corners shown if a model instance like `Gain#1` is selected. You can grab a corner to resize the graphical representation of the model instance in the Model Editor window.
1.4 Build a Simple Model

1.4.2.5 Add Parameter

We now need to add a parameter to the module. A parameter is used as a fixed value to change the behavior of the model without recompiling it.

1. Make sure no port or model instance is selected in the module window. If needed, click somewhere in the module window, without selecting any elements, to switch the property editor back to the Module Properties. The Property Editor should have the first entry named Logical Name with the name of your module MyAdderModule.

2. Right-click with the mouse somewhere in the Property Editor. Use the New Parameter context menu item. The new parameter (indicated with a P icon) is now listed on the bottom of the list. Click on the + icon to extend the parameter.

3. Click on the Name row and change the name of the parameter to ScaleFactor.

4. Change the Data Type row to float.

5. Change Value to 1.0

6. All other fields need not be changed.

1.4.2.6 Link Parameter

The next step is to link the actual parameter gain of the model instance Gain#1 to the formal module parameter ScaleFactor. Click once on the model instance Gain#1 to activate the Instance Properties tab in the Property Editor window. Activate the context menu in the gain field at the bottom of the Instance Properties and choose Link to.. context menu item to open a list of all parameters labeled with an P icon and global parameters labeled with a G icon. Choose ScaleFactor from this list. A green arrow is inserted in this field indicating you have successfully linked the actual parameter of a model instance to a formal parameter on the module (or system) level.

Before closing the Model Editor Window, save the module by clicking on the Save button in the toolbar.

1.4.3 Build a System

In this example we create a system with two sources, the module MyAdderModule to add the two inputs from these sources, and a sink to plot the results of the addition.
1.4.3.1 Create a New System

Proceed as follows:

1. Click **File** from the menu line and choose **New** or click the tool button **New Model** (fig. 1.4).
2. Choose **System** from the **Type of Model** drop-down menu.
3. Specify the name for your system by setting **Logical Name** to **MyAdderSystem**.
4. Click the icon to the right of the **Library** field. Select **MySubLibrary** from the selection dialog.
5. From the drop down menu **Modeling Domain** select **SDF**.
6. The **Physical Name** field is set automatically. You can change it if you want.
7. The **Physical Location** field is set automatically to $MLD_USER/First_Steps/MySubLibrary.

**NOTE:** A system is created by default as a sub-directory in within the directory of the library to which it belongs. Even though you can change the physical location, we do not recommend to change it.

8. You have the opportunity to type a brief description for your system in the **Description** input field. This text appears in the tool tip text when the cursor is placed over the system in the Tree View window and as an introduction for the online documentation which is automatically generated.

9. Click **OK** to save the system in your library. When creation of the model is finished, the new system model will open automatically as a grey square in a new Model Editor window.

As for creating a new library, you can also create a system quickly by selecting the **New Model - System** context menu item after right-clicking on the library item in the Tree View.

1.4.3.2 Add Model Instances

The next step is to create the model instances of required primitives and modules of the system by dragging them from the Tree View into the system. The empty system **MyAdderSystem** should be open in the Model Editor window and appears as a grey square.

Continue as follows:

1. Go to the Tree View window.
2. Switch to the Library tab and press the [+] icon to the left of **My Libraries. Expand the library First_Steps/MySubLibrary and select the module MyAdderModule that we have built in sec. 1.4.2.
3. Drag and drop this module into the system Model Editor Window.

The next step is to add sources to your system. Sources are models that do not have input ports, but one or more output ports at which they generate signals or events. Proceed as follows:
1. Click the [+] icon of the library SDF/Sources library and select the SawGen primitive in the list.

2. Drag and drop this primitive into the system Model Editor Window. You need two sources, so drag the SawGen primitive into the window again or use the Control key with left mouse-click combination while dragging the first model instance Add.input=2#1 in the Model Editor Window. Place the two primitives so that their output port is near the input ports of the model instance MyAdderModule.

It is not possible to run a simulation unless the output of the system is processed by a sink primitive. The Sinks library contains a collection of primitives that do exactly that. There are a number of ways to deal with output data resulting from a simulation. The data can be:

- printed to a console or to a file,
- displayed in a graph of type Xgraph, XMgraph, XYgraph or TkBargraph or
- consumed by a BlackHole if the data is not useful.

Click the [+] of the library SDF/Sinks to see the graph primitives. Search for a primitive called Xgraph and drag and drop it into the system Model Editor Window.

You now have all the elements needed for the system. The next phase is to connect the model instances.

### 1.4.3.3 Connect Model Instances

Double click the output port of the first model instance SawGen#1. The pointer changes to connection mode. Move the cursor towards the input port of the model instance MyAdderModule#1. A line is traced behind the cursor. You can go straight to the input port and complete the connection by a single mouse-click on the port or you can draw a line with 90 degree corners as described in sec. 1.4.2.4. Connect the second model instance SawGen#2 and the second input of the MyAdderModule#1 and the output of MyAdderModule#1 with the input of the Xgraph#1.

Your result should be similar to the system shown in fig. 1.8.
1.4.3.4 Export Parameters

In sec. 1.4.2.5 and sec. 1.4.2.6 we created a new formal parameter and afterwards we linked the actual parameter of a model instance to it. Instead of this we can do it in only one step by using the export mechanism. We want to link parameter ScaleFactor of model instance MyAdderModule#1 to a new created system parameter with the same name. Proceed as follows:

1. Click once on the model instance MyAdderModule#1 to activate the Instance Properties tab in the Property Editor window.
2. Activate the context menu in the ScaleFactor field and choose Export.

The next step is to export parameter Increment of model instance SawGen#1 as system parameter with a different name:

1. Click once on the model instance SawGen#1 to activate the Instance Properties tab in the Property Editor window.
2. Activate the context menu in the Increment field and choose Export as...
3. Change the value of the field to SawIncrement and click the OK button.

Last we want to link parameter Increment of model instance SawGen#2 to the new created system parameter SawIncrement:

1. Click once on the model instance SawGen#2 to activate the Instance Properties tab in the Property Editor window.
2. Activate the context menu in the Increment field and choose Link to... context menu item to open a list of all parameters labeled with an P icon, global parameters labeled with a G icon, and target parameters labeled with a T icon in the system.
3. Choose SawIncrement from this list.

Now, you can run the simulation. Click on the Simulate button in the toolbar. You can use the default run settings seen in the Simulation Properties window (bottom left), or change the Run Length entry to increase or decrease the run time. Press the Go button to run the simulation.

1.4.4 Multiple Iterations and Parameter Sets

It is possible to run simulation permutations using a combination of multiple iterations and a number of parameter sets.

We now want to run a simulation with two iterations. Click the Switch to Edit Mode button in the toolbar before proceeding. Make sure no port or model instance is selected in the system window. If needed, click somewhere in the system window, without selecting any elements, to switch the property editor back to the System Properties. Click on the ScaleFactor row at the bottom of the System Properties and enter 1.0; 2.0. Start the simulation by clicking the Go button after switching back to Simulation Mode. Two graphs display after the simulation is complete. You can add more values separated by a semicolon and run the simulation again. Every entry indicates
a new iteration or the whole simulation run.

The next step is to add a new parameter set. If you are still in the simulation mode, click the **Switch to Edit Model** button in the toolbar to leave to the edit mode. Click the left mouse button while the cursor is positioned in the grey bounding box of the system in the Model Editor Window to ensure no model instances are selected. A small window between Tree View window and the Property Editor Window is visible. Place the cursor in the window titled **Parameter Set 1** and activate the context menu. Choose **New Parameter Set**. The title of this window changes to **Parameter Set 2**. All the settings in the **System Properties** tab remain the same as in **Parameter Set 1**. Change the value field of parameters **ScaleFactor** to 3.0 and **SawIncrement** to 0.5; 0.25. Save the system and run the simulation again. You now see four graphs. The result of this simulation should be self-explanatory.

### 1.4.5 Xgraph Configuration

We now want to take a brief look at a few features regarding Xgraph configuration. The full range of options are covered in sec. 8.3. It would be useful to have x and y axis labels. Click on the **Xgraph#1** block in the Model Editor Window. In the **Instance Properties** click the **options** field and click the icon right to this input field to open the **Value Editor**. Append `-x` Step `-y` Value to the existing value in the Value Editor. It could also be useful having some output datasets plotted in one Xgraph. The parameter **Cumulation** defines how datasets are stored in Xgraph displays.

- **None**: Each output is plotted in a new Xgraph.
- **Iterations**: All iterations of a parameter set are collected and plotted in a single Xgraph.
- **Paramsets**: Outputs from iterations and paramsets are collected and plotted in a single Xgraph.

Set the **Cumulation** parameter to **Paramsets**. You may also want to have a grid pattern as background of the plot. Remove the `-tk` entry from the options dialog. Save the model and run the simulation. The result should be similar to fig. 1.9.

In the SDF/DDF domain the **Xgraph** has an additional parameter called **EndCondition**. If this parameter is set to **YES**, the **Xgraph** can tell the simulation scheduler to stop the simulation as soon as it has received the defined number of data given by parameter **NumberOfItems**. If more than one **Xgraph** block has set this parameter **EndCondition** to **YES**, the simulation stops once all **Xgraph** blocks have the required number of data.

### 1.4.6 Build a Primitive

This exercise takes you through the steps required to create a new primitive. The primitive interface can be defined using MLDesigner with no programming experience. To define the functionality of the primitive you need experience in programming with C++.

In this example we create a primitive with the same functionality as the module **MyAdderModule** we built in the previous example.
1 First Steps with MLDesigner

1.4.6.1 Create a New Primitive

Proceed as follows:

1. Click **File** from the main menu bar and choose **New** or click the tool button **New Model**.
2. Choose **Primitive** from the **Type of Model** drop-down menu.
3. Specify the name for your system by setting **Logical Name** to **MyAdderPrimitive**.
4. Click the button next to the **Library** field to browse available libraries. Select **MySubLibrary** in the selection dialog and press **OK**. If you choose a library that is Read only, an error message will be displayed.
5. Choose **SDF** from the **Modeling Domain** drop-down menu.
6. Enter a description for your primitive in the **Description** field. The description appears in the online documentation.
7. It is possible to derive a primitive from another primitive to inherit its interface including ports, parameters, a.s.o as well as its function to extend the derived primitives with additional interface elements and functions. For this example we will not derive from any other primitives.
8. The **Physical Name** field is set automatically when you click in the next input field. If not given manually, the physical name is derived from logical name by replacing all characters different from letter, digit, or ‘.’ with an ‘.’.
9. The **Load Mode** option **Dynamic** means the primitive is compiled and reloaded before every simulation. That is, changes made to the primitive will take effect immediately when the system is recompiled for the next simulation run. **Load Mode Permanent** on the other hand means that once a primitive is compiled and loaded, e.g., for the first simulation run, changes will not take effect before MLDesigner is restarted.

**NOTE:** If you want to derive another primitive from this one, it is necessary to load it
permanent.

10. Click the OK button to create the primitive. When creation of the primitive is finished, the new primitive model will open automatically as grey square in a new Model Editor window.

A primitive model has been created and you now need to define its functionality.

### 1.4.6.2 Create Ports and Parameters

You can create an arbitrary number of input and output ports. To create ports either use the appropriate tool buttons Add Input Port and Add Output Port, or use the shortcuts \( \texttt{I} \) for input port and \( \texttt{O} \) for output port. Alternatively click Edit on the main menu bar and select the appropriate options.

For this example we need two input ports and one output port:

1. Select the Add Input Port button from the toolbar.
   
   **NOTE:** Your pointer has switched to a different mode. Every time you click in the grey rectangle of your primitive in the Model Editor window which represents the bounding box of your model, a port will be generated.

2. Click on the side of the grey rectangle (bounding box) in the Model Editor window to create a new input port. It will be put near your cursor location. A second click creates a new input port. You may have to drag the ports to a suitable position if they are on top of each other for model instance. The ports Input1 and Input2 have been created.

3. Switch back to the default pointer mode by right mouse click or by clicking the Select Tool icon.

4. You can drag the newly created port and move it anywhere in your primitive view using the left mouse. The grey rectangle resizes automatically.

5. Select the Add Output Port button from the toolbar.

6. Click on the side of the grey rectangle (bounding box) in the Model Editor window to create a new output port.

Another method to create new ports is to press \(<\text{Control}>\) and then click on an existing port you wish to copy. You can now drag and drop the port you wish to duplicate.

When using the context menu option Add Input/Output Port your mouse pointer will not change and one new port will be placed in the Model Editor window. The Edit menu works in the same way as the context menu.

Your primitive should now look like the one in fig. 1.10. We now need to change the properties of each port.

1. Click on the first input port Input1. The Property Editor (in the lower left of your MLDesigner window) now displays the Port Properties of your port Input1.

2. Next to Data Type click on the anytype entry and change it to float.
3. You can change the name from Input1 if you like.
4. Repeat the procedure for Input2 and Output1 changing the Data Type to float for all ports.

![Simple Adder Primitive](image)

**Figure 1.10: Simple Adder Primitive**

We now need to add a parameter to the primitive. A parameter is used as a fixed value to change the behavior of the primitive without recompiling it.

1. Make sure no port is selected in the primitive window. If needed, click somewhere in the primitive window, without selecting any elements, to switch the property editor back to the Primitive Properties. The Property Editor should have the first entry named *Logical Name* with the name of your primitive *MySimpleAdder*.

2. Right-click with the mouse somewhere in the Property Editor. Use the *New Parameter* context menu item. The new parameter (indicated with a $\text{P}$ icon) is now listed on the bottom of the list. Click on the $\text{+}$ icon to extend the parameter.

3. Click on the *Name* row and change the name of the parameter to *ScaleFactor*.

4. Change the *Data Type* row to *float*.

5. Change *Value* to 1.0

6. All other fields need not be changed.

Before adding functionality to the primitive, save it by clicking on the *Save* button in the toolbar.

### 1.4.6.3 Change the Source

The new primitive is open in the Model Editor Window. Model elements that are created in the Model Editor define the interface of the primitive. The function of a primitive is defined in a separate file, the so-called primitive source. To edit the source, click on the *Open Source* button in the toolbar, select menu option from context menu or use the shortcut S. This will open a new window, the Primitive Source Editor, where you can edit the primitive source code.

**NOTE:** The source that is created automatically when you create a new primitive already contains templates for the most important primitive source items.

**NOTE:** Some of these items are generated directly from the interface definition developed in the Model Editor window. Changes to these items are lost when you save the primitive model next time.
1.4 Build a Simple Model

For this example only the go entry must be changed. Scroll down the source or use the search function to find the go entry. Edit the entry to read as follows:

```go
{
    Output1%0 << ScaleFactor * (double(Input1%0) + double(Input2%0));
}
```

This implements the adding functionality and makes the primitive work. Click on the Save button to save the source and then on the Compile button (the rightmost icon on the toolbar) or select Compile Source from the context menu.

If everything was typed in correctly, the compile will be done without error messages. If the compiler shows errors, they will appear in the console window at the bottom of the screen. Read them carefully to analyze what has gone wrong. Check your source code and make sure the names and properties of the ports and the parameters have been set correctly. Clicking on an error message will position the cursor at the beginning of the line where the error happened.
Chapter 2
Modeling with MLDesigner

For a description of the terms used throughout this manual, please refer to sec. 1.1.

2.1 Understanding Environment Variables

The model bases MLDExamples, MLD Experimentals, MLD Libraries, and MLD Addons are linked to fixed sub-directories in the installation directory of MLDesigner which are write-protected. However, the model bases My Libraries and Shared Libraries can be linked to any physical directory by setting the environment variables MLD_USER and MLD_SHARED. $MLD_USER normally points to a directory that serves as working directory for your modeling work. Normally, you have write permission to this directory and save all your own model libraries in this directory. $MLD_SHARED points to a directory where you share models with other users.

There are two different possibilities to set these environment variables:

1. You can define them outside MLDesigner by defining eponymous environment variables before starting MLDesigner using an according system command.
2. You can set them inside MLDesigner using the Environment item of the MLD category in the Settings Dialog.

**NOTE:** These settings are temporary for the active MLDesigner session and are lost when you start MLDesigner next time.

Setting these environment variables can be used to organize a large number of model libraries into a number of independent working directories or to access to a completely different directory temporarily, e.g., the working directory of another user.

If $MLD_USER is not defined, $HOME/MLD is used as default. $MLD_SHARED is not available in MLDesigner if it is not set or points to the same directory as $MLD_USER.

There are two further environment variables $MLD and $MLDARCH that specify the directory where MLDesigner is installed and the architecture of the system on which MLDesigner is running, respectively. Both variables can neither be set outside nor inside MLDesigner.

All used environment variables are printed to the command shell when you start MLDesigner, e.g.: 
Environment Variables
MLD        /opt/mld25r00
MLD_USER   /home/gs/MLD
MLD_SHARED /home/gs/MLD
MLDARCH    i386-linux-gcc296

2.2 Graphical User Interface

2.2.1 User Interface Structure

MLDesigner’s graphical user interface (GUI) provides an environment for developing and simulating hierarchical system models on varying levels of abstraction. This can be done graphically or textually. The built-in PTcl interpreter is integrated into the MLDesigner GUI (Command Console) and accepts PTCL commands for text-based modeling. See ch. 9 in the MLDesigner manual.

The main MLDesigner’s GUI elements are described in sec. 1.2.

2.2.2 Settings

User preferences vary a lot. As a result it is possible to change the defaults for a variety of Graphical User Interface settings. Click the main menu option Settings to open the dialog in fig. 2.1.

The following table contains a brief description of all options.

Figure 2.1: Settings Dialog

The items are listed in the order they appear in the dialog from top to bottom.
### 2.2 Graphical User Interface

#### MLD

- **Start as MDI application** - The graphical user interface is a Multi-Document Interface (MDI) when this option is checked. In non-MDI mode, each window is opened as a separate entity and can be handled separately making window management a lot more flexible.
- **Reopen models at startup** - All open models will be re-opened when MLDesigner is shut down and restarted.
- **Tooltips** - Turns all tooltip texts off when the tick is removed from this check box.
- **Tab size** - The tab size for all text editors.
- **Working directory at startup** - The default path of the command console is set with this variable. This setting becomes effective after shutting down and restarting MLDesigner.
- **GUI font at startup** - The font for all graphical user interface elements. You must shut down and restart MLDesigner before the setting takes effect.
- **Number of last opened models** - Sets the number of models shown in the history menu under the **File** main menu option.
- **Default HTML browser** - The browser you want to use when contacting support via the **Report Problems** option under the **Help** button in the top right corner of the GUI.

#### Environment

- **$MLD_USER** - The model base *My Libraries* points to the physical directory registered in this field. Changes to this setting are only valid for the current modeling session.
- **$MLD_SHARED** - The model base *Shared Libraries* points to the physical directory registered in this field. Changes to this setting are only valid for the current modeling session.
- **$MLD_TEMP** - Specifies the folder where the temporary files produced by MLDesigner are stored.

<table>
<thead>
<tr>
<th>Dialog Option</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MLD</strong></td>
<td><strong>General</strong></td>
</tr>
<tr>
<td></td>
<td>- <strong>Start as MDI application</strong> - The graphical user interface is a Multi-Document Interface (MDI) when this option is checked. In non-MDI mode, each window is opened as a separate entity and can be handled separately making window management a lot more flexible.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Reopen models at startup</strong> - All open models will be re-opened when MLDesigner is shut down and restarted.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Tooltips</strong> - Turns all tooltip texts off when the tick is removed from this check box.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Tab size</strong> - The tab size for all text editors.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Working directory at startup</strong> - The default path of the command console is set with this variable. This setting becomes effective after shutting down and restarting MLDesigner.</td>
</tr>
<tr>
<td></td>
<td>- <strong>GUI font at startup</strong> - The font for all graphical user interface elements. You must shut down and restart MLDesigner before the setting takes effect.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Number of last opened models</strong> - Sets the number of models shown in the history menu under the <strong>File</strong> main menu option.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Default HTML browser</strong> - The browser you want to use when contacting support via the <strong>Report Problems</strong> option under the <strong>Help</strong> button in the top right corner of the GUI.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tree View</th>
<th>Confirmations</th>
</tr>
</thead>
</table>
Every action in the following list can be performed with or without a confirmation screen. By default all options are active. When one of these actions are performed a confirmation window displays. You are asked if you really want to remove, delete, restore, move, or copy the model. The option **Do not show this message again** does exactly that. One click in the check box and the confirmation screen is not displayed again for the action you were performing. It is possible to reactivate this option in the **Settings** dialog.

The following options can be set:

- **Remove** - Specifies whether a warning message box is shown before a reference to a model is deleted in the library tree view.
- **Delete** - Specifies whether a warning message box is shown before a model or file is deleted from the Trashcan.
- **Restore** - Specifies whether a warning message box is shown before a model or file is restored from Trashcan.
- **Move** - Specifies whether a warning message box is shown before a selected model or file is moved to the Trashcan or to another directory.
- **Copy** - Specifies whether a warning message box is shown before a selected model or file is copied to another directory.
- **Add server** - Specifies whether a warning message box is shown when an URL is specified for addition to a model server list that does not exist.

**History**

- **History shown** - Check box option to activate the History mechanism for the Tree View. An option combo box stores information about your modeling sessions. A click on the down arrow expands the History list. A click on an item in the list expands the Tree View and highlights the item.
- **Clear the history on exit** - Specifies whether the history is cleared when MLDesigner is closed.
- **Maximum number of entries** - The number of items shown in the History combo box.
- **Maximum age of entries in days** - Specifies when History items should be removed from the list. **forever** means the items are never deleted unless the maximum number of items has been reached.
### 2.2 Graphical User Interface

<table>
<thead>
<tr>
<th>Dialog Option</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Editor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- <strong>Autosave model interval [s]</strong> - Specifies the time interval at which all models are saved automatically. Set the value to 0 to disable the autosave function.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Base cursor</strong> - You can select a new graphic for the cursor.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Zoom to Fit on open</strong> - The complete model is visible in the model window on opening. With large systems it is sometimes difficult to see the details of the system and some adjustments in the zoom level may be necessary. The default zoom when the check box is deselected is set to 100%.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Show grid</strong> - Show or hide the grid pattern in the Model Editor Window.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Highlight related objects on selection</strong> - When selecting a model element in the Model Editor Window or an item in the tree view, the corresponding element or related item is highlighted.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Set logical name as label for new instances</strong> - the model instances added to a model will be labeled with the logical name of the model they reference to; if this option is disabled, the label is composed of the physical name, followed by ‘#’ and an instance counter.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Show consistency warnings during save</strong> - Specifies whether a message box arises to report model structure errors while saving the model.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Show warning when memory is added</strong> - Specifies whether a message box arises when you add a memory to a model of a domain in which memories are not compatible with the model of computation and might lead to problems.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Show properties sorted by name</strong> - Specifies whether properties for parameters, memories, events, and resources are shown sorted by their name.</td>
</tr>
</tbody>
</table>
### Dialog Option

<table>
<thead>
<tr>
<th>Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Background</strong> - The model background color.</td>
</tr>
<tr>
<td><strong>Replace</strong> - The color of the model instance that is to be replaced. You can drag a replacement model instance into the Model Editor Window and drop it onto an existing model instance thereby replacing it.</td>
</tr>
<tr>
<td><strong>Inconsistent</strong> - Model instances that are not compatible with the domain of the module or system or have construction errors.</td>
</tr>
<tr>
<td><strong>Animation</strong> - The color of the model instance or port that is firing during simulation when graphical animation is active.</td>
</tr>
<tr>
<td><strong>State</strong> - Default color for FSM states.</td>
</tr>
<tr>
<td><strong>History</strong> - Default color for FSM histories.</td>
</tr>
<tr>
<td><strong>Default Entrance</strong> - Default color for FSM default entrances.</td>
</tr>
<tr>
<td><strong>Transition</strong> - Default color for FSM transitions.</td>
</tr>
<tr>
<td><strong>Port</strong> - The last 9 entries are used for changing the default color of the different port types.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong> - The default domain for new systems, modules and primitives in the Create New Model dialog.</td>
</tr>
<tr>
<td><strong>Copyright</strong> - The default copyright notice for user defined model elements. Appears in the tooltip text for primitives, in the online documentation and in primitive source code for primitives, modules and systems.</td>
</tr>
</tbody>
</table>

### Primitive Source Editor

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use external editors with xterm</strong> - If you have chosen an external editor that uses a xterminal console, you can choose to not see the xterminal as this often gets in the way and is not needed for editing source code files.</td>
</tr>
<tr>
<td><strong>Internal</strong> - The standard editor with built in error highlighting in cases of compile error.</td>
</tr>
<tr>
<td><strong>$EDITOR</strong> - The editor defined if you have set your environment variables for another editor.</td>
</tr>
<tr>
<td><strong>User-defined</strong> - You can replace the entry <code>vi</code> with the name of your favorite editor. The line should look like this when finished <code>emacs @$</code> if you prefer emacs.</td>
</tr>
<tr>
<td>Dialog Option</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td><strong>Colors</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Fonts</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>FSM Action Editor</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Action Editor Fonts</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Warnings</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### Run Control

#### General

- **Preferred external simulation method** - Specifies which external simulation method is preferred:
  - **C++ code** - The file type generated when running a simulation extern. The files are written to the target directory.
  - **PTcl code** - The file type generated when running a simulation extern. The destination directory must be selected from a file creation dialog.

- **Simulation start output** - The message written to standard output when an iteration of a simulation is started.
- **Simulation stop output** - The message written to standard output when an iteration of a simulation is finished / aborted.

### Warnings

The following warnings are generated after changes are made to a model while in simulation mode:

- **Show recompile warning** - Specifies whether a warning message box should be shown when the model has to be recompiled before next simulation run due to some changes.
- **Show compile warning** - Specifies whether a warning message box should be shown when the model has to be compiled before continue with an action.

The following warnings are generated when switching to simulation mode in a system that contains probes or breakpoints:

- **Show reinit breakpoint warning** - If a breakpoint parameter is changed during a simulation, a warning dialog displays when the simulation is continued.
- **Show reinit probe warning** - If a probe parameter is changed during a simulation, a warning dialog displays when the simulation is continued.

### Distributed

- **Start timeout for iterations [sec]** - The scheduler which distributes the iterations of an external simulation will wait this time in seconds until the simulation is ready to start.
2.2 Graphical User Interface

<table>
<thead>
<tr>
<th>Dialog Option</th>
<th>Debugging</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Breakpoints active</strong> - If breakpoint exist in a system it is possible to make them active/inactive when switching to simulation mode. The breakpoint can also be activated/de-activated using the <strong>Breakpoints on/off</strong> icon on the toolbar.</td>
</tr>
<tr>
<td></td>
<td><strong>Open models automatically</strong> - Hierarchical models are opened automatically on step actions. This option is only used for graphical simulations.</td>
</tr>
<tr>
<td></td>
<td><strong>Graphical animation</strong> - Activating this check box means that graphical animation will be active for every simulation. You must then deselect the <strong>Graphical Animation</strong> icon on the toolbar if you do not want graphical animation during a simulation. Simulations are a lot slower with these options and should only be used when needed. With graphical animation active the ports, modules and primitives are highlighted when they fire or accept particles during simulation.</td>
</tr>
<tr>
<td></td>
<td><strong>Textual animation</strong> - Activating this check box means that textual animation will be active for every simulation. You must then deselect the <strong>Textual Animation</strong> icon on the toolbar if you do not want textual animation during a simulation. Simulations are a lot slower with these options and should only be used when needed. With textual animation active, a log of all firings of ports, modules and primitives are printed to the Command console.</td>
</tr>
<tr>
<td></td>
<td><strong>Number of Textual Outputs</strong> - Limits the number of lines of textual output to the Animation tab of Console Window during a simulation. Setting this too high could lead to memory overload and system crash especially with extremely large systems.</td>
</tr>
<tr>
<td></td>
<td><strong>Used Source Code Debugger</strong> - Provides a list to select the debugger used for starting a simulation process within a debugger for source code debugging. You have to leave simulation mode before the changes to this option become effective.</td>
</tr>
</tbody>
</table>

### Console View

<table>
<thead>
<tr>
<th>Console View</th>
<th>Command Window</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Font</strong></td>
<td>Sets type, font, and size of font used in the Command Console.</td>
</tr>
<tr>
<td>Dialog Option</td>
<td>Log Window</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td><strong>Number of log entries (1-1000)</strong> - Limits the number of log entries. Oldest log entries are deleted once the limit is reached. Remember you can save or clear the Log Window via the context menu!</td>
</tr>
<tr>
<td></td>
<td><strong>Show information</strong> - Reduce the number of warnings or errors displayed in the Log Console by degree of severity. Many includes all trivial warnings that usually do not affect the simulation of the module or system.</td>
</tr>
<tr>
<td></td>
<td><strong>Show warnings</strong> - Display warnings that often do not affect the functionality of the model or model instance.</td>
</tr>
<tr>
<td></td>
<td><strong>Show errors</strong> - Display errors that are likely to affect the functionality of a model or model instance.</td>
</tr>
<tr>
<td></td>
<td><strong>Show log window on error</strong> - Activates the Log Window when an error occurs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conversion</th>
<th>COSSAP Model Converter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Error level</strong> - There are a number of warnings and error messages when converting COSSAP user defined project libraries to MLDesigner compatible format. This area is untested but if you are seeing too many error messages that you do not wish to see because they are perhaps unimportant, you can gradually filter them out by selecting the appropriate option from the drop-down menu.</td>
</tr>
<tr>
<td></td>
<td>- <strong>All</strong> - Shows all error messages.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Important</strong> - Filters out warnings that are not important.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Very Important</strong> - Only important messages are displayed.</td>
</tr>
<tr>
<td></td>
<td>- <strong>None</strong> - All error messages are filtered out.</td>
</tr>
</tbody>
</table>

| PTcl Converter | **Default save path** - The default path that appears in the Save As dialog when you convert an MLDesigner system to a PTcl file. |

Table 2.1: Default Settings Options
2.2 Graphical User Interface

2.2.3 Graphical User Interface Filters

Certain user interface windows, with exception of the workspace, can be hidden using these buttons situated on the toolbar. This is useful when working with large models. Hold the mouse pointer over each icon to see which window they hide or show.

2.2.4 Workspace

MLDesigner’s workspace is a multiple document interface, i.e., all opened models appear within Model Editor Windows inside the workspace area.

Each Model Editor Window consists of a headline, a model editing area, and scrollbars for vertical and horizontal navigation within the model. Clicking and holding the left mouse on the hand icon in the bottom right corner of the Model Editor Window enables you to quickly navigate to elements within large models. Open any system in MLD Libraries/Demos library and select the Model View tab in the Tree View Window to see the hierarchical structure of the system.

The headline of each window shows the type, the physical location, and the logical name of the open model. The active Model Editor Window information is also shown in the MLDesigner title bar.

To minimize, maximize, or go to full screen mode, use the arrow icons situated in the top right corner of the Model Editor Window.

The workspace can function in three different modes:

- **Cascade mode** - In cascade mode, all the Model Editor Windows have an arbitrary size and are placed in arbitrary positions inside the workspace. The size and position of each window can be set by clicking on the border of the window and dragging the window to the desired size.

- **Tile mode** - In tile mode, the workspace is divided in equally sized segments and all models are visible. This mode only applies to open workspace windows. Additional windows will be opened in cascade mode.

- **Full mode** - In full mode, all Model Editor Windows are sized to fit the workspace window size. Only the active Model Editor Window is shown in this mode.

If there is more than one window open, you can switch between the previous and next Model Editor Window by using the change window buttons.

**NOTE:** Menu items and icons on the toolbar are connected to operations of the active Model Editor Window only.

2.2.5 Tree View

There are four different views for navigating the Tree View Window:

- **File View** - Access models by their physical location.
- **Library View** - Access models by their logical hierarchy.
- **Model View** - This view is used for easy navigation of large systems or modules which are open in the Model Editor Window. Here you see all hierarchical modules and primitives contained within a module or system.
• **Search View** - This tab is used to find primitives, modules, systems or documentation if you are not sure about where they are stored. Searches using the Documentation or Description categories, can be time consuming in large libraries as all files must be opened before the end results are produced. The search criteria, All refers to all domains. Used in conjunction with the asterisk as wildcard a list of all primitives, systems and modules is produced in alphabetical order. To search for models click the Search tab in the Tree View window and activate the context menu with the right mouse-click. Choose **Model Search** and try a few searches using combinations with wild cards. You can search specific domains and limit the search using the options in the **Search in** drop-down menu (see fig. 2.2).

![Figure 2.2: Search Dialog](image)

**Figure 2.2: Search Dialog**

![Figure 2.3: Tree View Tabs](image)

**Figure 2.3: Tree View Tabs**

### 2.2.5.1 File View

The File View of existing models is constructed using the physical file structure. For the creation of the physical view, MLDesigner scans all files of a given directory and determines the type of each file. It does not analyze information about logical relations between models. If a directory contains recognized models, MLDesigner creates an item with the corresponding icon within the physical view. An item in the File View consists of the model directory name, the model type and an icon that represents the model type. MLDesigner recognizes all types of MLDesigner models including text based files that are related to models, like primitive source files, FSM models, and HTML documentation sources (see fig. 2.4). Unrecognized files are not visible in the physical view of the model hierarchy.
To view models within a directory, click the [+] sign next to the directory or library, or double click the directory. This expands the directory and MLDesigner repeats the procedure of scanning all files in the directory to determine the model type and create an icon for recognized models. To open a library item in your work area, double-click the entry.

The File View is also called **Physical View**.

The File View contains, by default, a number of top-level items, see Figure 2.3 (a). These are:

- **Favorites** - This item is convenient for creating a collection of links to systems, modules or primitives that are visited regularly. The easiest way to add models to the Favorites folder is via the Add to Favorites context menu option with the cursor placed over an item in the Tree View. Another method of adding links to the Favorites folder is to drag the item. In the Favorites directory it is possible to create new categories making it easy to organize your frequently used links in a logical hierarchy.

- **MLD Addons** - This model base points to the directory $MLD/MLD_Addons containing domain independent models which can be ordered by customers as special plug-in’s.

- **MLD Examples** - This model base points to the directory $MLD/Examples containing domain independent examples.

- **MLD Experimentals** - This model base points to the directory $MLD/MLD_Experimentals containing models shipped with MLDesigner (no support available).

- **MLD Libraries** - This model base points to the directory $MLD/MLD_Libraries containing models shipped with MLDesigner (supported by MLDesigner).

- **My Libraries** - By default this model base points to users MLDesigner model directory ~/MLD containing all models developed by you. To change the default directory the environment variable $MLD_USER must be set. This can be done via the Settings menu described in sec. 2.2.2. To change the variable permanently enter the following:

```bash
export MLD_USER=~/Directory/..
```

for bash or shell command lines and

```bash
setenv MLD_USER=~/Directory/..
```

---

**Figure 2.4**: Recognized physical file types

**Figure 2.5**: Recognized physical file types
for tcsh and csh command lines.

- **Shared Libraries** - This model base points to a directory where project work can be saved. This should be a directory where read/write rights have been set for a workgroup. To change the default directory the environment variable `$MLD_SHARED` must be set. This can be done via the Settings menu described in sec. 2.2.2. To change the variable permanently enter the following:

  ```
  export MLD_SHARED=~/Directory/..
  ```

  for bash or shell command lines and

  ```
  setenv MLD_SHARED=~/Directory/..
  ```

  for tcsh and csh command lines.

- **Home** - This item points to the user’s home directory.

- **Root** - This item points to systems root directory `/` and allows traversing through the entire directory tree.

- **Trashcan** - This item points to the directory `~/`. MLDesigner uses the Trashcan directory for removing and restoring models and directories.

### 2.2.5.2 Library View

In contrast to the File View, based on physical structure, the Library View is based on logical relations between libraries and other models. A logical relation between a library and other models is established by referencing the other models in the library model. MLDesigner analyzes all the referenced models in a given library. For all referenced models MLDesigner determines the model type and creates an item showing the logical name (if there is one) with a corresponding symbol in the logical view. Recognized model types are shown in fig. 2.6. Since it is not possible to reference text based models within an MLDesigner model, libraries can only contain models of these types.

![Logical Reference Icons in the Library View](image)

Figure 2.6: Logical Reference Icons in the Library View

As mentioned earlier, libraries in the Library View are collections of references to systems, modules, primitives and FSMs. By using the `[+]` to the left of the item, the view of the library is expanded. On expanding a reference, MLDesigner repeats the process of analyzing the type of referenced models, and creates relevant items in the logical view. By double-clicking on a library item, the model is opened in a Model Editor Window. The Library View is also called the Logical View.

The logical view contains a number of top-level items, see fig. 2.3 (b).

- **Favorites** - This item is convenient for creating a collection of links to systems, modules or
primitives that are visited regularly. The easiest way to add models to the Favorites folder is via the Add to Favorites context menu option with the cursor placed over an item in the Tree View. Another method of adding links to the Favorites folder is to drag the item. In the Favorites directory it is possible to create new categories making it easy to organize your frequently used links in a logical hierarchy.

- **MLD Addons** - This model base points to the directory $MLD/MLD_Addons containing domain independent models which can be ordered by customers as special plug-in’s.

- **MLD Examples** - This library is a reference to the model base MLD Examples and is a collection of libraries containing complex systems and modules. Expand the libraries by clicking the + to the left of the item. Note that some of the demos need SatLab installed and running in Access mode before they can be executed. The library Tutorials contains simple examples which are referred to in sections of the First Steps documentation. To access this document click the Help button in the top right corner of the GUI and choose the menu point First Steps.

- **MLD Experimentals** - This library is a reference to the model base MLD Experimentals and contains libraries of models and demos for unsupported domains shipped with MLDesigner. These libraries are sorted by domain. At the moment we cannot guarantee that these demos will work as described. The sub-library CTDE Domain is the only library in this collection that is supported at present but there may still be some bugs here.

- **MLD Libraries** - This item is a reference to the model base MLD Libraries and is the top-level library of the model hierarchy for supported domains shipped with MLDesigner.

- All other top-level items are references to libraries stored in the model bases My Libraries and Shared Libraries. Note that each library created within MLDesigner directories $MLD_USER and $MLD_SHARED are shown as sub-directories of My Libraries or Shared Libraries in the File View and as top-level libraries in the Library View.

### 2.2.5.3 Differences Between File View and Library View

It is important to understand the differences between the physical (File View) and logical (Library View) in the model hierarchy.

- The Library View is dependant on logical relations between models. A model can be referenced by a number of different libraries and can appear in numerous models. The Library View contains references to MLDesigner models. It is not possible to show primitive source files or other text based models within the logical Tree View.

- Deleting a model in the File View results in all redundant references to it becoming wrong. This is indicated in the Library View by a broken model icon.

### 2.2.5.4 Find in Tree View Context Menu Option

It is possible to expand the Tree View to find a selected item by activating the context menu over a model instance in the Model Editor Window and choosing Find in Tree.
2.2.5.5 Tree View Filter

Above the Tree View lies a toolbar that contains a number of buttons with icons for each model (see fig. 2.7). These buttons are used to select which types of models are shown in the File (physical) and the Library (logical) views. In the File View it is possible to filter out all types of models including text based files such as primitive source and FSM model files. The Library View contains items that refer to models of type library, system, module and primitive. Only items of this type (excluding libraries) can be filtered out in the Library View. For an explanation of each icon place the mouse pointer over the buttons of the Tree View filter toolbar.

![Figure 2.7: Tree View filter toolbar](image)

2.2.5.6 Context Menu

The context menu in Tree View can be used to deal with models and other model related files. To open the context menu click the right mouse button over the item in the Tree View. The available options differ between File View and Library View and vary according to the type of item selected within each view. Some options depend on whether the item is write protected or not. The typical context menus for the Library view are shown in fig. 2.8 where:

- a) is the System context menu,
- b) is the Library context menu, and
- c) is the Primitive context menu.

![Figure 2.8: Tree View Context Menu Examples](image)

The various context menu items are explained in detail here:

**Online Documentation** Used to open the document browser showing hypertext documen-
2.2 Graphical User Interface

tation for the model. This context menu item is available for all MLDdesigner models.

**Open Model**
Used to open the model in Model Editor. This context menu item is available for all MLDdesigner models.

**New Model**
Used to create a new model. This item is available over library and model base (excluding My Libraries and Shared Libraries) items in the File and Library View. The following options are available:

1. **Library**
2. **System**
3. **Module**
4. **Primitive**
5. **Fsm**
6. **Probe**

**Add to Favorites**
Add a model to the Favorites folder. Only available for systems, modules, and primitives.

**Save As**
Using this context menu item you can create a full copy of the model and save the model in a writable library. Available for Tree View items that refer to MLDdesigner models.

**Duplicate**
Using this context menu item you can create a full copy of the model in the same library. For differentiation the number of the copy is added to the model name, e.g., the first copy of a model with name Example is called Example_01. Available for Tree View items that refer to MLDdesigner models.

**Rename**
Using this context menu item you can change the physical and logical name of a model. Available for Tree View items that refer to MLDdesigner models.

**Remove Reference**
Used to delete the reference to a model but not the model itself. Available for systems, modules, and primitives for Library View only.

**Move to Trash**
Used to move the model to Trashcan and deletes the reference in the same library. All other references are wrong and appear as a broken icon in the Library View which can be removed using the Remove Reference option.

**Open Source**
Used to open and edit the primitive source code with the primitive editor. The primitive editor is a text editor that provides you with editing capabilities like syntax highlighting and primitive specific functions. Primitive source editors are not opened within the workspace, but as new top-level windows. This option is only available for primitives.

**Compile Source**
This option is used to start the compilation of the primitive source file. The following options are available:
1. Compile Optimized  
2. Compile With Debug

Only available over primitives. To compile all primitives contained in a library use Compile Library over a library.

Open FSM  
Opens the FSM model within an FSM Model Editor Window. The FSM Model Editor Window provides you with graphical editing capabilities for the development of FSM models. This context item is only available for Tree View items that refer to FSM models and FSM primitive components.

Open Special  
Using this context menu item, you can open a pre-configured special primitive model for an existing generic primitive. Only available for primitives.

Create Special  
Using this context menu item, you can create a pre-configured special primitive model for an existing generic primitive. For further information on how to create special models refer to sec. 2.3.3. This context menu item is only available for primitives where write permissions are set.

Delete Special  
Using this context menu item, you can delete a pre-configured special primitive model for an existing generic primitive. Only available for primitives where write permissions are set.

Browse File  
This item opens a hypertext browser showing online hypertext documentation of the selected model. Available for Tree View items that refer to hypertext files only.

Empty Trash  
This option deletes all files stored in the Trashcan. To delete single models, select the model in the Trashcan view and select Delete from the context menu. Only available within the Trashcan directory in File View.

Restore  
This context menu item restores the selected files stored in the Trashcan to their original directory. Only available in Trashcan directory in File View.

Export Library  
This context menu item exports a library to a mar archive. Only available over top-level libraries.

Import Library  
This context menu item imports a mar archive containing a top-level library to directory $MLD_USER or $MLD_SHARED. Only available over model bases My Library or Shared Libraries in File View.

Update Version  
If the Library needs a version update and the library is writable this option is visible and active over libraries in $MLD_USER or $MLD_SHARED. All imported models are also updated.

Refresh  
MLDesigner starts to update the Tree View for a selected Tree View item and the whole model hierarchy below this item. Available for
all Tree View items including the background if there are no Tree View items selected.

**Expand**

All items of a directory are expanded. Items filtered out using the Tree View filter will not be visible. To expand also all sub-items use **Expand Subtree**. The items have a $+\$ to indicate they are expandable items. Available for all Tree View items with a hierarchical structure.

**Collapse**

All items of a directory are collapsed. To collapse also all sub-items use **Collapse Subtree**. The items have a $-$ to indicate they are collapsible items. Available for all Tree View items with a hierarchical structure. Available for all expanded Tree View items with a hierarchical structure.

The Tree View allows the selection of multiple items and has tool tip functionality. To select multiple items in the Tree View press **Ctrl** and click the items you want to select using the left mouse button. To select all items in the Tree View click the first and last item while holding the **SHIFT** key pressed. To deselect items, hold the **Ctrl** key pressed while clicking the appropriate items. If the mouse pointer is held above a Tree View item, you will see:

- the physical filename of the model or the file
- the owner, the group and the access rights to the file
- the logical name of the model (if there is one)

### 2.2.5.7 Copying, Moving, and Referencing Models

In addition to the usage of context menu items **Save As**, **Duplicate**, **Move to Trash**, and **Add to Favorites** for copying/moving models or creating model references, you can use items from the pop-up menu appearing after drag & drop a model over any directory in Tree View. The various items are:

- **Copy** - Creates a full copy of the model and saves the model in a writable directory. This item is only available in File View.

  **NOTE:** In contrast to context menu item **Save As** all sub-directories of the model directory are copied too. In this way you can copy the library with all included models to another folder.

- **Move** - Creates a full copy of the model, saves the model in a writable directory, and deletes the model in the origin folder. This item is only available in File View.

- **Reference** - Creates a reference to a model that is visible only in Library View.

### 2.2.6 Property Editor

The Property Editor is used to modify the common properties of models as well as the properties of model instances and other model elements such as ports, memories, events and resources. The Property Editor is also used to define parameters and target parameters of models. Additionally,
model elements such as delays, buses and labels are configured using the Property Editor. Figure 2.9 shows an example of the Property Editor for a module and the instance of the module Ramp.

![Property Editor example](image)

(a) Model properties  
(b) Instance properties

Figure 2.9: Property Editor example

To edit the properties select the field you wish to edit by clicking the relevant item in the Value column. The text in non-editable fields is grey and italic. The data input method differs depending on the type of value. Certain types of values have a button which opens a dialog where the value can be defined. Other types of properties such as parameters, ports, memories, and events are expandable cells where a property must be selected from a drop-down menu. For parameters of model instances, only the value of the parameter can be set. The Property Editor is also used to link model instance properties like parameters, memories, events and resources to other model elements.

The simulation properties window is only visible when MLDesigner is in simulation mode. To see the simulation properties window click the Switch to Simulation Mode icon on the toolbar. All system parameters defined with external scope in the Model Property Editor, are visible in the Simulation Properties window. These parameters can be changed without modifying the system.

### 2.2.7 Console Window

The console window is used to interact with MLDesigner using the built-in interpreter and to inform you about the status of MLDesigner and has two panels when in Edit Mode:

- **Command** - The command window is the user interface with the built-in PTCL interpreter. The results of these commands are printed to this window. For detailed information about the built-in PTCL interpreter, refer to the MLDesigner manual ch. 9. The prompt shows you the current working directory.

- **Log** - The log window is the console where a record of your actions is displayed and error messages or warnings are printed. You can save the contents of the console to a file or clear the console by selecting the appropriate menu option from the context menu. The error messages are displayed as an expandable entry with the top level entry displaying the
name of the system containing the error. To see more details about the error or warning you must expand the entry by clicking the [+] next to the top level entry. To see the primitive or module responsible for the error you must click on the appropriate entry. The model instance is then highlighted in the Model Editor Window and the model instance editor displays all parameters of the model instance.

Further tabs appear when you are in Simulation Mode:

- **Breakpoints** - The breakpoints window is the Console where you see a list of all breakpoints. Breakpoints are useful for debugging systems and the list of breakpoints makes it easier for you to find and edit them. You can add and remove breakpoints while in simulation mode. To do so select Add Breakpoint or Remove Breakpoint from the context menu over the relevant entry in the breakpoints console.

- **Progress** - The progress window is the console where you can monitor the compile process and the progress of the simulation measured as a percentage value.

- **Animation** - Another tab, the animation tab, appears if the Textual Animation icon on the toolbar is checked. The port and instance firings are printed to the console in this mode and simulation is a lot slower.

### 2.2.8 Using Menu Bar

In the following sections the submenus of the menu bar are explained in brief. For more detailed information about operations triggered by the menu items, refer to the relevant section indexed next to the description.

#### 2.2.8.1 File Menu

Figure 2.11 shows the MLDesigner file menu. To open the File menu, a shortcut key Alt-F can be used. The items in the file menu have the following functions:
New... Ctrl-N Invokes MLDesigner to start the creation of a new model of type library, system, module, primitive, or FSM primitive.

Open... Ctrl-O Opens a model within the active Model Editor Window or in a new Model Editor Window. MLDesigner opens a new Model Editor Window if no Model Editor Window is open, or the active Model Editor Window is not compatible with the model or the file to be opened. MLDesigner opens a dialog for selecting the model to open.

Reload Ctrl-R Reloads the model in the active Model Editor Window. This menu item is only active if a Model Editor Window is opened and contains a model.

Close Ctrl-W Closes the active Model Editor Window. If the model in the active Model Editor Window has been modified, you will be prompted to save your changes.

Save Ctrl-S Saves modifications of a model in the active Model Editor Window. This menu item is only active if the model has been modified.

Figure 2.11: File menu
2.2 Graphical User Interface

**Save As**
Creates and opens a copy of the model in the active Model Editor Window. The whole model, including the model, the icon, and all related files are copied.

**Save All**
Save the modifications of all models opened in the work area. This menu item is only active if there are models with modifications.

**Print**
Prints the model in the active Model Editor Window directly to a printer or to a file as EPS.

**Export EPS**
Start the export of the model in the active window as EPS.

**Convert BONeS**
Starts the conversion of model libraries from BONeS Designer to an MLDesigner compatible format.

**Convert COSSAP**
Starts the conversion of COSSAP designs and models to MLDesigner compatible format.

**Quit**
Commands MLDesigner to exit. If there are unsaved modifications in any model, you will be prompted to save them.

The file menu shows, in addition to the file menu items listed above, a history menu containing models or models that were last opened. They can be opened directly using the corresponding entry.

### 2.2.8.2 Edit Menu
The edit menu can be opened by using the shortcut key Alt-E or by right mouse click in the active Model Editor Window. It is a dynamic menu. The configuration of the edit menu depends on the type of model opened within the active Model Editor Window as well as the type of object or objects selected.

The Edit menu on the menu bar is exactly the same as the context menu in the Model Editor Window. If you select a model instance or any other model element, the context menu shows items for all operations available for the selected model element.

In contrast, by clicking on the model background, the context menu shows items for all operations that are available for the model itself. Figure 2.12 shows three examples of edit menus. Figure 2.12 a) shows the simplest edit menu, which is available regardless of what has been selected. The edit menu shown in fig. 2.12 b) appears for a selected model instance. A number of edit menu items are only active if you have write access to the model. The last example in fig. 2.12 c) shows the edit menu for a model if it belongs to a model of type system.

Edit menu example in fig. 2.13 d) is activated when you select a relation between two model instances in a system and then activate the Edit menu or the context menu. You can change the color of relations (the lines connecting input and output ports) between model instances. You are also
Figure 2.12: Edit menu examples

Figure 2.13: Edit menu examples
able to add a Bus to a relation if the ports are of type multi-port. The option **Orthogonal** changes the way the relation lines react when you click on nodes to arrange the layout of the connections.

Edit menu fig. 2.13 e) is activated in cases when you select a port in the Model Editor Window and the open model is of type **System**. Here you have the additional options of adding Initializable or Non-initializable delays as well as disconnecting or terminating ports. Selecting **Terminate** while a port is selected, will disconnect and terminate the port simultaneously.

The following edit menu items are available in all model Model Editor Windows.

<table>
<thead>
<tr>
<th><strong>Undo</strong></th>
<th>Ctrl-Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useful for reversing the last changes made to your model or model and reverting back to the last saved state.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Redo</strong></th>
<th>Ctrl-Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useful for reverting back to last changed state of a model or model.</td>
<td></td>
</tr>
</tbody>
</table>

**Add Breakpoint**

This option is only available when MLDesigner is in Simulation Mode and a model instance, primitive, or port is selected in the Model Editor Window. Breakpoints are useful for debugging systems.

**Edit Breakpoint**

The breakpoint settings can be changed while in simulation mode. It is possible to deactivate breakpoints or set the ignore count property of the breakpoint.

**Remove Breakpoint**

Deletes the highlighted breakpoints in the breakpoint editor window.

**Delete**

Del

Deletes all selected objects in the active Model Editor Window. This item is only active if you have write access to the model and at least one model element is selected.

**Cut**

Ctrl-X

Moves all selected objects in the active Model Editor Window to the clipboard. The contents of clipboard can be used to paste the objects into any other model. This item is only active if you have write access to the model and at least one model element is selected.

**Copy**

Ctrl-C

Copy all selected objects in the active Model Editor Window to the clipboard. The contents of clipboard can be used to paste the object into another compatible model. This item is active if at least one model element is selected.
### Paste

**Ctrl-V**  
Copies the contents of the clipboard to the model in the active Model Editor Window provided that the content of the clipboard is compatible with the model. This item is only active if you have write access to the model.

### Select All

**Ctrl-A**  
Marks all model elements in the active Model Editor Window including text labels and connections. You can then delete, copy or cut all items.

The following edit menu items are used to deal with models. The availability of menu options depends on the type of model or model selected.

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Online Documentation</strong></td>
<td>Alt-D</td>
<td>Opens hypertext documentation for the model in the active Model Editor Window or the model referenced by the selected model instance. If the documentation does not exist, it is automatically generated from the model.</td>
</tr>
<tr>
<td><strong>Open Model</strong></td>
<td>M</td>
<td>Opens the model of the model. This menu item is available if a model instance is selected.</td>
</tr>
<tr>
<td><strong>Open FSM</strong></td>
<td>F</td>
<td>Opens an FSM model. This menu item is only available if either the model opened in the active Model Editor Window belongs to an FSM primitive or the selected model instance refers to an FSM primitive.</td>
</tr>
<tr>
<td><strong>Open Source</strong></td>
<td>S</td>
<td>Opens the primitive source file. MLDesigner opens the primitive source code using a primitive editor. This editor is a text editor that provides you with editing capabilities like syntax highlighting and primitive specific functions. This menu item is only available if the model opened in the active Model Editor Window belongs to a primitive model or the selected model instance refers to a primitive model.</td>
</tr>
<tr>
<td><strong>Open Base Primitive</strong></td>
<td></td>
<td>Opens the base primitive (built-in primitive). This menu item is only available if the model instance selected in the active Model Editor Window is a primitive model component and is a special primitive.</td>
</tr>
<tr>
<td><strong>Compile Source</strong></td>
<td></td>
<td>Compiles the primitive source file. This menu item is only available if the model opened in the active Model Editor Window is a primitive or the selected model instance is a reference to a primitive model component.</td>
</tr>
</tbody>
</table>
### Replace Instance
Replaces the model instance selected in the active Model Editor Window. A **Select Model** dialog is displayed where you can choose the type of model to replace the existing one with. Connections are only maintained for ports with exactly the same name as the existing model instance.

### Select Tool
Switches the cursor back to normal select modes. This is the same as right mouse-click if the cursor is in any other mode.

### Dynamic Instance
Creates a Dynamic Instance of the selected model instance. See ch. 3.14 for more details.

### Hide Port Labels
This option is only active if port labels are visible and a model instance is selected. The port labels can be turned off if the Model Editor Window gets too busy or as soon as you have connected all model instances. The port label is set by clicking the **Formal Port** of the Primitive or Module to activate the **Port Properties** Editor in the bottom left corner of the gui. The label can only be changed if the module is saved in a write environment.

### Show Port Labels
Makes the port label visible in the system or module Model Editor Window. This can be useful when connecting modules with multiple input/output ports where the order of connection is important. Once connections are made you can turn the labels off again. The model appears much neater with labels turned off. See **Hide Port Labels**.

### Rotate Ports
This option opens a sub-menu with the choice of rotating the ports clockwise, anticlockwise or 180 degrees. This can be useful with imported BONeS models if the port alignment is not satisfactory (usually a problem if ports were rotated in BONeS Designer).

### Mirror Ports
You can choose to mirror the ports on the X axis or Y axis of the model instances. This option has not been thoroughly tested on converted models.

### Swap Ports
Swaps port positions horizontally or vertically without the connections crossing over.

### Disconnect Connected
Disconnects all connected ports of selected model instances.
### 2 Modeling with MLDesigner

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminate Unconnected</td>
<td>Terminates the unconnected ports of selected model instances in a system or module.</td>
</tr>
<tr>
<td>Unterrminate Terminated</td>
<td>Unterrminates all terminated ports of selected model instances.</td>
</tr>
<tr>
<td>Background Color C</td>
<td>Using this menu item opens a color dialog for selecting background colors for models and models.</td>
</tr>
</tbody>
</table>

The following edit menu items are used to create new model elements within the model in the active Model Editor Window. These menu items are only available if the model belongs to a model of type library, system, module, or primitive. The configuration of the edit menu can differ for models of different types. If you do not have write access to the model these items are disabled.

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Model Instance B</td>
<td>Creates a new instance of a model at the current cursor position within the model. The model to be embedded is selected using the Select Model dialog.</td>
</tr>
<tr>
<td>Add Input Port I</td>
<td>Creates a new input port at the current cursor position within the model in the active Model Editor Window. The properties of the new input port are set to default values and are changed using the Property Editor.</td>
</tr>
<tr>
<td>Add Output Port O</td>
<td>Creates a new output port at the current cursor position within the model in the active Model Editor Window. The properties of the new output port are set to default values and are changed using the Property Editor.</td>
</tr>
<tr>
<td>Add Event E</td>
<td>Creates a new event element at the current cursor position within the model in the active Model Editor Window. The properties of the new output port are set to default values and are changed using the Property Editor.</td>
</tr>
<tr>
<td>Add Memory M</td>
<td>Creates a new memory element at the current cursor position within the model opened in the active Model Editor Window. The properties of the new output port are set to default values and are changed using the Property Editor.</td>
</tr>
<tr>
<td>Add Resource R</td>
<td>Creates a new resource element at the current cursor position within the model in the active Model Editor Window. The properties of the new output port are set to default values and are changed using the Property Editor.</td>
</tr>
</tbody>
</table>
2.2 Graphical User Interface

**Add Text Label**  
L  Creates a new text label at the current cursor position within the model in the active Model Editor Window. The properties of the new label are defined using the Property Editor.

The following edit menu items are only available for FSM models. If you do not have write access to the FSM model these edit menu items are disabled.

**Add State**  
S  Creates a new state in the FSM model in the active Model Editor Window. The properties of the new state can be changed using the Property Editor.

**Create Transition**  
A  Starts the creation of state transition arcs. MLDesigner switches to the arc creation mode indicated by a special cursor. In this mode, every click on a state will create or complete an arc. To switch back to selection mode click the right mouse button or the **Select tool button**. After creation you can change the arc properties using the Property Editor.

**Add History**  
A  Starts the creation of History model elements. MLDesigner switches to the History creation mode indicated by a special cursor. In this mode, every click on a state will create a History model element. To switch back to selection mode click the right mouse button or the **Select tool button**. After creation you can change the History property to **Recursive** using the Property Editor.

**Add Default Entrance**  
A  Starts the creation of Default Entrances. MLDesigner switches to the Default Entrance creation mode indicated by a special cursor. In this mode, every click on a state will create a Default entrance. To switch back to selection mode click the right mouse button or the **Select tool button**. After creation you can set the Default Entrance action using the Property Editor.

### 2.2.8.3 View Menu

The zoom can be set between 12 % and 400 % of the original size. The default zoom value is such that the whole model is visible in the active Model Editor Window. Figure 2.14 shows the MLDesigner view menu. The view menu is only active if at least one Model Editor Window is opened within the MLDesigner workspace.

**Zoom In**  
+  Selects the next higher zoom level for the active Model Editor Window.
2 Modeling with MLDesigner

Figure 2.14: View menu

- **Zoom Out** Selects the next lower zoom level for the active Model Editor Window.

- **Zoom To Fit** Selects a zoom level for the active Model Editor Window so the whole model is visible in the active Model Editor Window.

- **Refresh** Redraws the model in the active Model Editor Window.

### 2.2.8.4 Window Menu

The items of the window menu are used to control which window within the MLDesigner workspace is the active Model Editor Window. Figure 2.15 shows the window menu. It is only functional if there are at least two Model Editor Windows opened within the MLDesigner workspace.

<table>
<thead>
<tr>
<th>Command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close</td>
<td>Ctrl+W</td>
<td>Closes the active Model Editor Window. If the model opened in the active Model Editor Window has been modified, you will be prompted to save your changes.</td>
</tr>
<tr>
<td>Close All</td>
<td>Ctrl+Shift+W</td>
<td>Closes all Model Editor Windows. If there are modified models open in the Model Editor Window you will be prompted to save your changes.</td>
</tr>
<tr>
<td>Next</td>
<td>Ctrl+Right</td>
<td>Switches to the next Model Editor Window, i.e., the next Model Editor Window within MLDesigner workspace becomes the active Model Editor Window.</td>
</tr>
</tbody>
</table>

Figure 2.15: Window menu
2.2 Graphical User Interface

**Previous**   
Ctrl-Left  
Switches to the previously active Model Editor Window, i.e., the previous Model Editor Window within MLDesigner workspace becomes the active Model Editor Window.
Switches to cascade mode, i.e., all Model Editor Windows have an arbitrary size and are placed at arbitrary positions inside the workspace. Each Model Editor Window can overlap the other in this mode.

**Tile**   
Ctrl-T  
Switches to tile mode, i.e., the workspace is divided into equally sized parts and all Model Editor Windows are visible.

**Full**   
Ctrl-F  
Switches to the full mode, i.e., all Model Editor Windows are sized to fit the whole of the workspace area. Only the active Model Editor Window is visible in this mode.

The window menu is a dynamic menu. In addition to the static items explained above, the window menu contains a reference to the last four models or models opened. The active Model Editor Window is marked with a grey box.

### 2.2.9 Using Toolbars

For faster access to operations, the MLDesigner graphical user interface has a number of toolbars. The appearance of toolbars and their configuration depend on the type of model or model selected in the active Model Editor Window. You can hide the toolbars or shift them to any border of the MLDesigner user interface.

MLDesigner uses three toolbars

- standard toolbar,
- Tree View toolbar,
- editor toolbar.

#### 2.2.9.1 Standard Toolbar

The standard toolbar is always visible. Some options are only active if at least one Model Editor Window is open and you have write access for the model or model selected in the active Model Editor Window. Figure 2.16 shows the standard toolbar. In the following section the function of each tool button is explained briefly. For more detail see sec. 2.2.8

![Standard toolbar]

Figure 2.16: Standard toolbar

**New Model**  
Invokes MLDesigner to create a new model of type library, sys-
2 Modeling with MLDesigner

- **Open Model**: Opens a model in the active Model Editor Window or in a new Model Editor Window. MLDesigner opens a new Model Editor Window if there is no Model Editor Window open or the model is not compatible with the active Model Editor Window. MLDesigner opens a dialog for selecting the model or file to open. This tool button is always active.

- **Reload Model**: Reloads the model in the active Model Editor Window. This tool button is only active if a Model Editor Window is open.

- **Save Model**: Saves the modifications to the model in the active Model Editor Window. This tool button is only active if the model has been modified.

- **Save All Models**: Save the modifications of all models in open Model Editor Windows. This tool button is only active if the models have been modified.

- **Print Model**: Prints the model in the active Model Editor Window. Models can be sent directly to a printer or saved as a PostScript file. This tool button is active if at least one Model Editor Window is open.

- **Online Documentation**: Opens the document browser showing the hypertext documentation for the model. The document browser is described in more detail in This tool button is only active if at least one Model Editor Window is open.

- **Delete**: Deletes all selected objects of the model in the active Model Editor Window. This tool button is only active if you have write access to the model and at least one model element is selected.

- **Cut**: Moves all selected objects of the model in the active Model Editor Window to the clipboard. The contents of the clipboard can be used to paste the objects into another model. This tool button is only active if you have write access to the model and at least one model element is selected.

- **Copy**: Copies all selected objects of the model in the active Model Editor Window to the clipboard. The contents of clipboard can be used to paste the objects in another model. This tool button is only active if at least one model element is selected.

- **Paste**: Inserts the contents of the clipboard in the model in the active
Model Editor Window provided that the content of the clipboard is compatible with the model. This tool button is only active if you have write access to the model and the clipboard contains model elements that are compatible with the model.

**Undo**
This tool button is only active if at least one Model Editor Window exists within the workspace and at least one change has been made to the model.

**Redo**
This tool button is only active if at least one Model Editor Window exists within the workspace and at least one undo step has been performed.

**Zoom In**
Selects the next higher zoom level for the active Model Editor Window. This tool button is only active if at least one Model Editor Window is open in your workspace.

**Zoom Out**
Selects the next lower zoom level for the active Model Editor Window. This tool button is only active if at least one Model Editor Window is open in your workspace.

**Zoom To Fit**
Selects a zoom level for the active Model Editor Window where the entire model is visible in the Model Editor Window. This tool button is only active if at least one Model Editor Window is open in your workspace.

**Tree View**
Opens and closes the Tree View window. This tool button operates as a check button. If the button is checked, the Tree View window is visible. If the button is not checked, the Tree View window is hidden. This tool button is always active.

**Property Editor**
Opens and closes the Property Editor window. This tool button operates as a check button. If the button is checked, the Property Editor window is visible. If the button is not checked, the Property Editor window is hidden. This tool button is always active.

**Console Window**
Opens and closes the console window. This tool button operates as a check button. If the button is checked, the console window is visible. If the button is not checked, the console window is hidden. This tool button is always active.

**Data Type Editor**
Opens and closes the data type window. This tool button operates as a check button. If the button is checked, the data type window is visible. If the button is not checked, the data type window is hidden. This tool button is always active.

**Previous**
Switches to the previously active Model Editor Window, i.e.,
the previous Model Editor Window within MLDesigner workspace becomes the active Model Editor Window. This tool button is only active if at least two Model Editor Windows are opened within your workspace.

Next

Switches to the next Model Editor Window, i.e., the next Model Editor Window within MLDesigner workspace becomes the active Model Editor Window. This tool button is only active if at least two Model Editor Windows are open within your workspace.

Cascade

Switches to cascade mode, i.e., all Model Editor Windows have an arbitrary size and are placed at arbitrary positions inside the workspace. Each Model Editor Window overlaps the other in this mode. This tool button is only active if at least two Model Editor Windows are opened within the workspace.

Tile

Switches to tile mode, i.e., the workspace is divided in equally sized parts. The size of all Model Editor Windows is such that each window is visible and regularly spaced. This tool button is only active if at least two Model Editor Windows are opened within the workspace.

Full

Switches to full mode, i.e., all Model Editor Windows are sized to fit the entire workspace. Only the active Model Editor Window is visible in this mode. This tool button is only active if at least one Model Editor Windows is open within the workspace.

2.2.9.2 Tree View Toolbar

Above the Tree View lies a toolbar containing a number of buttons with icons for each model (see fig. 2.17). These buttons are used to select which types of models are shown in the File (physical) and the Library (logical) views. In the File View it is possible to filter out all types of models including text based files such as primitive source and FSM model files. The Library View contains items that refer to models of type library, system, module and primitive. Only items of this type (excluding libraries) can be filtered out in the Library View.

Figure 2.17: Tree View filter toolbar

Show Libraries

If the tool button is checked, library items are visible, in the Tree View. This tool button is only active in the File View.

Show Systems

If the tool button is checked, system model items are visible in the Tree View. This tool button is always active.
2.2 Graphical User Interface

### Show Modules
If the tool button is checked, module items are visible in the Tree View. This tool button is always active.

### Show Primitives
If the tool button is checked, primitive items are visible in the Tree View. This tool button is always active.

### Show Probes
Show Probes in the Probes Libraries.

### Show Primitive Source Files
If the tool button is checked, primitive source files are visible in the Tree View. This tool button is only active in the File View.

### Show FSM Models
If the tool button is checked, FSM model items are visible in the Tree View. This tool button is only active in the File View.

### Show HTML Files
If the tool button is checked, HTML file items are visible in the Tree View. This tool button is only active in the File View.

#### 2.2.9.3 Editor Toolbar

The editor toolbar is only visible if a Model Editor Window is open in the workspace. This tool bar is dynamic; the availability of tool buttons depends on the type of model open in the active Model Editor Window or the type of model selected in the active Model Editor Window. Figure 2.18 shows a typical editor toolbar. On clicking a tool button the cursor changes mode. To switch the cursor back to normal mode, click the Select Tool button or the right mouse button.

![Editor toolbar]

Figure 2.18: Editor toolbar

**Select Tool**
Switches the cursor back to normal selection mode. This tool button is active if at least one Model Editor Window is opened in the workspace.

**Zoom Tool**
Switches the cursor to zoom mode. You can zoom in on a section of the model by left clicking the mouse or by selecting a zoom level from the context menu while the cursor is in zoom mode. This tool button is always active if at least one Model Editor Window is opened.

**Pan Tool**
Switch to the pan mode. In pan mode, you can grab the model background to move the the model in all directions. This tool button is always active if at least one Model Editor Window is opened.

The following tool buttons are only available in the editor toolbar if the model in the active Model Editor Window is of type library, system, module or primitive. A different set of tool buttons is used for Model Editor Windows that deal with FSM models. Most tool buttons are only active if
you have write access to the model.

**Add Model Instance**

Switches the cursor to instance creation mode. In instance creation mode, you can create new model instances at the current cursor position with a mouse-click.

**Add Input Port**

Switches to input port creation mode. In input port creation mode, you can create new input ports at the current cursor position.

**Add Output Port**

Switches to output port creation mode. In output port creation mode, you can create new output ports at the current cursor position.

**Add Event**

Switches to event creation mode. In event creation mode, you can create new event objects at the current cursor position.

**Add Memory**

Switches to memory creation mode. In memory creation mode, you can create new memory objects at the current cursor position.

**Add Resource**

Switches to resource creation mode. In resource creation mode, you can create new resource objects at the current cursor position.

**Add Non-Initializable Delay**

Switches to delay property creation mode. In delay property creation mode, you can create delay properties at the current cursor position.

**Add Initializable Delay**

Switches to delay property creation mode. In delay property creation mode, you can create initializable delay properties at the current cursor position.

**Add Bus**

Switches to bus property creation mode. In bus property creation mode, you can create bus properties at the current cursor position.

**Add Text Label**

Switches to text label creation mode. In text label creation mode, you can create new text labels at the current cursor position.

The following tool buttons are available for FSM models. These tool buttons are specialized for dealing with FSM models and are only active if you have write access to the FSM model.

**Add State**

Switch to FSM state creation mode. In state creation mode, you can create new FSM states. The properties of the new states can be changed using the Property Editor.

**Add Arc**

Switches to FSM arc creation mode. In FSM arc creation mode,
2.3 Handling Models

every time you click on a state, an arc is created or completed. After the creation of an FSM arc, the arc properties must be set using the Property Editor.

The availability of the following tool buttons depends on the type of model open in the active Model Editor Window.

- **Open Model**: Opens a schematic of the model selected in a model or system.
- **Open FSM**: Opens the FSM model of the model. This tool button is only available if the model in the active Model Editor Window belongs to an FSM primitive.
- **Open Source**: Opens the primitive source file. MLDesigner opens the primitive source code within a primitive editor. This editor is a text editor that provides you with editing capabilities such as syntax highlighting and primitive specific functions. This tool button is only available if the model in the active Model Editor Window belongs to a primitive model.
- **Compile Source**: Compile the primitive source file. This tool button is only available if the model in the active Model Editor Window is a primitive model.
- **Switch to Simulation Mode**: Activates the simulation run control toolbar, the **Breakpoints** and **Progress** consoles, and the **Simulation Properties** window.

2.3 Handling Models

The following section explains the handling of models regardless of type. Themes explained in this section are:

- Creating new models
- Copying existing models
- Opening, saving and closing models
- Deleting models
- Printing models
- Export/import of models

2.3.1 Creating New Models

To create a new model select **New** from the **File menu** or click the tool button **New Model** on the standard toolbar. A **Create New Model** dialog where you define the properties of the new model appears. See fig. 2.19.
2 Modeling with MLDesigner

Figure 2.19: Creation Dialog for the Example

- **Type of Model** - From the drop-down menu select the type of model you wish to create. This can be of type Primitive, Module, System, Library, FSM, or Probe.

  **NOTE:** It is not possible to change the type of model after its creation.

- **Logical Name** - In this field you must define a logical name for the model. The logical name is used to identify the model in the Library View. The logical name is also visible in the title bar of the Model Editor Window and in the tool tip text if you hold the mouse over an instance of the model.

- **Library** - Here you can select the library to which the new model will belong. This determines the location of the new model both in the physical and the logical model hierarchy. The new model becomes a sub-directory of the chosen library and is identified in the File View of the tree window by its physical name. The same item in the Library View is identified by its logical name. Click the icon on the right of the library text field. A pop-up window is displayed where you can select a location to save the model. Here you have the option of selecting a directory in the File View or a Library in the Library View. If you want to create a new library leave this field empty. The new library will be located in $MLD_USER (normally "~/MLD") as a top-level directory identifiable by its physical name in the File View and by its logical name in the Library View.

- **Modeling Domain** - If the new model is of type system, module, or primitive select the domain that defines the model of computation for the model from the drop-down menu. If you select an experimental, unsupported domain, you will see a warning message.
• **Derive** - If the new model is of type primitive, this field is used to define from which existing primitive models the new one will derive, i.e., from which primitive the new one inherits model parameters, ports, memories, events and resources.

• **Physical Name** - In this field you can define the physical name of the new model component. If no physical name is entered, it is generated from the logical name and all characters which are not letters or digits are replaced with an underscore (_). Either the physical or the logical name must be defined.

• **Physical Location** - This field is only active for top-level libraries and is used to specify the model base (My Library or Shared Libraries) where the library and all models such as primitives, modules or systems contained in the library are stored.

• **Group Permissions** - While creating a new library, you can set access rights for members of your user group. By default the group members have read and write access to the models.

• **World Permission** - While creating a new library, you can set the access rights for users that do not belong to your UNIX user group. By default, other users have only read access to your models. If you create any other type of model, i.e., system, module, or primitive permissions are determined by the library to which the new model will belong.

• **Description** - This description is used for tool tip texts generated for instances of the new model as well as for short descriptions in automatically generated online model documentation.

• **Load Mode** - Defines the load mode. This property is only available for primitives and probes. FSM primitives do not use this option. You can select the load mode as **Dynamic** or **Permanent**. If you select load mode **Dynamic**, a shared object containing the primitive code is created on loading. This shared object is linked dynamically to MLDesigner so the primitive can be reloaded after changes are made to it. In contrast, by selecting load mode **Permanent**, the primitive is linked to MLDesigner on startup as is the case with built-in primitives (MLDesigner Libraries). Changes to permanently loaded primitives or probes only take effect after shutting down and restarting MLDesigner.

**NOTE:** It only makes sense to set the load mode to **Permanent** if you are going to create new primitives derived from the permanently loaded primitive. In such cases the permanently loaded primitive is known as the **base primitive**.

MLDesigner checks whether there is already a directory or a model with the same physical name in the directory of the selected library. If there is either a directory or a model with the same physical name, a warning message displays. In that case, you must change the physical name of the new model or select another library to save the model in.

It is possible to give the new model the same logical name as an existing model in the same location.

**NOTE:** It is NOT possible to create a primitive with the same physical name in the same domain regardless of the location. Primitives must be given unique names.

The new directory contains all files necessary to describe the model.
Dependent on its type these are:

* An XML (eXtensible Markup Language) model file with extension `.mml` and filename that corresponds to the Physical Name, e.g. `SDFMy_Primitive.mml`.
* A file containing the hypertext documentation of the model in HTML format, e.g. `SDFMy_Primitive.htm`.
* In case of a primitive, the primitive source file that serves as a template defining the functionality, e.g. `SDFMy_Primitive.pl`.
* In case of a primitive, a makefile needed for its compilation.

MLDesigner creates a reference to the new model within the selected library thereby integrating the new model in the logical model hierarchy (Library View). You can create further model references for the new model within other libraries to realize a logical model hierarchy that differs from the physical model hierarchy.

On clicking the OK button of the Create New Model dialog the new model is opened in your work area if it is of type system, module, or primitive.

### 2.3.2 Copying Existing Models

When copying models it is not necessary to define all the model properties. Most of the properties are defined by the original model and are not modified for the copy. There are two ways to copy existing models.

The first way is similar to the creation of new models. Either select the component in the Tree View window and select Save As from the context menu or open the module or primitive to be copied and select File - Save As. A Save As New Model dialog is displayed where the logical and the physical name of the copied model can be defined and the library to which the copied component will belong can be selected. All other properties are taken from the original model and cannot be changed.

The editable fields in the Create New Model dialog are:

* **Logical Name** - Define a logical name for the copied model. For detailed information on the logical name, see sec. 2.3.1. The default value for this field is taken from the original model.
* **Library** - Select the library to which the copied model component will belong. This determines the location of the copied model both in the physical and the logical model hierarchy. (See sec. 2.3.1.)
* **Physical Name** - Define the physical name of the copied model. If no physical name is given, it is generated from the logical name. For detailed information on the physical name, see sec. 2.3.1. The default value for this field is taken from the original model.

Once you have defined all the necessary properties click the OK button to create a copy of the model. As with the creation of new models, MLDesigner checks for the existence of models with the same physical or logical name saved in the same location. (See sec. 2.3.1).
2.3 Handling Models

The second way to copy a model is to drag & drop a model over any directory in Tree View and select **Copy** or **Duplicate** from the appearing pop-up menu. See sec. 2.2.5.7 for further details.

**NOTE:** There is a very important difference between **Save As** on the one hand and **Copy** and **Duplicate** on the other hand. **Save As** copies the model directory and only files which are necessary to describe the model, this are makefile, mml and pl file. Otherwise **Copy** and **Duplicate** copy the model directory including all files and sub-directories.

### 2.3.3 Creating Special Primitives

MLDesigner supports primitive definitions that specify how many input or output ports they have such as the SDF primitives Add found in the Arithmetic library and Fork primitive found in the Control library. Open the source code by clicking **Open Source** on the toolbar. You will see the port definition for the Add input port:

```plaintext
inmulti
{
  name {input}
  type {float}
}
```

As you can see the port is a multiple port and can accept particles with Data Type `float`.

The primitive Add has a double arrow input port when instantiated in a module or system. This is a multiple input port that allows you to connect any number of signals to it, if the domain in which the model is developed, has a built-in merge or fork primitive. It is not easy to read such block diagrams to determine which port represents what. A solution is to instantiate a special primitive where the exact number of ports is defined and graphically displayed when the special is instantiated. Each input port or output port of a special primitive has a unique identifier and can accept only one connection.

There are three ways to work with primitives that have multiple-input or multiple-output connections.

- You can draw multiple connections to or from the double-arrow port hole. This has some limitations as it is not possible to control what order the connections will actually be made in. That is not critical for an Add primitive, but for some primitive types it is important to know which connection corresponds to which element of the multiple ports. In some cases such as in the SDF domain, it is not possible to have multiple connections to a port as the domain does not contain a built-in merge primitive.

- You can attach HOF primitives (Nop) that realize “bus create” or “bus break out” to the multiple ports, choosing one that provides the right number of ports for the model. The Nop primitive fires in the order of the connections made. These primitives are available in MLD Libraries/HOF Domain in the Library View. This solves the problems with multiple connections to a single multiple input port, but may not make an easy to read block diagram.
• You can instantiate a special primitive with the specified number of simple ports. This is what the two-input Add model actually is. The result is an easy to interpret block diagram.

A number of special primitives with a predefined number of input and output ports exist in MLDesigner. These special models are not visible in the tree view because navigation of the libraries is easier without long lists of references to specialized versions of a primitive. These special primitives define the exact number of input or output ports and gives each port a unique identifier. The purpose of the special primitive is to make the systems and modules easier to interpret as each transition between multi-portholes can be clearly traced. To instantiate a special primitive, drag and drop a primitive with multiple input or output ports into a system or module. A Select Special Primitive dialog displays. You must select the appropriate special primitive from the list of existing primitives. If you do not find a special primitive with the appropriate amount of ports, you can create your own special primitive. An example of how to create a special model primitive that has 17 single input ports based on the existing Add primitive, that supports multiple inputs, follows.

• Go to \texttt{MLDLibraries/SDF/Arithmetic/Add}.
• Activate the context menu with the cursor over the Add entry in the Tree View. (This can be done in the Library View or the File View.)
• Choose \texttt{Save As}. Give the primitive a unique name and save it to one of your libraries. You can give the new special a logical name that differs from the physical name. The logical name can be set to be the default name if the correct options are selected in the \texttt{Settings} dialog. See the menu bar item \texttt{Settings - Model Editor - Set Logical Name as Label for New Instance} option. Notice the radio button option group \texttt{Specials} shown in fig. 2.20. The option \texttt{include} means that all .mml files describing the special primitive’s interface are copied to the new library.
• Find your newly saved Add primitive and activate the context menu.
• Choose \texttt{Create Special} to activate the Create Special Primitive dialog.
• The Add primitive has a multiple input port and a single output port. As a result only the \texttt{Input ports} drop-down menu is active. Select \texttt{Input} from the drop-down menu to activate the New Value dialog.
• Type \texttt{17} in the input field and click \texttt{OK}.
• In the field \texttt{Physical Name} you can see the physical name of the special primitive. If you want you can enter a logical name in the field \texttt{Logical Name}.
• If you have finished the special primitive definition you can click on \texttt{Create} button. An error message displays if a special exists with the same number of ports (and following with the same physical name).

While creating special primitives it is not possible to edit incorrect entries. To remove any specials close the Create Special Primitive dialog and activate the Delete Special Primitive dialog by choosing \texttt{Delete Special} item from the context menu. Choose the special you want to delete and click on \texttt{OK} button.

From the drop-down menus of the Create Special Model dialog shown in fig. 2.21, you can conclude that it is also possible to create special models with predefined parameter values. In this example the other two fields \texttt{Output Ports} and \texttt{Parameters} are not active. An example of a primitive with predefined parameter values can be found in the \texttt{SDF Domain/Logic} library and is called \texttt{Logic}. This is a special case of a primitive having predefined func-
2.3 Handling Models

Figure 2.20: “Save as” Including Specials

Figure 2.21: Create Special Primitive Dialog
tions where the name of the primitive determines the function it performs. The Logic primitive is instantiated by clicking and dragging the primitive into the Model Editor Window. A dialog displays and the special you need must be selected from the list. (Look at the setup() method in the primitive source code.)

Figure 2.22: Create Special Primitive - Parameter Logic

2.3.4 Create Model from Source

This section is of interest to those who have stars (primitive source code) created using Ptolemy or have written primitive source code by hand. These source code files are defined using the Ptolemy markup language and have the extension .pl. MLDesigner has a tool to convert source code files and include them in the MLDesigner standard libraries. If you have created your own Ptolemy stars and would like to incorporate them into MLDesigner, you need to save the .pl files to a $MLD_USER library, where you have read/write permission. These files will then be visible in the File View window of MLDesigner if you have checked the filter icon for primitive source files.

Activate the context menu over the relevant item indicated by the primitive name with the .pl extension. Select the menu option Create Model to open the Create Model from Source dialog. The input fields Logical Name and Library can be entered. The Physical Name field is not editable. Click OK to convert the source files to MLDesigner primitives, the parser checks for name conflicts with MLDesigner standard libraries. If a built-in primitive with the same name already exists, you will not be allowed to convert the .pl file to a user defined built-in primitive.
2.3 Handling Models

unless you change the physical name and edit the appropriate reference to the filename in the source code.

To convert source code files with name conflicts to an MLDesigner type, proceed as follows:

- Open the source code of the Filename.pl file.
- The reference to the name is normally in the first couple of lines of code and looks like this

```plaintext
defstar {
    name { Filename }
}
```

- Change the Filename in the source code to Filename new.
- Save the file as Filename_new.pl in an MLDesigner user library.
- You can now select the primitive file in the File View window of MLDesigner and activate the context menu with the right mouse. (If you do not see the source code file, check the Show Primitive Sources icon in the tree view filter toolbar).
- Choose Create Model from the context menu. You can now create a MLDesigner primitive with the same physical name as the primitive source code file Filename_new.

2.3.5 Open Existing Models

There are different methods for opening existing models in both the File View or the Library View:

- Double-click the model item in the Library or File View or select Open Model from the context menu.
- Use the file menu or the tool button Open Model.

The context menu item is only available if the selected item refers to a model of type library, system, module, or primitive including FSM primitives.

**NOTE:** Consider which Tree View filters are activated by the check buttons in tree filter toolbar. (See sec. 2.2.5 and sec. 2.2.9.2.)

If the selected item refers to a primitive source file, you can use the context menu item Open Source to open the primitive source in a new Primitive Editor. If the selected item refers to an FSM model, you can use the context menu item Open FSM to open the FSM model within a new FSM Model Editor. Using the context menu has the same effect as double-clicking on the relevant item in the Tree View window.

Another way to open a model is to use the file menu item File - Open which has the same effect as using the tool button Open Model. This opens a dialog from which you can select the model component you wish to open (See figref:fig:3-28). After selection, click the OK button to open the model. If the selected item refers to a primitive source file, it is opened within a Primitive Source Editor. If the selected item refers to an FSM model, MLDesigner opens the file with the FSM Model Editor.
2.3.6 Model Update

Whenever you open a library, primitive, module, or system in MLDesigner, the XML file (.mml) is parsed by the version mapper. If the version number is lower than the actual number of your installed version of MLDesigner, a Model Update dialog is displayed (see fig. 2.23). If the version number is higher than the actual number of your installed MLDesigner version, the model cannot be opened.

In case of a lower version number, there are two options to choose from:

- **Save** - The model version is updated to the current version number and all other changes are updated.
- **Cancel** - A temporary file is generated ensuring that you can run simulations and use the model instance in systems for the duration of your MLDesigner session. On shutdown of MLDesigner, all references to the model are deleted. You will be prompted to update the model when you attempt to open it in a later session using MLDesigner.

You can save the model to ensure faster loading the next time you open the model and to avoid being prompted to update the model every time you open it.

![Figure 2.23: Update Model dialog](image)

2.3.7 Deleting Models

When deleting models from libraries or model bases you must remember the differences between the File View and the Library View.

As described in sec. 2.2.5 the Library View is a view of the model base hierarchy resulting from
2.3 Handling Models

Logical relations between models. Deletion of a logical relation is only possible in Library View by selecting the **Remove Reference** item from the context menu over the model reference.

To physically delete a model, you must select the item **Move to Trash** from the context menu under the File or Library View tab. The model is moved to Trashcan. From the Trashcan it is possible to restore deleted models or permanently remove them from the system. You can also remove models from the File View by moving them to the Trashcan using the drag and drop function. It is also possible to select more than one item for deletion by pressing Ctrl while selecting items with the mouse.

**NOTE:** Before deletion of a model component ensure that the deleted model component is not open in a Model Editor Window.

Remember, the same model can be instantiated or referenced by more than one model. Deleting a model physically before removing all references to this model in other model libraries can result in library inconsistencies. Such inconsistencies are indicated in the library by a broken icon. It is possible to remove the broken icon using the context menu option **Remove Reference**.

Deleting a model physically can lead to inconsistencies if other models contain instances of the deleted model. If you open a model containing instances of models that no longer exist, an error message is displayed and the model cannot be opened.

After deletion of a model it is sometimes necessary to update the tree view affected. To do this select **Refresh** from the context menu in the Tree View.

2.3.8 Printing Models

You can print models of type library, system, module, and primitive as well as FSM models if they are open in the active Model Editor Window.

**Printing**

To print the model in the active Model Editor Window, use either the file menu item **File - Print** or the tool button. The print job commences immediately. The Print dialog is used to configure the printer as well as the document before printing (see next section). Once you are happy with your printer settings, click **Print** in the Print window.

The output format used for the printing function is PostScript. This is the most common printing format for UNIX. Normally the printer supports PostScript format or processes are running in the background to convert the printed PostScript to a language supported by the printer. The printer device can also print the model to a PostScript file. fig. 2.24 shows an example of printed document.

**Printing Configuration**

Before you start printing, you should configure the printer device as well as the printed document by using the **Print** dialog.

**NOTE:** The configuration of the printed document also affects the export of the vector graphic.
Figure 2.24: Printed model example

to a file.

Figure 2.25: Print setup dialog

Use the file menu item **File - Print** to open the print configuration dialog (see fig. 2.25). To configure the printer click the button **Printer Setup**. This opens the printer configuration dialog (see fig. 2.26).

In the Printer Setup dialog, you can select printers and printer related options such as paper size and paper orientation.

- **Print destination** - Define the destination of the printed model.

  **Print to printer**  Check this radio button to define the printer as destination for printing. If this option is activated, you can select the printer from the list of known printers shown below this radio button.
2.3 Handling Models

Print to file Check this radio button to save the output in a file in Postscript format. You can either edit the file location line or click the Browse button to select a directory and specify a filename.

- **Printer settings** - Define whether printer is color or grey-scale. Normally, the default option can be used, since all monochrome printers can interpret color output.
- **Paper format** - Define the paper format. This option is used to define the printing orientation and paper size.
- **Options** - Define several miscellaneous options. Some options are only available if document is larger than one page.
  - Define whether you want to print all pages or certain pages.
  - Define the order of printed pages.
  - Specify the number of copies of the printed document. Usually the value of this option is 1.

![Printer Configuration Dialog]

Figure 2.26: Printer Configuration Dialog

Click OK to return to the Print dialog.

- **Headline** - Define the headline text for the printed document. The check box to the left of the edit field determines whether a headline is generated. If selected you can define the headline text format. The headline is centered at the top of the printed document. You can enter your own headline or use the predefined symbols to generate the headline. The predefined symbols for generation and formatting of headlines are explained here.

  - **$NAME** Includes the physical location, the model component type, and the logical name (if there is one). The information is the same as shown in the headline of the Model Editor Windows.

  - **$AUTHOR** Includes the author of the model. The author is the owner of the physical files of the model component identified by the UNIX user name. If a real
name for the UNIX user is registered, the real name is used.

$\text{DATE}$  
Includes the last modification date of the physical files of the model component.

$\text{TIME}$  
Includes the time the physical files were last modified.

- **Footline** - The check box to the left of the edit field determines whether a footline is generated. If selected you can define the footline text format as described under **Headline**. The footline is centered at the bottom of the printed document.

- **Draw Bounding** - Define whether a bounding box is drawn around the model graphic.

- **Scale** - Define the scaling of the model. Here you can define the size of the printed document. By checking the box **Keep Aspect Ratio**, the ratio between width and height is preserved. There are three options for modifying the scale.
  
  - **Pages**  
  Modify the size given by the number of pages.

  - **Size**  
  Modify the size given in millimeter.

  - **Zoom**  
  Modify the size given as a percentage of original size.

Once you are happy with your settings click **Print**.

### 2.3.9 Exporting EPS

You can export a models as a vector graphic. Select **Export EPS** from the **File** menu. MLDesigner opens a file selection dialog where you can select the directory and specify the filename. The format of the exported vector graphic is Encapsulated PostScript (EPS). EPS is a text based standard format for exchanging and embedding vector graphics into documents. There are tools available to convert the resulting EPS file into other formats. It is possible to embed the exported vector graphic representation of the model into different document formats. To format the EPS file use the Printer Setup dialog (See above).

### 2.4 Shared Libraries

It is possible to develop a library within the environment specified by the **$MLD\_USER$** environment variable and then move this library to the shared environment specified by the **$MLD\_SHARED$** environment variable. However, since the **$MLD\_USER$** variable is dynamic (because it can be different for every user) the following prerequisite applies.

- All modules and files needed by systems in the library must be located within this library, or in a location that never changes, such as the directory to which **$MLD\_SHARED$** and **$MLD$** point.

The reason for this is that the **systemName.mml** file contains references to all model elements needed for the system to function. If these variables change then MLDesigner will not be able to locate the missing model elements.
2.4 Shared Libraries

When starting MLDdesigner the environment variable settings are printed in the shell. For those that need to share their work with a workgroup, the MLD_SHARED environment variable should point to a directory where all members of the workgroup have read/write rights.

**NOTE:** This directory is only visible in the File view if the MLD_SHARED environment variable has been set.

To set the environment variable type the following:

For bash or shell command lines enter:

```bash
export MLD_SHARED="/Directory/.."
```

and for tcsh and csh command lines enter:

```bash
setenv MLD_SHARED="/Directory/.."
```

### 2.4.1 Exporting a Top Level Library to Shared Libraries

There are two methods for moving a Library to the Shared Libraries directory or any other top level directory. The first is to tar the library into an archive and untar it in the appropriate location. The second method is exactly the same as the first except it is performed using the MLDdesigner GUI. The library is first exported as a tar archive using the context menu over the library and then imported into Shared Libraries using the context menu over the directory folder in the tree view.

The dependency checking mechanism ensures that all models used or referenced by models in the library you are exporting are listed as separate entries in the Export Library dialog. You can choose whether to export all libraries listed or not. To deselect a listed library click with the cursor in the appropriate check box to remove the tick sign.

### 2.4.2 Export Libraries

To export an archive activate the context menu over a top level library in the tree view and select Export Library from the menu (see fig. 2.27). Exporting of libraries is restricted to top level libraries only. The following formats are supported:

* .tar  
* .tar.gz  
* .tar.Z

The specified directory should be a directory where read/write rights have been set for a workgroup.

#### 2.4.2.1 Protecting Intellectual Property

Intellectual property can be protected if needed. Deselecting the include .pl files check box, in the Export Library dialog, ensures that only compiled objects (.o) files are exported to the archive. Note that precompiled systems are platform dependent. It is not possible to export precompiled objects and execute the simulation on another platform. To do this the entire library must be re-compiled after being exported and untarred.
With extremely large systems the size of the archive could be an inhibiting factor. To reduce the size of the archive exclude the .htm online documentation and the .o files.

![Export Top Level Library Dialog](image)

**Figure 2.27:** Export Top Level Library Dialog

### 2.4.3 Import Libraries

The Import Libraries option is only available in the context menu over the top level libraries **My Libraries** and **Shared Libraries** in the File view.

**NOTE:** Ensure that all modules or primitives used by systems in the top level library are stored in that library. If this is not the case the library that contains the module or the module itself must also be exported to the shared library. If the module is exported to the library the reference to it must be changed by re-instantiating the module in the relevant system.

To import .tar archives to the top level Directories **MyLibraries** and **Shared Libraries** choose the menu option **Import Library** from the context menu in **File** view. The import dialog shown in fig. 2.28 allows selection of tarred and zipped archives with formats 


The untaquared archive will appear as a top level library in the tree view.
2.4.4 Environment Variables and Dynamic Referencing Mechanism

For a better understanding of the dynamic referencing mechanism read the following example:

Let's assume, you design a system "S1" in a top level library "L1" that uses an instance of a module "M1" from the same top level library "L1". The reference to the module in the "S1.mml" file is stored as follows:

\[
\text{class} = "\text{file:$MLD\_USER/L1/M1/M1.mml}"\]

However, when you instantiate a module "M2" from another top level library "L2", the reference is

\[
\text{class} = "\text{file:$MLD\_USER/L2/M2/M2.mml}"\]

That is, the reference points to a location that depends on the user's setting of the $MLD\_USER variable.

What happens if you started to develop your model with $MLD\_USER variable set to "/home/shared/MLD" and then changed the environment variable to "/home/users/MLD". The reference would point in our example to

\[
\text{class} = "\text{file:/home/shared/MLD/L2/M2/M2.mml}"
\]

in the first case and
in the second. This would not be a problem if library "L2" would already be moved to $MLD_SHARED, since the reference would be

```
class="file:$MLD_SHARED/L2/M2/M2.mml"
```

and this should be the same for all users. In other words developing a library within $MLD_USER and then moving it to $MLD_SHARED works for top level libraries as long as all models inside the library use modules from the same library or from a library located at $MLD_SHARED.

A top level library is located directly below $MLD_USER or $MLD_SHARED. All the libraries you can see as top level items in the library view are top level libraries.

### 2.4.5 Set User Environment Variables

It is possible to set your $MLD_USER environment variable to point to a project library or external library. Let's assume you want to work on a project called $MLD.project. This project is the $MLD_USER directory of another user. You want to access the systems and share libraries that exist in the other user’s environment. Enter the following command where you would normally open MLDesigner.

For bash or shell command lines enter

```
export MLD_USER=~/MLD.project
```

And for `tcsh` and `csh` type command lines.

```
setenv MLD_USER ~/MLD.project
```

You have now changed your environment. Shut down and restart MLDesigner as usual. The libraries you see will be those of the other user and not your own. You could also do the same.

![Figure 2.29: Library Structure and System](Image)
locally on your own computer if you wanted to separate libraries and projects. You can create a new directory ~/MLDProject in your home directory. Set your MLD_USER environment variable to point to the new directory. When you open MLDesigner again you will see the tree view with MLDesigner libraries and no user libraries. You could then create a top level library with read and write rights for a workgroup on your network.

At the moment there is no version control mechanism within libraries when numerous designers are working on one project. As a result the team members must communicate with each other regarding who is going to make changes to the systems or libraries and when. With version 2.3 a mechanism will be in place where copies of files will be saved with version numbers and the file will be read only for all other members of a team as soon as a module or system is opened.

NOTE: If you set MLD_SHARED variable to model libraries that are normally stored in $MLD_USER, those models contain a lot of references to $MLD_USER/... When you now copy a model from MLD_SHARED to MLD_USER, MLDesigner tries to replace all $MLD_SHARED/... (the origin of the copy operation) by $MLD_USER/... If you have a different library structure in your models, references can become invalid by such an operation.
Chapter 3

Developing Models

In sec. 2.3, the handling of model components, including their creation, opening and deletion was described. This section describes the construction of functional models for model components of type library, system, module and primitive. It does not describe the construction of FSM models. For detailed information on how to develop FSM models, refer to ch. 24.

3.1 Introduction

MLDesigner allows you to construct models schematically using design windows. The schematic of a model is also known as a Block Diagram or System. “Design windows” (otherwise known as block diagram editor) provide you with graphical model construction capabilities. By placing model instances (blocks or modules) in the design area, and connecting them, the functional behavior of the model component is represented schematically. The model instance can reference predefined library modules or new modules, whose functional model can be provided later, in “top-down fashion”.

In addition to the model instances and the connections between them, a model can contain other model elements such as:

- model instances
- input and output ports
- shared elements
- connections
- parameters and target parameters
- text labels
- delay properties, and
- bus properties.

Note that not all these model elements can be used in all types of models. The type of model and the modeling domain of the model component determine what kind of model elements can be used for functional modeling.
3.2 Steps to Develop Models

Models are manipulated in Model Editor Windows by using the command options found in the edit menu, the context menu, or on the editor toolbar. The construction of a model using MLDesigner normally involves some or all of the following tasks:

- Creating or opening a model;
- Editing the common model component properties (see sec. 3.3);
- Adding input and output ports (see sec. 3.4);
- Adding parameters and specifying the default values (see sec. 3.5);
- Adding shared elements (see ch. 10);
- Adding model instances and editing their placement (see sec. 3.6);
- Setting parameters of model instances;
- Connecting model instances and placing connection property elements (see sec. 3.8);
- Saving the model.

To construct a new model you must first create the model components

**sinMod Example. Step 1.**

This example takes you through the steps required to create a complete model.

The first step is to create a library to group all the components. You will then create the System and Module components. This is done as follows:

1. Click the tool button **New Model** to open the **Create New Model** dialog.
2. Select **Library** from the **Type of Model** drop-down menu.
3. Enter the **Logical Name** as Sine Modulator Library.
4. Leave the **Library** field empty.
5. Type **This library contains a sine modulator** as **Description**.
6. Click the **OK** button.
   
   You now have a top level Library in the **Library** view with the name **Sine Modulator Library** and in the **File** view with the name **Sine_Modulator_Library** as a sub-entry of the **MyLibraries** directory.

7. Click the tool button **New Model** to open the **Create New Model** dialog.
8. Select **System** from the **Type of Model** drop-down menu.
9. Enter the **Logical Name** as Sine Modulator System.
10. Select the library Sine Modulator Library.
11. Select **SDF** from the **Modeling Domain** drop-down menu.
12. Enter **This is the sine modulator system model.** as **Description**.
13. Click the **OK** button.

14. Click the tool button **New Model** to open the **Create New Model** dialog.
15. Select **Module** from the **Type of Model** drop-down menu.
16. Enter the **Logical Name** as Sine Modulator.
17. Enter **This is the modulator model.** as **Description**.
3.3 Modifying Model Properties

18. Click the OK button.

19. Use the tool button **New Model** to open the **Create New Model** dialog.

20. Enter the **Logical Name** as **Sine Generator**.

21. Enter **This is the generator model** as **Description**

22. Click the OK button.

Now you have created the model components including the library needed for this example. As explained in ch. 2.3, MLDesigner opens a Model Editor Window for each created model component excluding libraries. You should now have three design windows open in the workspace.

The models created for our example have predefined properties according to the values we specified in the **Create New Model** dialog (see fig. 3.1). To view the parameters in the Property Editor for these models, switch to the relevant Model Editor Window. Example continued on 3-10.

(a) Sine modulator system

(b) Sine modulator module

Figure 3.1: Model properties of the example

### 3.3 Modifying Model Properties

Every model has a number of properties in common. The properties assigned to a model depend on the type of model component it belongs to. Table 3.1 summarizes the common properties and their availability for the different model types. These properties can be edited in the **Property window**. The model properties are:
**Logical Name**
Defines the logical name of the model component. The logical name is used to identify the model component in the Library view. The logical name is shown in the headline of the Model Editor Window and is a string.

**Model Type**
Defines the type of the model component. This could be of type *Primitive*, *Module*, *System*, or *Library*. It is not possible to change the value of these properties once the component or module has been created.

**Copyright**
Defines the copyright notice. For primitives, the copyright is used to generate the copyright item in the primitive source code. This property is a free text string.

**Version**
Defines a version number string. For primitives, the version string is used to generate the version item and must follow the conventions for the version string described in sec. 13.5.1.5. For all other types of model component the version string is free text.

**Domain**
Defines the modeling domain, that determines the model of computation for your model component. The property value is selected using a drop-down menu which contains entries for all known domains. If you select an experimental, unsupported domain, a warning message is displayed. For primitives this property is used to generate the domain item in the primitive source code. This property cannot be changed in primitive models.

**Target**
Defines the target or top-level manager for the execution of the model. This property is only available for systems and modules. The value is selected from a drop-down menu. The drop-down menu contains all possible targets applicable to the selected model component/domain.

**Import Libraries**
This field is normally completed. This is necessary, if you want to use composite data types (data structures) for model elements like parameters, ports and shared model elements (see sec. 12.3.5). You have to import the libraries before you can use the relevant data structure types.

**Load Mode**
Defines the load mode. This property is only available for Primitives and Probes. FSM primitives do not use this option. You can select the load mode as *Dynamic* or *Permanent*. If you select load mode *dynamic*, a shared library containing the primitive code is created on loading. This shared library is linked dynamically to MLDesigner so the primitive can be reloaded after changes are made to it. In contrast, by selecting load mode *permanent*, the primitive is linked to MLDesigner on startup as is the case with built-in primitives (MLDesigner Libraries). Changes to permanently loaded Primitives or Probes only take effect after shutting down and restarting MLDesigner.

**NOTE:** It only makes sense to set the load mode to Permanent if you are going to create new primitives derived from the permanently loaded primitive. In such cases the permanently loaded primitive is known as the *Base primitive*.

**Derive**
This property is only available for Primitives and Probes. A *Select Prim-
3.4 Modeling Input/Output Ports

When creating new model components, you must define its interface, consisting of input and/or output ports, parameters and shared model elements such as events, memories and resources. Therefore, the first step is usually to create port objects for each model component of type module,
3 Developing Models

<table>
<thead>
<tr>
<th>Property</th>
<th>Library</th>
<th>System</th>
<th>Module</th>
<th>Primitive</th>
<th>FSM</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Name</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Model Type</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Copyright</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Version</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Domain</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Target</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Import Libraries</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Mode</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derive</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compile Options</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Documentation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.1: Common properties for different model types

primitive and FSM primitive. This section describes how to create input and output ports.

### 3.4.1 Introduction

Ports are used to connect model instances for exchanging data objects. The kind of porthole and the type of data objects accepted or produced by the model component to which the port belongs must be defined. With primitive model components, ports can be single portholes or multiple portholes. Table 3.2 shows the possible combinations of port types as well as the keywords used for ports in the primitive sources. At this stage inoutmulti and inout ports are not supported by MLDesigner.

**NOTE:** Ports of FSM primitives can only be single portholes.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Port number</th>
<th>Icon representation</th>
<th>Type name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Single</td>
<td>single arrow</td>
<td>input</td>
</tr>
<tr>
<td>Input</td>
<td>Multiple</td>
<td>double arrow</td>
<td>inmulti</td>
</tr>
<tr>
<td>Output</td>
<td>Single</td>
<td>single arrow</td>
<td>output</td>
</tr>
<tr>
<td>Output</td>
<td>Multiple</td>
<td>double arrow</td>
<td>outmulti</td>
</tr>
</tbody>
</table>

Table 3.2: Possible data type combinations
Data objects that pass from one port to another in MLDesigner are called particles or tokens. There are different types that are supported by MLDesigner (see table 3.3). A model that operates on anytype particles is said to be polymorphic. Polymorphic models operate on multiple types of data. For example, a Printer primitive can produce a textual representation of any type of particle.

MLDesigner usually makes conversions between numeric particle types automatically. The float to complex conversion does the obvious thing, putting the float value into the real part of the complex number and setting the imaginary part to zero. The complex to float conversion computes the magnitude of the complex number. integer to float conversion is easy enough. float to integer conversion rounds to the nearest integer.

In some situations, automatic type conversions cannot be made. A common difficulty involves several outputs of different types feeding a Merge primitive. MLDesigner must assign a specific type to the Merge primitive’s output, but in this case it will be unable to decide which type to use. An error message is displayed stating “cannot determine DataType” for the output. The solution is to insert one or more types of conversion primitives, so that all the values arriving at the Merge primitive have the same type. The conversion primitives can be found in the Conversion libraries of the appropriate domain. Some domains are more restrictive about particle type conversions than others. Assignment of types to anytype portholes and resolution of type conflicts is discussed further in sec. 14.5.

The type of port and type of data objects consumed or emitted by the port determines the appearance of the port. fig. 3.3 shows different types of ports and illustrates the type of data compatible with each port type. A port definition in the model component is also called a formal port.

The black triangle inside the formal port is the point where you connect it to ports of model instances within the model. The instance represents the model component if it is embedded into another model. The port representation of an instance, the so-called actual port, serves as a connection point for connecting model instances and is sometimes called a terminal.

- The position of the black triangle indicates whether the port is input or output.
- The color of the port indicates the type of data objects consumed or emitted by the port (see table 3.3).
- The number of arrows indicates whether the port is a single input-output port “one arrow” or multiple input-output port “two arrows”.
- The stem inside the port representation indicates whether the port accept or emits scalar or
### 3.4.2 Creating Ports

By using the tool buttons **Add Input Port** and **Add Output Port**, you can switch to input / output port creation mode indicated by a special cursor. Click the tool button **Select Tool** or the right mouse to deactivate port creation mode. In port creation mode, every click on the model background will create a new input or output port of type `anytype`. Each port gets a unique name. You can change the properties later using the Property Editor window.

### 3.4.3 Changing Port Properties

You can use the property editor to change the port properties. With a click on a port, the **Port Properties** plane in the Property Editor window becomes active and the current properties of the selected port are shown (see fig. 3.4). The properties in this plane are:

**Name**

If you change the name, you have to take care that the name is unique for the model. The port name has to be a valid identifier, i.e., it can contain letters, digits and the underscore `_` but must start with a letter or an underscore.

---

<table>
<thead>
<tr>
<th>Data type</th>
<th>Data structure</th>
<th>Port color</th>
<th>Type name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any type</td>
<td>Scalar</td>
<td>red</td>
<td>anytype</td>
</tr>
<tr>
<td>Floating point</td>
<td>Scalar</td>
<td>blue</td>
<td>float</td>
</tr>
<tr>
<td>Floating point</td>
<td>Matrix</td>
<td>blue</td>
<td>float_matrix_env</td>
</tr>
<tr>
<td>Complex</td>
<td>Scalar</td>
<td>white</td>
<td>complex</td>
</tr>
<tr>
<td>Complex</td>
<td>Matrix</td>
<td>white</td>
<td>complex_matrix_env</td>
</tr>
<tr>
<td>Integer</td>
<td>Scalar</td>
<td>orange</td>
<td>int</td>
</tr>
<tr>
<td>Integer</td>
<td>Matrix</td>
<td>orange</td>
<td>int_matrix_env</td>
</tr>
<tr>
<td>Fixed</td>
<td>Scalar</td>
<td>violet</td>
<td>fixed</td>
</tr>
<tr>
<td>Fixed</td>
<td>Matrix</td>
<td>violet</td>
<td>fixed_matrix_env</td>
</tr>
<tr>
<td>Message</td>
<td>Scalar</td>
<td>green</td>
<td>message</td>
</tr>
<tr>
<td>String</td>
<td>Scalar</td>
<td>black</td>
<td>string</td>
</tr>
<tr>
<td>File</td>
<td>Scalar</td>
<td>yellow</td>
<td>file</td>
</tr>
<tr>
<td>Data Structure</td>
<td>Scalar</td>
<td>cyan</td>
<td>datastruct:</td>
</tr>
<tr>
<td>Continuous</td>
<td>Scalar</td>
<td>light blue</td>
<td>continuous</td>
</tr>
</tbody>
</table>

Table 3.3: Possible data type combinations
3.4 Modeling Input/Output Ports

**Port Type**
Defines the type of port. This property defines whether the port is input or output as well as whether it is a single or multiple porthole. Possible values are defined in table 3.2. For modules and FSM primitives, the values *input* and *output* are available. For primitives, the values *inmulti* and *outmulti* are also possible.

**Data Type**
Defines the type of data objects (particles) consumed by the port. This value defines the data type as well as whether the port accepts scalar values or matrices. Please refer to table 3.3 for possible values.

**Data Structure**
If you selected the data type *datastruct*, you can use this property to define which composite data type the port should accept or emit. In all other cases this property is not editable. Use the button in the input field to open a dialog for selecting the composite data type.

**Description**
The description is used for generating the hypertext documentation of the model component.

**Instance Label**
Using this property you can define the text label of the port as it is shown on instances of the model component. If you do not define an instance label, the port is not labeled on instances.

**Visibility**
Defines whether a port is visible or not. This attribute is normally used in conjunction with derived ports where a model component should hide a port that is derived from another model component.

**Alignment**
Sets the alignment of the port within the model component. With the cursor in add input/output port mode the port will be positioned according to which border you click on. The position of the port can be changed by altering the settings in the Property Editor.

**Token Number**
This property is only available for primitive ports. It defines the number of particles consumed or produced by the primitive on this port. This property is only applicable to data flow domains such as the SDF domain.

![Figure 3.4: Port property editor plane](insert image here)
sinMod Example. Step 2.

Continued from page 3-2

The next step in this example is to create ports for Sine Modulator and the Sine Generator modules created on 3-2. Ports cannot be defined in the system model Sine Modulator System.

1. Select the Model Editor Window containing the Sine Generator module.
2. Use the Add Output Port tool button to switch to port creation mode.
3. Click on model background at the position where the port will be created.
4. Click the right mouse button to switch back to normal selection mode.
5. Select the created port. MLDesigner will show port properties in the property editor.
6. Set the port name to Output.
7. Select the data type float
8. Set the description to The sine signal output.
9. Save the model using the tool button Save Model (Ignore construction error messages).

10. Select the Model Editor Window that contains the Sine Modulator module.
11. Use the Add Input Port tool button to switch to port creation mode.
12. Click on model background at the position where the port will be created.
13. Click the right mouse button to switch back to normal selection mode.
14. Select the created port. MLDesigner will show port properties in the property editor.
15. Set the port name to Input.
16. Select the data type float
17. Set the description to The input signal.

18. Remain in the Model Editor Window containing the Sine Modulator module.
19. Use the Add Output Port tool button to switch to port creation mode.
20. Click on model background at the position where the port will be created.
21. Click the right mouse button to switch back to normal selection mode.
22. Select the created port. MLDesigner will show port properties in the property editor.
23. Set the port name to Output.
24. Select the data type float
25. Set the description to This output signal.
26. Save the model using the tool button Save Model (Ignore construction error messages).

Figure 3.5: Example models after the creation of the ports.
3.5 Definition of Parameters

The next step is to define the parameters of the model. Parameters can be defined for model components of type system, module, and primitive, but not for libraries and FSM primitives.

3.5.1 Introduction

Parameters are used to control the functionality of model components. **Formal** parameters of modules or primitives define the interface on embedding into other modules or systems. The corresponding parameters that appear in instances of the model component are called **actual parameters**. The terms formal and actual parameters are analogous to formal and actual parameters in any procedural programming language. See sec. 3.5.3 for more details.

**System parameters** are used to configure a system model for execution. Such parameters are used, for example, for simulation trade-off analysis to configure a set of different simulation runs. For that purpose, a set of values can be defined for each system parameter.

![Figure 3.6: Property editor for defining new parameters](image)

For models that have defined targets i.e. systems and modules, special types of parameters called **target parameter** are used. Target parameter are used to configure the target that controls the execution of the model and not the model itself. Target parameters are predefined by the target and cannot be created by you.

Parameters as well as target parameters of a model are shown as a list of parameter properties in the Property Editor windows. Normal parameters are indicated by a leading [P], whereas target parameter are shown by a leading [T] string.

3.5.2 Creating Parameters

There are two different ways to create new parameters. You can either:
3 Developing Models

- create a new formal parameter using model properties, or
- use an actual parameter of an existing model instance and export it to the model.

Creating a new parameter

To create a new formal parameter, you must ensure that the **Model Properties** plane is activated in the Property Editor window by clicking on the model background. Open the context menu in the Property Editor (see fig. 3.7 a). Select **New Parameter** to create a new parameter entry in the model property list. MLDesigner creates a default collapsed entry that shows the new parameter name and a default value. MLDesigner generates a unique name for the new parameter that can be changed later. Expand the view by clicking the \[+\] to the left of the new parameter if necessary.

![Figure 3.7: Context menus for creating formal parameters](image)

3.5.2.1 Exporting a Model Instance Parameter

By selecting a model instance, the property editor will show entries for all parameters defined by the referenced model component. Click with the right mouse button on any of these parameter entries to open the context menu shown in fig. 3.7 b. Then you can select menu items **Export** or **Export as** to create a new formal parameter for the model with the same parameter properties. The difference between **Export** and **Export as** is that you can define the name of the exported parameter in the latter case. If a parameter with the name already exists MLDesigner generates a unique one by extending the specified name with an integer.

Exporting an instance parameter automatically links the actual instance parameter to the exported formal model parameter (See sec. 3.7.5).

3.5.2.2 Changing Parameter Properties

The property editor can be used to change the properties of formal parameter definitions. Figure 3.6 shows the expanded parameter entry.

**Name**  
If you change the name, you must ensure that the name is unique for the model. The port name has to be a valid identifier, i.e., it can contain letters, digits and the underscore \_ but must start with a letter or an underscore.
3.5 Definition of Parameters

**Data Type**
Defines the data type of the formal parameter. Table 3.4 shows the data types for parameters supported by MLDesigner. The parameter types *fix* and *precision* are only allowed for primitive model components.

**Data Structure**
If you selected the data type *datastruct*, you can use this property to define which composite data type the parameter has. Click the button on the right of the input field to open the dialog for selecting the composite data type. In all other cases this property is not editable.

**Value**
Defines the default value. This value is used for model instances where the default value is not set explicitly. For more detailed information about the syntax of parameter values, see sec. 3.7.

**Attributes**
Defines parameter attributes. Attributes are only used for primitive models. Click the button on the right of the input field to open the dialog where you can select the parameter attributes.

**Scope**
Defines whether a parameter is visible in the simulation control window. This property is only available for system parameters. A parameter must be *External* if you want to control it in the simulation control window.

**Description**
The description is used in the hypertext documentation of the model component.

---

**3.5.3 Deleting Parameters**

Activate the *Model Properties* plane in the property editor by clicking the background of the model. Then, click with right mouse button on the parameter entry in the property editor window to open the context menu shown in fig. 3.8. Select the *Delete Parameter* context menu option to delete the parameter.

---

**sinMod Example. Step 3.**

Continued from page 3-10

The next step in this example is the creation of parameters *Frequency* and *SampleRate* for our *Sine Generator* module. Switch to the Model Editor Window that contains the *Sine
3 Developing Models

Generator module and follow the steps below. Exporting parameters will be demonstrated for
the creation of the parameters of the Sine Modulator module in sec. 3.7.

1. Select the Model Editor Window that contains the Sine Generator module.
2. Click with right mouse in the property editor and select the item **New Parameter**.

3. Click on the plus sign to the left of `new_parameter_1` to expand the item (if necessary).
4. Set the name to `Frequency`.
5. Set the type to **float**.
6. Set the default value to `PI/50`.
7. Set the description to The frequency of the generated sine wave.
8. Click on the minus sign to collapse the changed parameter `Frequency`.

9. Click with right mouse in the property editor and select the item **New Parameter**.
10. Click on the plus sign to the left of `new_parameter_2` to expand the item (if necessary).
11. Set the name to `SampleRate`.
12. Set the type to **float**.
13. Set the default value to `2*PI`.
14. Set the description to The sample rate of the module.
15. Click on the minus sign left to the changed parameter `SampleRate`.

16. Save the model using the tool button **Save Model** (Ignore error message).

3.6 Adding Model Component Instances

To specify the functionality of the model you must create a number of model instances and connect them. This section describes the creation of model instances as well as the capabilities of MLDesigner to handle the model instances.

3.6.1 Add Model Instance

There are different ways to create new instances.

- Using the drag and drop capability of the tree view item.
- Using the copy and paste capability between Model Editor Windows.
- Using the context menu item **Add Instance** of the Model Editor Window.
- Using the tool button **Add Model Instance**.

The easiest way to create a new model instance is to select the item in the File or Library view and drag the item onto the Model Editor Window. A model instance that refers to the model component referenced by the tree view item is created in the model Model Editor Window.

It is possible to copy and paste model instances between different Model Editor Windows. You can select and copy model instances from one Model Editor Window into the clipboard and then paste the contents of the clipboard into another Model Editor Window. If complete connections belong to the selection, they are copied, too. Connections are complete if all the model instances...
3.6 Adding Model Component Instances

to which the connection is attached were selected.

The third way is to click the right mouse button at the position where you would like to create a new model instance. Select the context menu item **Add Model Instance** to open a **Select Model** (see fig. 3.9). From this dialog you can select the model component you would like to instantiate in the active Model Editor Window.

![Select Model Dialog](image)

**Figure 3.9: Select model dialog**

By using the **Add Model Instance** tool button, MLDesigner switches to the block creation mode indicated by a special cursor. In block creation mode, you can create new model instances at the current cursor position wherever you click on the Model Editor Window background. As with the creation of model instances using the context menu a **Select Model** window is displayed from which you can select the model component to be instantiated. Right click the mouse to cancel block creation mode or click the tool button **Select Tool**.

MLDesigner generates a unique name for every created model instance. This name is generated using the physical name, also called “class name”, of the model component and extending it with an integer preceded by a hash (#). On selecting a model instance the **Instance Properties** plane in the property editor window is activated. The property **Name** shows the generated name of the model instance. The name is a model instance property that cannot be changed. It is used to identify the model instance uniquely within the hierarchical system model during simulation, see ch. 7.

### 3.6.2 Setting Text Label

Instances of model components are labeled by default with the model instance name. Sometimes, the default label of a model instance does not contain enough information about what the instantiated model component does. In that case, it is better to label the model instance with a descriptive text.

The quickest way to add a descriptive label to a model instance is to position the cursor over the existing label and click the mouse two times slowly. The first click highlights the text area with a broken grey border. The second click highlights the text in the text area. You can now type over
3 Developing Models

the existing label. You can use the line breaks within the text area. You will see the model instance expanding to accommodate the text.

On selecting a model instance MLDesigner activates the **Instance Properties** plane in the property editor window. Select the property **Label** to add a short description. Click the button to the right of the input field to open a text editor where you can type text labels that need more than one line. If you delete the text in the label property the default label is shown again.

**NOTE:** Another method of labeling model instances is to enter the details in the Description field of the Instance property editor. The model instance remains small in the Model Editor Window, and the detailed information is available via tool tip text or online documentation.

### 3.6.3 Placement of model instances

After the creation of model instances they can be placed anywhere in the model using the drag and drop function.

You can copy selected model instances by pressing **Ctrl** while clicking and dragging the model instance.

**sinMod Example. Step 4.**

Continued from page 3-13

The next step is to include the model instances for the model components we need for the sine modulator example. Follow the steps below to add the instances for the Sine Generator, Sine Modulator modules, and system model.

1. Open the Sine Generator module.
2. Select the Library View tab in the tree view.
3. Expand the library MLD Libraries/SDF Domain/Sources.
4. Drag item Ramp into the Model Editor Window.
5. Expand the library MLD Libraries/SDF Domain/Nonlinear.
6. Drag item Sin into the Model Editor Window.
7. Arrange the components in the Model Editor Window as shown in diagram (a)
8. Save the model using the tool button **Save Model**.
   Ignore the construction errors. The error is displayed because there is no output/input port for the module and/or the ports are not connected or terminated.

9. Open the Sine Modulator module.
10. Select the Library View tab in the tree view.
11. Expand the library MLD Libraries/SDF Domain/Arithmetic.
12. Drag item Mpy into the Model Editor Window.
13. From the Select Special Primitive dialog select **Mpy.input=2**
14. Expand your library Sine Modulator Library.
15. Drag item Sine Generator into the Model Editor Window.
16. Arrange the components as shown in figure (b).
17. Save the model using the tool button **Save Model** (Ignore construction error messages).

18. Open the Sine Modulator System.
19. Select the **Library View** tab in the tree view.
21. Drag item `XMgraph.input=1` into the Model Editor Window.
22. Expand your library Sine Modulator Library.
23. Drag item `Sine Generator` into the Model Editor Window.
24. Drag item `Sine Modulator` into the Model Editor Window.
25. Arrange the components as shown in diagram (c)
26. Save the model using the tool button **Save Model** (Ignore construction error messages).

![Diagram](image)

**Figure 3.10: Example models after the creation of the model instances**

The component type can be determined by observing the colored triangle in the upper right corner of the instance. (See table 3.5).

### 3.7 Setting Parameters

The next task is setting the parameters of the created model instances. Remember that the parameters of model instances are actual parameters, i.e., they are defined by the formal parameters of the instantiated model component. Actual parameter properties cannot be changed. Only the value of these parameters can be set. However, if you do not set the value of an actual parameter, the default value defined by the formal parameter of the instantiated model component is used instead. For parameters of type `datastruct`, that specify a composite data type, you can select another composite data type provided that the selected composite data type of the actual parameter is derived from the composite data type of the formal parameter.
This section describes the different methods used to set the values of model instance parameters.

### 3.7.1 Changing Parameter Values

The parameter values of model instances are edited using the property editor. On selecting the model instance to be changed, MLDesigner activates the **Instance Properties** plane. In this plane, the property editor shows the actual parameters of the selected model instance. On clicking the value field of the relevant parameter, it becomes editable. To open a multiple line text editor, you can click on the icon to the right of the edit field. Figure 3.11 shows the parameters for the Ramp model instance of the Sine Generator module. Parameter properties that cannot be edited have grey text in italics.

![Figure 3.11: Model instance properties example](image)

### 3.7.2 Parameter Expressions

Parameter values set through MLDesigner can be arithmetic expressions. This is particularly useful for propagating values from a system model parameter to primitive parameters lower down in the hierarchy. An example of a valid parameter expression is:

\[
\frac{\pi}{2 \times \$\text{order}}
\]

where \$\text{order} is a formal parameter defined in the module or system model. The basic arithmetic operators are

- addition,
- subtraction,
- multiplication,
- division, and
- exponentiation.

These operators work on integers and floating-point numbers. Currently all intermediate expressions are converted to the type of the parameter being computed. Hence, it is necessary to be very careful when, for example, using floating-point values to compute an integer parameter. In an integer parameter specification, all intermediate expressions will be converted to integers.
3.7 Setting Parameters

3.7.3 Complex-valued parameters

When defining complex values, the basic syntax is

\[(\text{real}, \text{imag})\]

where \text{real} and \text{imag} evaluate to integers or floats.

3.7.4 Fixed-point parameters

Fixed-point parameters may be directly assigned a precision. To do this, the parameter is given in the syntax \((\text{value}, \text{precision})\), where \text{value} is an ordinary number and \text{precision} is given by either of two syntaxes:

**Syntax 1:** As a string like \(3.2\), or more generally \(m.n\), where \(m\) is the number of integer bits (to the left of the binary point) and \(n\) is the number of fractional bits (to the right of the binary point). Thus length is \(m+n\).

**Syntax 2:** A string like \(24/32\) which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one. Thus, for example, a fixed-point parameter might be defined as \((0.8, 2/4)\). This means a 4-bit word will be used with two fraction bits. Since the value 0.8 cannot be represented in this precision, the actual value of the parameter will be rounded to 0.75.
A fixed-point parameter can also be given a value without a precision. In this case, the default precision is used. This has a total word length of 24 bits with the number of integer bits set as required to store the value. For example, the number 1.0 creates a fixed-point object with precision 2.22, and a value of 0.5 would create one with precision 1.23.

The precision of internal computations in a primitive is typically given by a parameter of type precision. A precision parameter has a value specified using either of the two syntaxes above.

### 3.7.5 Linking Parameters

One possibility to set the value of an actual model instance parameter is to link it with a formal parameter of the model in which the model instance is embedded. Linking means that the values of the model parameter are used as parameter values of model instances, seen in fig. 3.13. In this way, you can pass parameter values from the system model level down to the primitive model level. Linking is realized using the parameter names, i.e., if an actual parameter value expression of a model instance contains the name of a formal parameter of the model that embeds the model instance.

![Parameter linking in the singen module](image)

**Figure 3.13:** Parameter linking in the singen module

Figure 3.13 shows an example of linked actual parameter expressions. This is a parameter linking in the singen module that can be found in the MLD Libraries/DEMO/SDF Demo/-Basic/sinMod system model. In this example the model defines the formal parameter sample_rate, frequency, and phase_in_radians. The values of these formal parameters are used to set the actual parameter values of the Ramp primitive instance. The example shows two kinds of linking:

1. the instance parameter value is determined by an expression that contains formal parameter names, see step, or
2. the instance parameter value is determined directly from the formal parameter, see value.

The former case is called indirect linking whereas the latter case is called direct linking. Direct linking is indicated by a green arrow in the parameter value field.

To create a direct link you can either export the actual model instance parameter as a new formal model parameter (see sec. 3.5.2), or you can link the actual model instance parameter to an existing formal model parameter. In the latter case, select the model instance of the actual parameter to link...
to and activate the **Instance Properties** plane. Click with the right mouse in the property editor on the parameter entry. From the context menu select **Link to**. On selecting this item, MLDesigner opens a sub-menu that contains items for all existing formal parameters of the model. Selecting one of these items creates a link.
You will then see a green arrow indicating that a link exists.

### sinMod Example. Step 5.

Continued from page 3-16
To demonstrate parameter setting and linking mechanisms, let us set the model instance parameter values of our example. Different methods for achieving the same result are demonstrated here such as linking and export. First, you must set the actual parameters of the Ramp instance in the Sine Generator module.

1. Select the Model Editor Window containing the Sine Generator module.
2. Click on the Ramp model instance.
3. Click on the value field of the parameter step.
4. Set the value to $2\times\pi \times \frac{\text{Frequency}}{\text{SampleRate}}$.
5. Save the model using the tool button **Save Model** (Ignore the construction errors).

You have now set the actual parameter step of the Ramp instance to an expression that contains the formal model parameter names. This realizes indirect linking between the Ramp parameter and the Sine Generator parameter.

---

**Figure 3.14: Parameter settings for the example**
The next step is to set the actual parameter of the Sine Generator instance in the Sine Modulator model. The Sine Generator has two actual parameters Frequency and SampleRate. Since the sample rate should be constant in the whole system model hierarchy it should be settable on the system model level. The Frequency of the sine modulator should be a parameter on the system model level. Therefore, it is necessary to export the SampleRate and Frequency parameters to the next higher model level. Follow the steps below to export these two parameters.

1. Select the Model Editor Window that contains the Sine Modulator module.
2. Click on the Sine Generator model instance in the Model Editor Window.
3. Click with right mouse button on parameter Frequency.
4. Select the context menu item Export.
5. Click with right mouse button on parameter SampleRate.
6. Select the context menu item Export.
7. Save the model using the tool button Save Model (Ignore the construction error messages).

NOTE: The green arrows in both fields indicate that these parameters are successfully exported.

By exporting the actual parameters Frequency and SampleRate of the Sine Generator instance, you have automatically created the two formal model parameters of the Sine Modulator module.

The last step is to create the system model parameters. You need two different frequencies; one for the signal frequency, and a carrier frequency for the sine modulator as well as a system model parameter for the sample rate.

1. Select the Model Editor Window that contains the Sine Modulator System model.
2. Click on the Sine Generator model instance.
3. Click with right mouse button on parameter Frequency.
4. Select the context menu item Export as.
5. Set the name of exported parameter to SignalFrequency and click OK.
6. Click with right mouse button on parameter SampleRate.
7. Select the context menu item Export.
3.7 Setting Parameters

8. Click on the Sine Modulator model instance.
9. Click with right mouse button on parameter Frequency.
10. Select the context menu item Export as.
11. Set the name of exported parameter to CarrierFrequency and click OK.
12. Click with right mouse button on parameter SampleRate.
13. Select the context menu item Link to.
14. Select the context sub-menu item SampleRate.

15. Click on the Sine Modulator System model background.
16. Click on the value field of the parameter SignalFrequency.
17. Set the value to PI/100 and click OK.
18. Click on the value field of the parameter CarrierFrequency.
19. Set the value to 0.2*PI.

20. Save the model using the tool button Save Model, ignore the construction error messages.

You have created the system parameters SignalFrequency, CarrierFrequency, and SampleRate by exporting the model instance parameter. However, the setting of the actual parameter SampleRate of the Sine Modulator model instance were realized by linking the parameter to the existing system model parameter SampleRate. Now you can control the whole model using system parameters. By setting the Scope attribute to External in the System Properties window, you can control them directly in the Simulation Properties window (see ch. 7).

3.7.6 Inserting Comments in Parameters

Comments are supported for non-string parameters. A comment is specified with the (#) symbol. Everything after the (#) until the end of the line is ignored when the parameter is evaluated. Comments are especially useful in combination with files, as they remind you about which module or primitive parameter the code refers to. Comments are, however, not supported for the string parameter or stringarray parameter types. In fact, when the image processing primitives use string states to represent a filename, the # character is used to denote the frame number of the image being processed.

3.7.7 Using Tcl Expressions in Parameters

Arbitrary PTCL expressions can be embedded in a parameter expression by preceding the expression with the tcl keyword as in the following example:

tcl(expression)

Firstly, parameters in the form of {parameter} appearing in the expression are replaced by their values. Then the string is sent to the MLDesigner built-in interpreter for evaluation. Finally, the result is spliced into the parameter expression and re-parsed. The interpreter is the same built-in PTCL interpreter that appears in the console window under the Command tab. This facility is generalized and supports both numeric and symbolic computing of expressions. Through PTCL, you can access all of its math functions, which generally behave the same as
### 3 Developing Models

#### Figure 3.16: Parameter settings for the example

**Table 3.1: Parameter settings for the example**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import Libraries</td>
<td>Sine_Modulator_Library</td>
</tr>
<tr>
<td>Description</td>
<td>This is the sine modulator system</td>
</tr>
<tr>
<td>Decimation</td>
<td></td>
</tr>
<tr>
<td>Global Seed</td>
<td>1</td>
</tr>
<tr>
<td>Run Length</td>
<td>100</td>
</tr>
<tr>
<td>P Tcl Script</td>
<td></td>
</tr>
<tr>
<td>Signal Frequency</td>
<td>PI/100</td>
</tr>
<tr>
<td>Control Frequency</td>
<td>0.2*PI</td>
</tr>
<tr>
<td>Sample Rate</td>
<td>2*PI</td>
</tr>
<tr>
<td>Input File</td>
<td></td>
</tr>
<tr>
<td>Loop Scheduler</td>
<td>DEF</td>
</tr>
<tr>
<td>Schedule Period</td>
<td>0.0</td>
</tr>
</tbody>
</table>

(a) Sine Modulator System

(b) Sine Generator instance

(c) Sine Modulator instance

MLDesigner Version 2.8
3.7 Setting Parameters

ANSI C functions of the same name: abs, acos, asin, atan, atan2, ceil, cos, cosh, double, exp, floor, fmod, hypot, int, log, log10, pow, round, sin, sinh, sqrt, tan, and tanh.

A parameter expression could be

\[
\text{tcl(expr sqrt(2.0 / \$BitDuration))}
\]

for the amplitude of the oscillators in a binary frequency shift keying system, in which \text{BitDuration} is a parameter. The \text{expr} command is a PTCL command that treats its arguments as a single mathematical expression that must evaluate to a number.

There are several PTCL commands embedded in MLDesigner that help support parameter calculations. They are: \text{listApplyExpression}, \text{max}, \text{min}, \text{range}, \text{rangeApplyExpression}, and \text{sign}. For example,

\[
\text{tcl(min [max 1 2 3] [sign {-2}])}
\]

first evaluates to \text{min} 3 \text{ -1} and then to \text{ -1}. The procedure \text{range} returns a consecutive sequence of numbers:

\[
\text{tcl(range 0 5)}
\]

returns 0 1 2 3 4 5. The \text{rangeApplyExpression} procedure generates a sequence of values by applying a consecutive sequence of numbers to a PTCL expression that is a function of \text{i}. For example, you can generate the taps of an FIR filter that is a sampled sinusoid by using

\[
\text{tcl(rangeApplyExpression \{ cos(2*{PI}*$i/5) \} 0 4)}
\]

This generates one period of sinusoidal function and returns

1.0 0.309042 -0.808986 -0.809064 0.308916

The \text{listApplyExpression} is similar to \text{rangeApplyExpression} except that it only takes two arguments: the second argument is a list of numbers to substitute for \text{i} in the expression. The command

\[
\text{tcl(listApplyExpression \{ cos(2*{PI}*$i/5) \} [range 0 4])}
\]

is equivalent to the previous example of the \text{rangeApplyExpression} function.

You can receive help on the new PTCL procedures \text{listApplyExpression}, \text{max}, \text{min}, \text{range}, \text{rangeApplyExpression}, and \text{sign}, by typing

\[
\text{help sign}
\]

at the prompt.
3.7.8 Using MATLAB and MATHEMATICA to Compute Parameters

Since PTCL can be used to compute parameters as described in the previous section, MLDesigner’s PTCL interface to MATLAB [HL96] and MATHEMATICA [Wol91] can be used to compute parameters. This allows even more expressiveness, but the drawback is that demonstrations relying on MATLAB and MATHEMATICA will only work at sites that have MATLAB and MATHEMATICA installed. For example, MATLAB can be used to design a 32-order FIR half-band filter using the Parks-McClellan optimal equiripple FIR filter design algorithm:

```matlab
matlab"getpairs c {c=remez(32, [0 0.4 0.6 1], [1 1 0 0])})"
```

Similarly, MATHEMATICA can be used to derive formulas to be used as parameters:

```mathematica
mathematica"get c {c=Integrate[A x, {x, 0, 1}]})"
```

This command returns the symbolic expression \( \frac{A}{2} \) which is re-parsed by MLDesigner. MATLAB and MATHEMATICA can be used to keep track of how parameter values are computed. MATHEMATICA can also be used to return symbolic expressions that can be used in conjunction with higher-order functions to define scalable systems [EGKL95].

3.7.9 Array parameters

When defining arrays of integers, floats, complex numbers, fixed-point numbers, or strings, the basic syntax is a simple list separated by spaces. For example,

```
1 2 3 4 5
```

defines an integer array with five elements. The elements can be expressions if they are surrounded by parentheses:

```
1 2 PI (2*PI)
```

Repetition can be indicated using the following syntax:

```
value[n]
```

where \( n \) evaluates to an integer. An array or portion of an array can be an input from a file using the symbol < as in the following example:

```
1 2 < filename 3 4
```

Here the first two elements of the array will be 1 and 2, the next element(s) will be read from file filename, and the last two elements will be 3 and 4. The latter capability can be used in combination with the WaveForm primitive to read a signal from a file.
3.7 Setting Parameters

3.7.10 String Parameters

There are complications when you wish to set a string parameter or stringarray parameter equal to the value of a module or system model parameter. This is because a distinction must be made between a sequence of characters that give the name of a symbol and a sequence of characters to be interpreted literally. The syntax to use is:

```
This string has the word {word} taken from another parameter
```

Here `{word}` represents the value of a string module or system model parameter. This capability is especially useful for constructing labels for output plots. When using string parameter to specify options for a UNIX command, as in the options parameter in `Xgraph` primitives. You need either double quotes or single quotes to include whitespace:

```
-0 'original signal' -1 'estimated signal'
```

String arrays have a few more special restrictions. Each word (separated by whitespace) is a separate entry in the array. To include whitespace in an element of the array, use quotation marks. Thus, the following string array

```
first "the second element" third
```

has three elements in it. The string array

```
repeat[10]
```

has ten separate copies of the string `repeat` in 10 separate entries in the array. Curly braces are used to substitute in values from module parameters. Thus, in

```
{paramname}
```

`paramname` must be the name of either a string array or a scalar-valued parameter (an integer, float, or complex array, for example, is not permitted). If it is a string array, then each element of `paramname` becomes an element of the parameter. If it is some other kind of parameter, the value becomes a single element of the string array.

To use one of `[`, `]`, `{`, or `}` literally, quote them with double quotes. To turn off the special meaning of a double quote, precede it with a backslash: `\"`. Similarly, use `\` to get a single backslash.

String array values may also be read from files using the `<` symbol.

```
Note that for string arrays, the filename can be a literal string such as
```

```
< \$MLD/data/filename
```

as well as a string that refers to parameters such as

```
< \$MLD/\{data_dir\}/data_file
```

in this case the value of the parameter `data_dir` would be substituted. MLDesigner does not perform expansion of filenames such as `file.{1,2}` into `file1 file2` as a UNIX shell might do.
3.8 Connecting Blocks

After the configuration of model instances, you need to connect them. To create connections, you must switch to connection creation mode either by clicking on the Add Connection tool button or by double-clicking on a port of a model instance. In the connection creation mode, every time you click on a port you start or finish a connection. Clicking anywhere on the model background after clicking an input or output port while the cursor is in connection mode, will result in a connection line being drawn to that point. Connections are also known as transitions.

Connection Nodes

A node is a point on a connection between model instances where the line turns a corner or where a connection makes a fork or a merge. New nodes can be created by double clicking a connection line and drawing a connection to a port or another connection line. It is also possible to create a Corner Node by clicking on the connection and dragging the line. This is sometimes necessary in order to neaten up your model or block diagram.

To switch back from the connection creation mode click the right mouse button or the tool button Select Tool. The name of the model instance and the name of the port are shown in the tool tip box if the mouse pointer is held over these elements.

sinMod Example. Step 6.

Continued from page 3-21

The next step is to connect the model instances of the example models. Start with the creation of connections in the Sine Modulator model.

1. Select the Model Editor Window that contains the Sine Modulator model.
2. Double-click on the Output port of the Sine Generator instance to start a new connection.
3. Click on input#2 port of the Mpy.input=2 instance to complete the connection.
4. Click on the output of the model port Input to start a new connection.
5. Click on input#2 port of the Mpy.input=2 instance to complete the connection.
6. Click on the output port of the Mpy.input=2 instance to start a new connection.
7. Click on the input port of the model port Output to complete the connection.
8. Save the model using the tool button Save Model.

Repeat these steps for the Sine Generator and Sine Modulator System examples. Your models should now look like those in fig. 3.17.

NOTE: Unused ports must be terminated. This is achieved by choosing Terminate from the context menu while the relevant port is selected. In these cases the appropriate BlackHole primitive for output ports and the Const or NULL primitive for input ports for the domain are instantiated.
3.9 Auto-Forking

In MLDesigner there are a wide variety of options when connecting ports, primitives, or modules. In some domains it is possible to make use of the autoforking and automerging facility built into MLDesigner (see table 3.6). In some instances, delays and buses are allowed but are not necessary. The following section explains where buses and delays are allowed and where autofork and automerge functions are supported.

3.9.1 Relations without formal ports

In cases where one single input port is connected to a single output port or where a multi-input is connected to a multi-output:

- delay is allowed
- bus is allowed if both ports are multiports
- ports must be compatible or must be of type Anytype

In cases where one single output is connected to many single/multi input ports:

- delay is allowed
- bus is not allowed
- port types must be compatible

In cases where many single/multi output ports are connected to one multi input port:

- delays are allowed

Figure 3.17: Example models after connecting the model instances
3 Developing Models

Figure 3.18: Single to Multi and Multi to Single Port

Figure 3.19: Single to Many Multi Port

- port types must be compatible
- buswidth must be equal to one

Figure 3.20: Many Single to Single/Multi Ports

In cases where many single/multi ports are connected to many ports

- delays are allowed
- busses are not allowed
- inmulti ports are not allowed. The transitions are broken as shown in fig. 3.21(a) if a multiple input port is used in conjunction with a fork.
- buswidth on multi output ports must be equal to one
- all ports must be compatible
- a new merge primitive is created if a built-in merge primitive exists in the domain and all outputs are connected with the merge’s input.
3.9 Auto-Forking

3.9.1.1 Relations with formal inputs

In cases where a single output is connected to a single input port:

- delays and buswidth have no effect
- ports must be compatible
- connections to outputs are only allowed if a Identity primitive exists in the actual domain. A block of the Identity primitive is created between the ports.

In cases where an input port is connected to many single/multi input ports:

- a new fork primitive is created if a built-in fork exists in the actual domain and the input is connected with the fork’s output. If the fork primitive does not exist for the domain, such connections are not allowed.

3.9.1.2 Relations with formal outputs

In cases where a single output port is connected to a single input port:

- delay and buswidth have no effect
• ports must be compatible
• connections to formal input ports are only allowed if a built-in identity star exists in the actual domain. A block of the identity primitive is created between the formal ports

In cases where many single/multi input ports are connected to a formal output:

• a new merge primitive of the actual domain is created (if one exists in the actual domain) and the formal output is connected with the merge’s outputs
• If no merge exists for the actual domain, such connections are not allowed
• Such connections may alter the behavior of the

![Diagram](image)

(a) Auto merge in DE domain  
(b) What really happens

Figure 3.23: Example models after connecting the model instances

### 3.10 Using Buses and Delays

There are additional property elements like buses and delays that are placed on top of connections. These property elements can be used to influence the behavior of the model.

#### 3.10.1 Buses

If you create a connection between multiple output and multiple input ports, it is often necessary to specify the number of connections the single wire represents. This can be realized by the bus property element. The bus property is much like a delay in that it is placed directly over the multiple connection. Its single parameter `busWidth` specifies the number of connections the single wire represents. Take a look at the following example. The `Map` primitive represents a number of model instances of the same model component, in this example the `Gain` primitive. Here, the bus width has to be three or the `Map` primitive will issue an error message. This is because there are three inputs to the `Map` primitive, so three instances of the `Gain` primitive will be created. The three outputs from these three instances need somewhere to go.

To create a bus property element, use the **Add Bus** tool button to switch to the bus property creation mode.

You can specify the bus width by the parameter `busWidth`. To do this select the bus icon in the Model Editor Window. MLDesigner activates the **Instance Properties** plane in the property editor window. This property can be used to modify the bus width.
MLDesigner provides a number of primitive components that realize capabilities to model the conversion from single connections to bus connections and vice versa. For that purpose, you need the **bus create** and **bus merge** primitives from the HOF domain available, for example, in library MLD Libraries/HOF Domain. The figure below shows an example for using a bus merge primitive.

![Bus Merge Example](image)

### 3.10.2 Delays

In several domains, delays can be placed on connections. A delay is not a primitive, but rather a property of the arc connecting two primitives. The interpretation of the delay in the data flow domains (SDF, DDF, BDF, and most code generation domains) is as an initial particle on the connection. An initial particle for the scalar data types is one whose value is zero. When the connection passes particles containing message type data, a delay on the connection will create an "empty" message. Most often, the destination primitive of the connection must be able to interpret such "empty" messages explicitly in context of the user-defined type because a "zero" might have different meanings depending on the type. Any feedback loop in the SDF domain must have a delay, or the computation in the loop would not be able to begin. You can specify the number of delays.

Other domains (besides data flow) also use delays, but the meaning can be quite different. See the chapters describing the domain.

A new feature added to MLDesigner releases is the support of initializable delays for simulation domains. These delays use a different icon from the old white diamond with green borders. The new delays use an icon that is a green diamond with a white border and has an "I" in the middle of the diamond to signify that it is initializable. MLDesigner has kept around the old delays for backward compatibility, but the syntax for the two is quite different and you should probably use just one type to prevent confusion.

The syntax for the new delays is that the arguments to the delay are the initial value themselves. There is no value in the argument that signifies the number of delay particles. Instead, a count of the number of values in the delay arguments is the number of delay particles that will be added to the buffer of the connection corresponding to the delay. These arguments are specified as a string and are parsed according to the data type associated to the connection. For example, an initializable delay with parameter `1 0 1` on an connection passing float particles will have a buffer with three initial particles. The three particles will have the values `1.0, 0.0,` and `1.0` respectively. If the connection was working on complex particles instead, an error would be given since complex numbers must be specified using a pair of numbers. A proper argument list for the delay in
that case would be \((1,0)\ (0,0)\ (1,1)\). The shorthand for declaring multiple values in the argument list is valid, just as in the arraystate case. For example, an argument list of \(2\ [5]\) would specify five initial particles with value 2.

Initializable delays also work on connections which handle matrix particles. The argument string in this case is parsed differently than above. The first two values in the last specify the number of rows and columns in the initial matrix, respectively. For example, an initializable delay with parameter \(1\ 2\ 3\ [2]\) on an connection passing integer matrices would place one matrix with dimension one row by two columns, whose entries all have the value three, in the buffer for that connection. For the case where multiple initial matrices are desired, simply give enough entries in the delay argument string to fill multiple numbers of initial matrices of the given size. For example, an initializable delay with parameter \(1\ 2\ 3\ 3\ 4\ 4\ 5\ 5\) on an connection passing integer matrices would create three matrices, all of dimension one row by two columns, such that the first initial matrix on the buffer has all entries equal to three, the second has all entries equal to four, and the last matrix has all entries equal to five.

To use delays in MLDesigner, you must place a delay or initializable icon on top of the wire connecting two instances. The delay icon is a white diamond with a green border. As mentioned above, the initializable delay use an icon that is a green diamond with a white border and has an 'I' in the middle of the diamond to signify that it is initializable. To create a delay, use the 

**Add Non-initializable Delay** tool button or the **Add Initializable Delay** to switch to the delay property creation mode. In this mode, every time you click in the model port, a delay property element, initializable or not, is created. You can specify the number of non-initializable delays and the expression of initializable delays. To do that, you must click on the top of the delay icon. MLDesigner activates the **Instance Properties** plane in the property editor window.

### 3.11 Using Labels for Annotation

It is often useful to annotate a block diagram with titles and comments. Use the tool button **Add Text Label** to switch to the label creation mode. In this mode, every click on the model background opens the **Add Text Label** dialog shown in fig. 3.24. The text of the label and a number of label properties can be defined using this dialog. To switch back from the label creation mode explicitly, use the second mouse button or the tool button **Select Tool**.
3.12 Color Settings

You can set the colors for the model background, the instances, and connections. Use the context menu item **Background Color** or **Color** of the selected model element. The menu option **Color** is only available for relations between models. The menu option **Background Color** applies to model instances and systems.

The **Select color** dialog allows you to define colors for model components (see fig. 3.25).

![Color Selection Dialog](image)

Figure 3.25: Color Selection Dialog

**sinMod Example. Step 7.**

Continued from page 3-28

Setting colors for model instances and system backgrounds is possible. The color setting dialog is available from the context menu. When a model instance is selected the context menu option is **Color...** and when no model elements are selected the menu option is **Background Color.**

![Sine wave modulator example system](image)

Figure 3.26: Model after colors are changed
3.13 Using Shared Elements

With MLDesigner you can use a number of so-called shared elements to model the functionality of a component. Such elements are called shared elements as they are used to share information without exchanging data. For that purpose they can be linked over different model hierarchy levels. Linking means that different model components use the same model element to manipulate the data. Shared elements are:

- memories
- events, and
- resources.

This section covers using shared elements. For more detailed information see ch. 10.

3.13.1 Creating Shared Elements

To create shared model elements, use the tool buttons Add Event, Add Memory, and Add Resources. The cursor changes mode and every time you click on the model background a new shared element with default properties and a unique name is created. To switch the cursor back to normal mode click the right mouse button. You can also use the relevant context menu to create shared model elements. This method only creates one model instance before the cursor reverts back to normal mode.

**NOTE:** Not all shared model elements are available in each modeling domain. Events and resources can be used only in the DE domain. In all other domains only memories are available.

3.13.2 Setting Shared Model Elements

There are a number of ways to share data between blocks of a hierarchical model. You can use the property editor to set the properties of shared model instances once they have been created. To do this select the shared element to activate the relevant Properties window. The properties a shared element has depend on its type (see ch. 10). The common properties of shared elements are the **Name** and the **Scope**.

The **Scope** of a shared element specifies whether it is visible in instances of the model component. If visible it can be linked to shared elements of the same type on the next higher model hierarchy level. A shared element is visible if you set the scope as **External**. In a system (the highest level of a hierarchical model) the scope is set to **Internal** and cannot be changed.

**NOTE:** A shared element with external scope is indicated by a small triangle at the top of the visible model element.

3.13.3 Exporting Shared Model Elements

It is now possible to export shared model elements from one module in a hierarchical model to the next highest level until the element is instantiated in the topmost level - the system. Till now it
was necessary to instantiate a shared element in each level of the hierarchical model ensuring the element had the same name and the same data type as the defining element in the lower level of the hierarchy. Exporting elements makes modeling quicker and reduces the possibility of errors such as typos. It is possible to choose to **Export** the shared element in which case the name remains unchanged unless an element exists with the same name in the next highest level of the model in which case the name of the element is extended by a `#instance_number`. Another possibility is to choose the **Export as** option in which case the name of the shared element must be defined manually.

The exported shared element has its scope set to **internal** by default. To share the element with the next highest level of the system set its scope to **external** and save the module. Instantiate the module in a system or module and click once on the model instance to activate the **Instance Properties** window. Activate the context menu over the relevant entry in the Instance Properties window and select **Export**. The shared element is then instantiated in the model in the Model Editor Window with its scope set to **internal**.

Events and Memories have a data type which can be any data type which is derived from the **Root** data type (see ch. 12.3.5). By default, Memories and Events are of data type **Root**. If you define an external event or external memory, you must link them, in instances of the model component, to according shared element which are of the same data type or a data type that is derived from the data type you defined for the event or the memory.

In instances of model components you must link shared elements with external scope. External shared elements are shown as properties in the property editor when you select the according model instance. The property entries are marked with the prefix **[E]** for event, **[M]** for memories, and **[R]** for resources. To link a shared element click with the right mouse button on the relevant property to open the context menu (see fig. 3.27). If there are shared elements with compatible data type in the model in which the model instance is embedded, the **Link to** context menu item is enabled. In that case, a list of shared elements to which the selected one can be linked is shown. You can select one from this list to create the link. Linking is indicated by a green arrow in the default value field of the linked shared element.

**NOTE:** If you change the type of the shared element of the model to which a shared element of an embedded model instance is linked, the link becomes invalid. In that case, MLDesigner will throw a construction error message if you try to save the model.

![Figure 3.27: Property context menu for shared element](image.png)
3.14 Dynamic Instances

The creation of dynamic model instances in the DE domain simplifies modeling of complex systems where flexibility is needed. Using dynamic model instances, DE module or primitive instances can be created and deleted at run time.

3.14.1 Create a Dynamic Instance

To create a dynamic model instance, activate the context menu over an instance in the Model Editor Window (DE domain). Select the menu option Dynamic Instance. The model instance now has auxiliary ports and parameters in addition to those of the conventional model instance. The additional ports and parameters are described here:

- Associated with each input port is an index port of type integer with the same name and the extension Idx. The index port specifies, on which instance of the model, the data of the associated input port will be placed. If no instance with this index exists, it will be created automatically.
- Associated with each output port is an index port of type integer with the name and the extension Idx. The index port specifies which instance releases the data on the associated output port.
- The dynamic instance has an auxiliary port named Create of type integer. This port can be used to create an instance with the index specified by the integer value placed on the port. If an instance with the given index already exists, the input will be ignored.
- The dynamic instance contains an auxiliary port named Delete of type integer. This port can be used to delete an instance with the index specified by the integer value placed on the port. If no instance with the given index exists, the input will be ignored.

**NOTE:** A deleted instance cannot be re-created with the same index, unless the parameter recreateDeletedInstances is set to TRUE.

- The dynamic instance contains an auxiliary port named Statistics of type Root.Statistics.DIStats. Each time the number of instances changes (instance created/deleted) this port outputs a data structure of type root.Statistics.DIStats, which contains a member containing an integer representing the current number of existing instances.
- The dynamic instance contains a parameter called recreateDeletedInstances that, if set to TRUE, allows the re-creation of an instance with an id that was already used and deleted in the current simulation run. By default, the value of this parameter is set to FALSE, and re-creation is not allowed.

**NOTE:** Models of the DE AddressMapping library as well as models of the DE GotoGroup library using the global address table are not working in connection with dynamic instances.

3.14.2 Example Tutorial

An example of a dynamic primitive instance can be found in the Library View under MLD Examples/Tutorials/DynamicInstance/XGraphWithDI. The "Graph" is instantiated as
3.15 Dynamic Linking

Dynamic linking of objects in external libraries with systems in MLDesigner is possible. This is a useful feature if you want to include functions from a variety of different libraries without rewriting the function.

You can only link objects such as files with extension .o (object files), .a (static libraries), and .so (shared objects). MLDesigner checks to see if the object files exist in the specified directory. With previous versions of MLDesigner it was only possible to link precompiled objects. Now if the specified files no longer exist MLDesigner will look for a makefile in the location where the object files were specified. If a makefile does not exist you must create it in the correct location. MLDesigner will then compile the object files thus ensuring that any changes made to the object are updated.

You can dynamically link on a primitive level where the function defined in the external library is defined as `extern` in `defparameter` of the source code before being called in the `go` method. You can also link on a system level where an external library is defined.
For this example we need to create a small system containing two Ramp primitives, a user defined Add primitive and a Dump primitive to write the simulation results to a file. This system can be found in the Tutorials library in MLD Examples and is called DynamicLinking.

1. Create a new system in the SDF domain called **dynamLinking**, in a library where you have write permission (see fig. 3.30).
2. Drag and drop the Ramp primitive from the MLD Libraries/SDF/Sources library into your system twice.
3. Save your system.

You must now create the Add primitive called **AddExt**:

- Click the **New Model** icon and create a primitive in the SDF domain called AddExt.
- Add two input ports and one output port using the appropriate icons on the toolbar.
- In the Model Properties editor change the Data Type of each port from anytype to int.
- Save the primitive.

You must now open the primitives source code and modify the code manually to look like the following:

```c
#include {}  

code  
{  
    extern int externalLibraryAdd( int, int );  
}  

constructor  
{  
}  
```

The primitive now tells the parser to look in an external library for the described function. The next step is to describe the behavior of the primitive during simulation, i.e. call the function of the external library.

```c
go  
{  
    int a = (int)(input1%0);  
    int b = (int)(input2%0);  
    int c = externalLibraryAdd( a, b );  
    output1%0 << c;  
}  
```

The next step is to show MLDesigner where the external file is physically saved.

- Click on the background of the primitive in the Model Editor Window. In the Model Properties window click the folder icon in the **Linked Objects** field to open the Select Multiple Files dialog window.
- In this dialog, click on the folder icon right to the **Select File** field.
3.15 Dynamic Linking

- In the **Open File** dialog window change to directory MLD Examples/Tutorials/DynamicLinking/LinkedObjects and select `libextadd.so`.

  **NOTE:** You can select multiple files using the `<Shift>` or `<Control>` keys combined with mouse clicks.

- Click the **Open** button. You will now see the selected file(s) in the **Select File** field of the **Select Multiple Files** dialog.
- Click the **Add** button to add the selected file(s) to the list of linked objects (see fig. 3.29).

  **NOTE:** If you leave the dialog using the **OK** button without clicking **Add** before, your selection will be lost.

  **NOTE:** You can delete files from the list or add more files if needed. Furthermore, you can determine the order in which the files are linked by ordering the files in this list. All files in the list can be located in different folders.

- Click **OK** to take over the selection.

  **NOTE:** You can select as many files as are needed for your system by using the `<SHIFT>` or `<CONTROL>` and mouse-click combination. Once you are happy with your selection click the **Add** button to add the complete file path to the Selected Files list (see fig. 3.29). You can delete files from the list or add more files if needed.

![Linked Objects Dialog](image-url)
Save the primitive and click and drag it into the *dynamLinking* system. You now need to add the Dump primitive to the system. Select the primitive from MLD Libraries/SDF/Sinks/-DmpNInt and drag it into the Model Editor Window. Click the DmpNInt once. In the Instance Properties window click the folder icon next to the [P]ods.OUT_DATA.FileName input field. Select a location where you would like the simulation results to be saved and type in a filename or select an existing file to overwrite. The Dump primitive also has a parameter *showGraph*. To print the results to a graph change the default setting to *Yes*.

### 3.15.1 Linked Objects and External Simulations

It is important in to observe the order in which linked objects are used by a system when the system is being simulated Extern. The objects should be listed in the correct order otherwise the simulation will not run. It is easy to change the order in which linked objects are listed by selecting the appropriate object in the *Linked Objects* dialog and using the Up/Down arrow shown in fig. 3.29 to move the object up or down in the list. The appropriate order should also be observed in the ∼/.mld/.mldrc file.

Connect the ports as shown in fig. 3.30.

![Figure 3.30: Example of Dynamic Linking](image)

### 3.15.2 Permanently Linking Objects to MLDesigner at Startup.

Permanently loading code at startup in order to extend the functionality of MLDesigner such as including a new domain or your own interface for interaction with other tools for co-simulation is possible. Locate the file ∼/.mld/.mldrc. You will see place holders for filenames with extension .o (object files), .a (static libraries), and .so (shared objects). You must enter the full path of the file you wish to dynamically link to MLDesigner in the appropriate place holder. Before these setting become active you need to shutdown and restart MLDesigner. On restarting MLDesigner, the file ∼/.mld/.mldrc is parsed and the dynamic linking command is activated.
3.16 Model Documentation

One crucial point of modeling is the documentation of developed model components. MLD designer provides a mechanism to automatically create and browse hypertext documentation. The hypertext documentation is used by the online documentation browser to provide information about model components.

3.16.1 Creating Documentation

Every time saving a model, MLD designer automatically creates a hypertext documentation. This hypertext documentation contains descriptions for

- model properties,
- ports,
- parameters, and
- model instances.

For each model instance, MLD designer creates a hyperlink documentation so that you can browse through the documentation of a whole model component hierarchy. Figure 3.31 shows the generated hypertext documentation for the Sine Modulator System example. Furthermore, for user libraries, usually stored in My Libraries ~/MLD you can generate an index of model component documentations.

Before starting the generation of a documentation index for a certain library, you must select the according model component item in the tree view, either in the library or the file view. After that, use the context menu item Generate Index & Documentation to start the generation. MLD designer starts with the generation of an index file in the library directory and creates a hypertext link to the hypertext documentation for each model component that belongs to the library. The index generation works recursively, that is, if there are sub-libraries in the library, MLD designer continues to generate the index files for the sub-libraries a.s.o If one of the modules in a library does not have hypertext documentation, MLD designer creates it automatically. Figure 3.32 shows the generated index for the Sine Modulator example library

3.16.2 Browsing Documentation

There are different ways to open the hypertext documentation for a model component. If the model is not opened in a design window, you can take the tree view. For that purpose, he has to select the according model component item either in the file or in the library view. After selecting the model component item, he can use the context menu item Online Documentation to open the hypertext documentation.

If the model is opened, the Online Documentation tool button is used to open the documentation of the model component to which the model belongs.

It is also possible to open the hypertext documentation for a model instance. If the context menu item Online Documentation is used for a selected model instance, MLD designer opens the hypertext documentation of the model to which the model instance refers.
Figure 3.31: Generated hypertext documentation

Figure 3.32: Generated hypertext documentation
NOTE: Only one hypertext documentation browser can be open at any given moment.

3.17 Source Code Editors

MLDesigner has a built-in editor which is extremely useful for debugging systems or modules. In certain situations you may have your environment variables set so that when you click the open source icon in MLDesigner an external editor is activated. The ability to choose an external editor for checking or changing source code may be useful to programmers who have invested a lot of time in customizing their favorite editor. There are, however, some disadvantages to using external editors. As described earlier in this document, MLDesigner allows you to open the source code for a primitive, once only. This avoids the type of confusion that could arise if the same primitives source code is open in more than one window of an external editor. With some editors the changes made in one buffer are not automatically applied to the other buffer. The changes made to one buffer could be overwritten by another buffer depending on which buffer is closed last. Another disadvantage is that not all editors have the compile option with an error console as is the case with MLDesigner’s built-in editor.

To change your environment variables so that source code is opened using the built-in editor, open a console and at the prompt where you would normally start MLDesigner, enter the following:

For **sh** and **bash** command shells:

```bash
export MLD_EDITOR=
```

For **csh** and **tcsh** command shells:

```bash
setenv MLD_EDITOR=
```

As an example for setting the environment variables to open the source code using an external editor we will describe how to set Emacs or XEmacs as default external editor for MLDesigner.

**NOTE:** The built-in editor is used as default editor in cases where compile errors occur. The reason is that you can highlight errors in the built-in editor error console and the cursor will be automatically placed in the correct line of the source code editor.

At the prompt where you would normally start MLDesigner, enter the following for **sh** and **Bash** command shells:

```bash
export MLD_EDITOR=xemacs
```

and for **csh** and **tcsh** command shells enter:

```bash
setenv MLD_EDITOR=xemacs
```

at the prompt. You can now start MLDesigner as usual. To reset the environment variables back to the built-in editor enter

```bash
export MLD_EDITOR=
```

or

```bash
unsetenv MLD_EDITOR=Xemacs
```
respectively.
<table>
<thead>
<tr>
<th>Type name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer</td>
<td>10</td>
</tr>
<tr>
<td>float</td>
<td>floating point number</td>
<td>0.2/PI</td>
</tr>
<tr>
<td>complex</td>
<td>pair specified as (real-part, imag-part)</td>
<td>(1.0, 2.0)</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>this is a string</td>
</tr>
<tr>
<td>fix</td>
<td>fixed point numbers</td>
<td>(2.546, 3.5)</td>
</tr>
<tr>
<td>intarray</td>
<td>array of integers</td>
<td>1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>floatarray</td>
<td>array of floating point numbers</td>
<td>0.0 [10] 1.0 0.0 [10]</td>
</tr>
<tr>
<td>complexarray</td>
<td>array of complex numbers</td>
<td>(0.1, 0.2) (0.3, 0.4) (0.5, 0.6)</td>
</tr>
<tr>
<td>stringarray</td>
<td>array of strings</td>
<td>this string array ”has five” elements</td>
</tr>
<tr>
<td>fixarray</td>
<td>array of fixed point numbers</td>
<td>41.78 2.546 [2] -3.5</td>
</tr>
<tr>
<td>precision</td>
<td>precision for fixed point numbers</td>
<td>3.5</td>
</tr>
<tr>
<td>boolean</td>
<td>enumeration with values FALSE and TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>enum</td>
<td>enumeration with user-defined values</td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>an expression of constant values, parameters, ports and memories</td>
<td>$Input+$Mem*$Param-3.2</td>
</tr>
<tr>
<td>file</td>
<td>filename (obsolete)</td>
<td>/tmp/input.test</td>
</tr>
<tr>
<td>filename</td>
<td>filename</td>
<td>$MLD_USER/output.txt</td>
</tr>
<tr>
<td>filenamelist</td>
<td>a list (array) of filenames</td>
<td></td>
</tr>
<tr>
<td>datastruct</td>
<td>data structure instance</td>
<td>Root</td>
</tr>
<tr>
<td>datastructname</td>
<td>data structure name</td>
<td>Root</td>
</tr>
<tr>
<td>datastructmembername</td>
<td>name of a composite data structure’s member</td>
<td>Byte1</td>
</tr>
<tr>
<td>modelname</td>
<td>an XML or OCT model</td>
<td>$MLD_USER/M/M.mml</td>
</tr>
<tr>
<td>integrator</td>
<td>an array describing continuous states</td>
<td>see floatarray</td>
</tr>
<tr>
<td>stateevent</td>
<td>continuous state variable used by ODE solver</td>
<td>see floatarray</td>
</tr>
</tbody>
</table>

Table 3.4: Parameter data types supported by MLDesigner
### Table 3.5: Colors defined for model instances

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Expander Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>red</td>
</tr>
<tr>
<td>Module</td>
<td>green</td>
</tr>
<tr>
<td>Primitive</td>
<td>blue</td>
</tr>
<tr>
<td>FSM Primitive</td>
<td>yellow</td>
</tr>
</tbody>
</table>

### Table 3.6: Domains containing Merge and Fork Primitives

<table>
<thead>
<tr>
<th>Domain</th>
<th>Identity</th>
<th>Merge</th>
<th>Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>Oneway</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SDF</td>
<td>Gain</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>BDF</td>
<td>Gain</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DDF</td>
<td>Gain</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FSM</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HOF</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.6: Domains containing Merge and Fork Primitives
Chapter 4

Debugging and Analyzing Systems

The combination of Probes, Breakpoints, Graphical Animation and Textual Animation are useful when you need to understand how a model works or want to check if the model is behaving as expected. With breakpoints, you can specify where and when you want a system to pause during a simulation. Probes are used to see what data is consumed or emitted by a specific port or shared model element.

The debug features Add Breakpoint and Add Probe discussed in the following sections are only available via the context menu when MLDesigner is in simulation mode.

4.1 Breakpoints

There are a number of types of breakpoints. The following types are implemented in MLDesigner:

- Instance Breakpoint,
- Port Breakpoint,
- Memory Breakpoint,
- Event Breakpoint,
- Resource Breakpoint,
- FSM Transition Breakpoint,
- Module Breakpoints,
- FSM State Breakpoints, and
- FSM History Breakpoints.

To place a breakpoint you need to switch to simulation mode. Click the **Switch to simulation Mode** icon on the toolbar and activate the context menu over the model instance where you would like to place the breakpoint. Select Add Breakpoint from the context menu.

4.1.1 The Breakpoints Console

A list of all breakpoints in the active system is displayed in the **Breakpoints** console when the system is switched to simulation mode. When you select an entry in the Breakpoints console the model instance flashes in the Model Editor Window. Each breakpoint has a unique ID and an
Object Name. The entries Enable and Ignore Count are editable entries. To change the settings or disable the breakpoint activate the context menu over the item in the Breakpoints console. The following options are available (see fig. 4.2):

- **Edit** - Opens the Edit Breakpoint dialog.
- **Enable/Disable** - If status is Yes the option to Disable the breakpoint is available. If status is No the option to Enable the breakpoint is available. The breakpoints can be toggled off and on using the Break Point icon on the simulation control toolbar.
- **Remove** - Removes the selected breakpoint.
- **Remove All** - Removes all breakpoints from the system in the active Model Editor Window.

### 4.1.2 Unconditional Breakpoints

It is possible to place unconditional breakpoints on ports, modules, primitives and shared model elements such as resources and memories when MLDesigner is in simulation mode. The simulation pauses as soon as a model instance containing a breakpoint fires or an event is produced at the specified point in the simulation. You can adjust the breakpoints to ignore a certain number of firings or events before pausing the simulation by setting the value in the Ignore Count input field.
The input field **Ignore Count** sets the amount of times a model instance or port fires before stopping the simulation. Once the instance has fired the specified number of times, the simulation is paused. Thereafter the simulation pauses every time the model instance fires.

### 4.1.3 Module Breakpoints

Sometimes a module is instantiated in a system more than once. Consider the sinMod system found in $MLD/MLD_Libraries/SDF/Demo/sinMod. This demo system contains two instances of the same module. The **SinusGenerator#1** module is instantiated in the **modulator#1** module and in the top level hierarchy of the system. Both instances of the module contain the same two primitives namely **Ramp#1** and **Sin#1**. When you place a breakpoint on either of these two primitives, the simulation will be stopped every time the module fires. You may want the simulation to stop when the primitive in question fires within a specific model instance of the module. This means you need to change the type of breakpoint.

To change the breakpoint type click on the appropriate breakpoint to activate the **Breakpoint Properties** window. Change the Module Breakpoint property from **Yes** to **No**. Click on the **Source Models** property. As shown in fig. 4.3 it is possible to select one or more model instances of the module from this list by mouse-click. If all are selected then you essentially have a **Module Breakpoint**.

![Figure 4.3: The Select Source Module dialog for Breakpoints](image)

**Example**

A simple system containing two instances of the same module is found in the File view under SDF/Demo/ and is called sinMod.

1. Open the system in the Model Editor Window and double-click the **modulator#1** model instance.
2. Notice the model instance **singen#1** is contained in the modulator module and in the sinMod system.
3. Double-click the **singen#1** model instance to open it in the Model Editor Window.
4. Click the **Switch to Simulation Mode** icon on the toolbar.
5. Click on the Ramp#1 output port and activate the context menu. Select **Add Breakpoint**

Two entries are visible in the **Add New Breakpoint** dialog. The entries show the system name, the module name, the model instance name, the primitive name and the selected Port name of the primitive.

6. Click the **Module Breakpoint** check box to select both instances of the Ramp primitive in the sinMod system and click **OK**.

7. Click the **Graphical Animation** icon on the toolbar.

8. Click the **Go** icon on the toolbar to start the simulation.

![Image](image.jpg)

**Figure 4.4: sinMod with singen#1 Model Instance**

The system pauses every time the Ramp primitive port fires and the module that contains the breakpoint is highlighted in the Model Editor Window. In simulation mode the tool tip text facility no longer displays a name and description for the model instance but rather displays values and parameters of the model instance or input/output port. When the system is paused at a breakpoint you can see what values are on which ports in order to analyze the system or suspected bug.

### 4.1.4 Breakpoints in Dynamic Instances

Breakpoints are, by default, set to stop the simulation every time a dynamically instantiated instance of a module fires. This behavior is determined by the **Breakpoint Properties** parameter **Module Breakpoint** which is set to **Yes**. This parameter must be set to **No** if you want the breakpoint to stop the simulation when a specific dynamically instantiated instance of the module fires. The **Source Model** property then becomes editable and must be defined. To do so click the icon of the **Source Model** property in the **Breakpoint Properties** window (see fig. 4.5). Activate the context menu in the **Select Source Models** dialog and choose the **Add supposed Breakpoint** menu option. All information regarding the module is automatically generated and entered in the dialog except for the instance number. The instance number is represented by the asterisk [*] as shown here:

```
SystemName.ModuleName#1.ModuleName#*.ModelInstance#1.port
```
You need to define on which instance you want the simulation to stop. Activate the context menu over the entry in the Select Source Models dialog and choose **Edit Breakpoint**. Replace the asterisk with an appropriate instance number. The instance number should be between zero and the maximum number of instances that will be instantiated during the simulation.

![Breakpoint Properties window and Select Source Models Dialog](image)

You can add more dynamic breakpoints by selecting **Add Supposed Breakpoint** from the context menu and defining on which dynamically instantiated instance of the model you want the simulation to stop.

**NOTE:** Only entries that are selected (highlighted) when the **OK** button is clicked are active breakpoints. You can select/deselect all breakpoints via the appropriate context menu option or by single mouse click.

### 4.1.5 FSM Breakpoints

There are a number of new FSM breakpoints that can be used to stop an FSM simulation when certain model items change their state or when an entry action occurs. Each breakpoint has options that can be set in the **Breakpoint Properties** window. By default all options are active. To deactivate certain breakpoint conditions select no from the drop down menu in the appropriate item in the Breakpoint Properties window.

#### FSM State Breakpoint properties

The following options are available for State Breakpoints:

- **Before Entry:** The breakpoint pauses the simulation before execution of the Entry Action of the associated State, even if the Entry Action is empty.
- **After Entry:** The breakpoint pauses the simulation after execution of the Entry Action of the associated State, even if the Entry Action is empty.
- **Before Exit:** The breakpoint pauses the simulation before execution of the Exit Action of the associated State, even if the Exit Action is empty.
- **After Exit:** The breakpoint pauses the simulation after execution of the Exit Action of the associated State, even if the Exit Action is empty.
FSM Transition Breakpoint properties

The following options are available for Transition Breakpoints:

- **Before Guard**: The breakpoint pauses the simulation before evaluation of the Guard Condition of the associated Transition, even if the Guard Condition is empty.
- **After Guard**: The breakpoint pauses the simulation after evaluation of the Guard Condition of the associated Transition, even if the Guard Condition is empty.
- **Before Action**: The breakpoint pauses the simulation before execution of the Action of the associated Transition, even if the Action is empty.
- **After Action**: The breakpoint pauses the simulation after execution of the Action of the associated Transition, even if the Action is empty.

FSM History Breakpoint properties

The following options are available for History Breakpoints:

- **Read Access**: The breakpoint pauses the simulation while reading a memorized State from the History.
- **Write Access**: The breakpoint pauses the simulation while writing a State to the History.

### 4.2 Probes

Probes are useful model elements when it comes to searching for bugs or simply understanding what a model instance or module does during simulation runs. Probes are recognized in the tree view by the probe icon shown here. A probe reads data from a model element and displays the information in one of the following ways:

- **Online Display** - A re-sizeable white display with a thin black border is placed close to the model element on which the probe is placed.
- **File** - The output data is written to a file in ASCII format. Whitespace separation followed by a hard return.
- **Tcl Variable** - The data is defined as a Tcl variable.

Probes are capable of reading information from input/output ports, Memories and Events. The information displays or is printed to a file when the value of the information on the appropriate model element changes.

**NOTE:** Probes can only be placed on or deleted from model elements before or after a simulation run while MLDesigner is in simulation mode

### 4.2.1 Probe Properties

The Probe Properties editor shown in fig. 4.6 is visible when a probe is selected in the Model Editor Window while MLDesigner is in simulation mode. The probe properties are listed and explained in table 4.1.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| Probe | Defines the type of Probe primitive to use. Default is the Dump primitive. Other types of probes are:  
  - FloatingMean  
  - Maximum  
  - MeanValue  
  - Minimum |
| SourceModel  | Instance from which to sample data. |
| Display  | Yes/No: Defines whether the probe is a display which is visible in the Model Editor Window during simulation. |
| File  | Yes/No: Defines whether the data should be saved to a file or not. |
| Filepath  | Opens a dialog where it is possible to select or create a file to write the data to. The data is printed to the file in ASCII Format. Data is separated by whitespaces and a newline for each new element. The data is overwritten with each simulation run or with each iteration. |
| Workspace  | Yes/No: Whether the data should be written to a Tcl variable or not. |
| Workspace variable name  | Name of a Tcl variable to print the data to. The variable is overwritten with each triggering. |
| Control Type  | Time/Trigger: Defines whether the probe must write data after a certain simulation interval or after a certain number of triggerings. |
| Start Time  | The time to start reading data from the model element if Control Type is set to Time. |
| Stop Time  | The time to stop reading data from the model element if the Control Type is set to Time. |
| Ignore Count  | The number of triggerings to ignore before reading data from the selected model element if the Control Type is set to Trigger. |
| Trigger Count  | The number of triggerings to read data from the selected model element if the Control Type is set to Trigger. |

Table 4.1: Probe Properties Editor


4 Debugging and Analyzing Systems

4.2.2 Probe Primitives

The probe primitives define the algorithm used to analyze data samples. The primitives have an input and an output port. Data samples are collected on the input port. Additional parameters can be defined for probe primitives. Additional ports, Memories, Events or Resources may \textbf{NOT} be defined.

Probes are not specific to a particular domain and can be used in all simulations.

![Probe Properties Editor](image)

Figure 4.6: Probe Properties Editor

4.2.3 Port Probes

The probe is triggered every time data is written to or read from a port. Numbered ports of special primitives are handled as normal ports.

Probes do not differentiate between multiports and single ports. All ports represented by the multiport deliver data to the probe. The probe cannot determine which port delivered which data.

4.2.4 Probes on Memories and Events

Probes on Memories and Events are triggered when data is written to the shared model element. The Dump primitive in the Probes library is designed to handle data of type “Anytype” and can be used for reading from Memories and Events. The data is placed on the input port of the probe.
4.2.5 Creating User Defined Probes

The procedure for creating user defined probes is exactly the same as creating a new primitive. Open the Create New Model dialog. From the Type of Model drop-down menu select Probe. Give the probe an appropriate name and save it in a library. A probe primitive with one input port and one output port is now open in the Model Editor Window.

Open the source code of the new primitive. The three methods init(), trigger() and finish() must be defined. These methods are briefly explained here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void init()</td>
<td>Is called when a probe is instantiated or when a simulation is started. There is no data on the port but the parameter is initialized.</td>
</tr>
<tr>
<td>trigger()</td>
<td>Typically reads data from the input, analyzes the data and puts the result on the output.</td>
</tr>
<tr>
<td>finish()</td>
<td>Is called when the simulation is complete or when the probe is deleted.</td>
</tr>
</tbody>
</table>

Table 4.2: Probe Methods

4.2.6 The DataNew Flag

The programmer must ensure that the DataNew flag for a port is set to true when a probe places data on an output port in the finish method. The following example demonstrates a Sum probe that collects all values and passes them on once they have all been received.

```c
output
{
    name { Output }
    type { float }
    num { 0 } <= don’t put out any data on trigger()
}

protected
{
    double sum;
}
init
{
    sum = 0.0;
}
trigger
{
```
sum += double(Input%0); // accumulate
}

finish
{
    Output.setPort(FLOAT, 1); // put one value of float at the end
    Output%0 = sum;
    Output.setDataNewFlag(true); // set the flag!
}

4.2.7 Probes on Dynamic Instances

The mechanism for placing probes within modules that are dynamic instances is the same as that used for placing breakpoints. As explained previously the dynamic instance does not exist when the probe is placed and must therefore be defined as a supposed probe on a model instance that will be dynamically created at some stage in the future. The difference is that only one probe can be placed on a given port whereas numerous breakpoints can be defined for a port. Breakpoints can also be placed on a model instance whereas the placement of probes is restricted to ports and shared model elements.

4.3 Argument Dependency Highlighting

When a shared model element is selected in the Model Properties window, all instances that use the element are highlighted in the Model Editor Window. This feature works for model elements such as Parameters, Resources, Events and Memories and works in both directions. When a model instance is selected in the design window all shared elements that are used by the model instance are highlighted in the Model Editor Window.

When a Parameter is selected in the Module Properties or System Properties window, all instances that use the Parameter are highlighted in the design window.

4.4 Compile with Debug Option

It is possible to compile all primitives in a library with debug symbols for execution using an external debugger. Only the primitives physically contained in the library are compiled. This option is available via the context menu over library entries in the tree view or via the context menu over the primitive in the Model Editor Window. A warning is printed to the command console to remind you that the primitive is compiled with debug symbols.

**NOTE:** Systems containing primitives which are compiled with debug symbols run slower in
MLDesigner. Once you have finished debugging it is recommended that you recompile the primitive optimized.

The debug symbols are removed from the object files when the system is recompiled Optimized. Once again you can choose to recompile all primitives in a given library or a single primitive via the respective context menu.

### 4.5 Debugging With External Debugger

Debugging using an external debugger may be useful to programmers. To start MLDesigner with an external debugger proceed as follows:

- Depending on the type of command line you are using, enter one of the following at the prompt:

  for **bash** or **shell**

  ```
  export MLD_PREBIN=ddd
  ```

  for **tcsh** and **csh**

  ```
  setenv MLD_PREBIN ddd
  ```

- The **ddd** entry refers to the ddd debugger supplied with every Linux package and can be replaced with a call referring to your favorite debugger.

Start MLDesigner as normal by typing `mld2` in the command line. The debugger you have chosen will start. If you are using **ddd** proceed as follows:

- Click the **View** menu and choose **Command Tool**.
- Click **Run** to start MLDesigner. This can take longer than normal.
- Once MLDesigner is started, you can open and run a system. It is possible to load the source files of those primitives compiled with debug information and to place breakpoints in the source code.

Starting with MLDesigner version 3.0 it is easier to debug the simulation process: the **Switch to Simulation Mode** toolbar button is a drop down button offering the possibility to start the simulation in the normal way or within a debugger.

If you want to debug an external simulation proceed as follows:

- Start MLDesigner as normal by typing `mld2` in a command window without setting the `MLD_PREBIN` variable before.
- Select a system to run, e.g. `$MLD_USER/MyLibrary/MySystem`.
- Click the **Switch to Simulation Mode** icon on the toolbar.
- Click the **Generate and Simulate Extern** toolbar icon.
- MLDesigner generates all the files needed to describe the system, compiles the generated files, and executes the simulation.
4 Debugging and Analyzing Systems

- Leave MLDesigner or switch to another command shell.
- Set the MLD_PREBIN variable as shown above.
- Change to the directory where MLDesigner created the files for the external simulation, e.g. 
  $MLD_USER/SYSTEMS/MyLibrary/MySystem.
- Start the external simulation by typing the name of the system in the command line, e.g.
  ./MySystem.
- A new ddd window is opened and the simulation is ready for execution.
- Activate the Command Tool and click the Run button. ddd now runs the simulation completely independent of MLDesigner.
Chapter 5

MLDesigner Kernel

The core of MLDesigner is a compact software infrastructure upon which specialized design environments, so-called domains, can be built. The software infrastructure, called the kernel, is made up of a family of C++ class definitions. For detailed information on all the kernel classes, please refer to Ptolemy 0.7 Kernel Manual [BH97]. The MLDesigner kernel was taken from the original Ptolemy system with some slight modifications. Therefore, the items MLDesigner kernel and Ptolemy kernel are used synonymously.

Domains are specific implementations of a model of computation. They are defined by creating new C++ classes derived from the base classes in the kernel. Domains can operate in either of two modes:

- **Simulation**: A scheduler invokes code segments in an order appropriate to the model of computation.
- **Code generation**: Code segments in an arbitrary language are stitched together to produce one or more programs that implement the specified function.

The use of an object-oriented software technology permits domains to interact with one another without knowledge of the features or semantics of the other domain. Thus, using a variety of domains, a team of designers can model each subsystem of a complex, heterogeneous system in a natural and efficient manner. These different subsystems can be nested to form a tree of subsystems. This hierarchical composition is the key in specifying, simulating, and synthesizing complex, heterogeneous systems.

In summary, the key idea in the Ptolemy kernel and thus also in the MLDesigner kernel was to mix models of computation, implementation languages, and design styles, rather than trying to develop one all-encompassing technique. The rationale is that specialized design techniques are

1. more useful to the system-level designer, and
2. more amenable to high-quality high-level synthesis of hardware and software.

The MLDesigner kernel demonstrates a way to mix tools that have fundamentally different semantics, and provides a laboratory for experimenting with such mixtures.
5.1 Models of Computation

The MLDesigner kernel does not define any model of computation. Every effort has been made to
keep data flow semantics out of the kernel. Thus, for example, a network of blocks could just as
easily represent a finite state machine, where each block represents a state. It is up to a particular
domain to define the semantics of a computational model.

The semantics of a domain is defined by classes that manage the execution of a specification. These
classes could invoke a simulator, could generate code, or could invoke a sophisticated compiler.
The principal classes responsible for control and execution of the system model are the target and the scheduler.

Targets

Targets take on particular importance in code generation domains where they describe all the
features of the target of execution, but they are used to control execution in simulation domains as well. Since class Target is derived from the most common kernel class Block, the target object itself has methods called setup, run, and wrapup. To define a simulation domain called XXX, for example, one would define at least one object derived from Target that runs the simulation. A Target can be quite sophisticated. It can, for example, partition a simulation for parallel execution, handing off the partitions to other targets compatible with the domain. See sec. 6.5 for a more in-depth look at how targets function.

Scheduler

A target will typically perform its function via scheduler objects derived from class Scheduler. The schedulers control the order of execution of blocks under their control. In some domains, the scheduler does almost everything. In such domains, the target simply starts it up. In others, the scheduler determines an execution order and the target takes care of many other details, such as generating code in accordance with the schedule, downloading the code to an embedded processor, and executing it. The Scheduler defines the operational semantics of a domain by controlling
the order of execution of functional modules. Sometimes, schedulers can be specialized. For instance, a subset of the data flow model of computation called Synchronous Data Flow (SDF) allows all scheduling to be done at compile time. The MLDesigner kernel supports such specialization by allowing nested domains. For example, the SDF domain, see fig. 5.3 is a subdomain of the Boolean Data Flow (BDF) domain. Thus, a scheduler in the BDF domain can handle all primitives in the SDF domain, but a scheduler in the SDF domain may not be able to handle primitives in the BDF domain. A domain may have more than one scheduler and more than one target.

5.2 Mixing Models of Computation

Large systems often mix hardware, software, and communication subsystems. The hardware sub-
systems may include pre-fabricated components such as: custom logic processors with varying
degrees of programmability, systolic arrays, and multiprocessor subsystems. Tools supporting
each of these components are different and possibly use different data flow principles such as:
5.3 Simulation Domains

regular iterative algorithms, communicating sequential processes, control/data flow hybrids, functional languages, finite-state machines, and discrete-event system theory and simulation.

In MLDesigner, domains can be mixed and even nested. Thus, a system-level description can contain multiple subsystems that are designed or specified using different styles. The kernel support for this is shown in fig. 5.2. An object of class XXXWormhole in the XXX domain is derived from class XXXStar, so that from the outside it looks just like a primitive in the XXX domain. Thus, the schedulers and targets of the XXX domain can handle it just as they would any other primitive block. However, inside, hidden from the XXX domain, is another complete subsystem defined in another domain, say YYYY. That domain gets invoked through the setup, run, and wrapup methods of XXXWormhole. Thus, in a broad sense, the wormhole is polymorphic. The wormhole mechanism allows domains to be nested on many levels, e.g., one could have a DE domain within an SDF domain within a BDF domain. The FSM domain is designed to always be used in combination with other domains.

5.3 Simulation Domains

Data Flow Models

One of the most mature domains included in the current system is the synchronous data flow (SDF) domain [LM87a, LM87b]. This domain is used for signal processing and communications algorithm development, and has particularly good support for multirate algorithms [BHLM91].

A dynamic data flow (DDF) domain extends the SDF domain by allowing data-dependent flow of control, as in Blosim. Boolean data flow (BDF) [BL93c, BL93b, Buc93] has a compile-time
Figure 5.2: Hierarchical Model Structure

Figure 5.3: MLDesigner domains
5.4 Code Generation Domains

Several code-generation domains use data flow semantics [PHLB93, Mur93]. These domains are capable of synthesis of C code, assembly code for certain programmable DSPs [Won92], VHDL, and Silage [KL93]. A significant part of the research that led to the development of these domains has been concerned with synthesizing code that is efficient enough for embedded systems [BL94a, BBHL93, BL93a, BL94b, BL93b, Buc93]. A large amount of effort has also been put into the automatic parallelization of the code [HL91, Ha92, SL93a, SL93b], and on parallel architectures that take advantage of it [Lee91a, SL93c].

Discrete-Event Models

A number of simulation domains with discrete-event (DE) semantics has been developed for MLDesigner, but the DE domain is the only pure discrete event domain released with MLDesigner. The DE domain is a generic discrete-event modeling environment useful for simulating queuing systems, communication networks, and hardware systems.

Discrete Event / Continuous Time Models

Synchronous Reactive Modeling

The software analogy of synchronous digital circuits has been realized by Stephen Edwards in the Synchronous Reactive (SR) domain [Edw92]. This model of computation is better suited than data flow to control-intensive applications and is more efficient than DE.

Finite State Machines

Another approach to designing control-intensive applications is to mix Finite State Machines (FSM) domain with data flow, DE, or other domains. Through FSMs you have the ability to mix timed and untimed domains into hierarchical systems.

5.4 Code Generation Domains

Domains in fig. 5.3 are divided into two classes: simulation and code generation. In simulation domains, a scheduler invokes the run methods of the blocks in a system specification, and those methods perform a function associated with the model. In code generation domains, the scheduler also invokes the run methods of the blocks, but these run methods synthesize code in some language. That is, they generate code to perform some function, rather than performing the function directly. The target then is responsible for generating the connecting code between blocks (if any is needed). This mechanism is very simple, and language independent. Ptolemy has released code generators for C, Motorola 56000 assembly, and VHDL languages, and these are now included in the MLDesigner kernel (see fig. 5.3).

An alternative mechanism that is supported but less exploited in current MLDesigner domains is for the target to analyze the network of blocks in a system specification and generate a single monolithic implementation. This is what we call compilation. In this case, the primitive blocks
must have functionality that is recognized by the target. In previous code generation mechanisms, the functionality of the blocks was arbitrary and could be defined by the end user.

Figure 5.4: Hierarchical system model structure.
6.1 Foreword to the domain concept

In MLDesigner, a System, also known as a Hierarchical Model, is made up of primitives, modules and model instances with ports connected together via transitions. Each subsystem or model instance may be modeled in a different domain to that of the parent System or model instance. In mixing domains, the key is to ensure that at the interface, the child module obeys the semantics of the parent domain. This interface method is called a wormhole.

Domains perform either:

- **Simulation.** Interpreters run an executable specification of a system on a local workstation.
- **Code generation.** Code generation domains translate the specification into some language such as C or C++ and then optionally manage the execution of that generated code.

The model of computation represents the semantics of the network of blocks. It defines what is meant by an interconnection of blocks, and how the interconnection will behave when executed.

**Timed and Untimed Domains**

Simulation domains can be either timed or untimed. Untimed domains carry no notion of time in their semantic model. Instead of chronology, they deal only with the order of particles or actions. Timed domains have a notion of simulated time, where each particle or action is modeled as occurring at some particular point in this simulated time. Particles and actions are processed chronologically. Timed and untimed domains can be mixed. From the perspective of a timed domain, actions in an untimed domain will appear to be instantaneous. Timed domains can exist at several levels in the hierarchy, or in parallel at a given level of the hierarchy, separated by untimed domains, and their chronologies will be synchronized.

That is, the notion of simulated time in MLDesigner is a global notion. When particles and actions are processed chronologically in each timed domain present, they will be processed chronologically globally.
6.2 Supported domains

6.2.1 Synchronous Data Flow (SDF)

The SDF domain is the most mature and widely used domain in MLDesigner. SDF is a special case of the data flow model of computation developed by Dennis [Den75]. The specialization of the model of computation is to those data flow graphs where the flow of control is completely predictable at compile time. It is a good match for synchronous signal processing systems, those with sample rates that are rational multiples of one another.

The SDF domain is suitable for fixed and adaptive digital filtering, in the time or frequency domains. It naturally supports multirate applications, and its rich primitives library includes polyphase real and complex FIR filters. Applications with examples in the demo library include speech coding, sample-rate conversion, analysis-synthesis filter banks, modems, phase-locked loops, channel simulation, linear prediction, chaos, filter design, Kalman filtering, phased array beamforming, spectral estimation, sound synthesis, image processing, and video coding.

6.2.2 Higher-Order Functions (HOF)

A function is higher-order if it takes a function as an argument and/or returns a function. A classic example is mapcar in Lisp, which takes two arguments, a function and a list. Its behavior is to apply the function to each element of the list and to return a list of the results.

The HOF domain implements a similar function, in the form of a primitive called Map, that can apply any other primitive (or module) to the sequence(s) at its inputs. Many other useful higher-order functions are also provided by this domain.

The HOF domain provides a collection of primitives designed to be usable in all other MLDesigner domains. It is intended to be included as a sub-domain by all other domains.

6.2.3 Dynamic Data Flow (DDF)

The predictable control flow of SDF allows efficient scheduling, but limits the range of applications. In particular, data-dependent flow of control is only allowed within the confines of a primitive. To support broader applications, the DDF domain uses dynamic (run-time) scheduling. For long runs, involving many iterations, this is more extensive than the static scheduling that is possible with SDF. But in exchange for this additional cost, the result is a model of computation that is as versatile as that of conventional programming languages. It supports conditionals, data-dependent iteration, and true recursion.

Although the DDF domain is, in principle, a fully general programming environment, it is nonetheless better suited to some applications than others. We have found that signal processing applications with a limited amount of run-time control are a good match. Examples include systems with multiple modes of operation, such as modems (which have start-up sequences and often implement multiple standards), signal coding algorithms (which often offer a range of compression schemes), and asynchronous signal processing applications, such as timing recovery and arbitrary
6.2 Supported domains

SampleRate conversion. The demos provided with the domain show how to realize conditionals, iteration, and recursion.

The SDF domain is in fact a sub-domain of DDF, which means that SDF primitives can be used in DDF systems. For greater efficiency on long runs, the two domains can also be mixed using the MLDesigner hierarchy. A module within a DDF system can be SDF, meaning that it will use an SDF scheduler. Conversely, a module within an SDF system can be DDF.

6.2.4 Boolean Data Flow (BDF)

Boolean data flow was developed by Joe Buck [Buc93]. Like DDF, it supports run-time flow of control. Unlike DDF, it attempts to construct a compile-time schedule. Thus it achieves the efficiency of SDF with the generality of DDF. It currently supports a somewhat more limited range of primitives than DDF, and does not support recursion, but the model of computation is, in principle, equally general. Its applications are the same as those of DDF.

The basic mechanism used in BDF is to construct an annotated schedule, by which is meant a static schedule where each firing in the schedule is annotated with the Boolean conditions under which it occurs. Thus, any sequence of firings can depend on a sequence of Boolean values computed during the execution. Executing the annotated schedule involves much less overhead than executing a dynamic data flow schedule.

6.2.5 Discrete Event (DE)

The DE domain is a relatively mature domain using an event-driven model of computation. In this domain, particles carry time stamps, and represent events that occur at arbitrary points in simulated time. Events are processed in chronological order. Two schedulers are available. The default scheduler is based on the "calendar queue" mechanism developed by Randy Brown and was written by Anindo Banerjea and Ed Knightly. Since this scheduler is relatively new, an older and simpler but less efficient scheduler is also provided.

DE schedulers maintain an event queue, which is a list of events sorted chronologically by time stamp. The scheduler selects the next event on the list, and determines which primitive should be fired to process the event. The difference between the efficient calendar queue scheduler and the naive simple scheduler is in the efficiency with which this queue is updated and accessed. Considerable effort was put into consistent and predictable handling of simultaneous events.

The DE domain is suitable for high-level modeling of communications networks, queuing systems, hardware systems, and transportation networks. The demos included with the domain include a variety of queuing systems, shared resource management, communication network protocols, packet-switched networks, wireless networks, and multimedia systems. The latter class of applications take advantage of the ability that MLDesigner has to mix domains by modeling speech and video encoding algorithms using the SDF domain and a packet switched network using the DE domain. There are also some more specialized uses of the DE domain, such as modeling shot noise and synchronizing a simulation to a real-time clock.
6.2.6  FSM Domain

MLDesigner 2.3 introduces a new Finite State Machine domain which is a significant upgrade over its predecessor. The new FSM supports an extended set of finite state machine elements, including

- events
- states
- transitions
- actions
- arguments
- histories

The FSM supports multiple levels of hierarchy. Most MLDesigner elements can be linked to states, transitions and actions.

When developing FSM models, the Model Editor is automatically configured as an FSM Editor through the addition of tool buttons for FSM model actions such as adding ports, arguments, states, transitions, histories, and labels.

The FSM domain is a sub-domain and can never be the domain of an MLDesigner system. Instead, FSM Models are always incorporated into other domains via the MLDesigner wormhole mechanism. In addition, the new FSM domain provides:

- reusability of the old MLDesigner FSM model
- support of the UML Statechart semantic
- support of all MLDesigner data types and data structures
- support of FSM relevant shared elements
- interaction with concurrency MLDesigner domains
- storage of FSM blocks via XML
- mapping of the BONeS FSM model

6.2.7  CTDE Domain

The purpose of this domain is to design and simulate continuous-time and mixed signal systems (mixed-signal simulation). There are many instances where pure discrete event or continuous time models of computation are not sufficient and are in fact problematic. By combining the two models of computation into a new domain it is now possible to design heterogenous systems with completely different signal types. Connections between ports of different types are made possible by inserting a primitive to convert the signal. These Primitives can be found in the EventInterpreters library (to convert a discrete into a continuous signal) and in the EventGenerators library (to convert a continuous to a discrete signal). This greatly enhances the designers ability to create models which were till now problematic.

The CT part of this mixed domain is suited for modeling physical systems with linear or nonlinear algebraic equation descriptions and has been extended to handle discrete events. This is achieved by the scheduler switching between continuous time and discrete event mode. At a given time in
the simulation interval the Discrete-Event scheduler processes all events that have a current time-
stamp. The simulator then switches to the Continuous-Time scheduler. The continuous trajectory
of the signal is calculated by a numerical ordinary differential equations (ODE) solver until the
next scheduled discrete event is reached where the scheduler switches back to discrete event mode.

Direct connection between ports of different types are not allowed. One has to insert a primitive
to convert the signal. These Primitives can be found in the:

- EventInterpreters library; to convert a discrete into a continuous signal and in the
- EventGenerators library; to perform the opposite conversion.

Continuous signals are vectors of real numbers. Many primitives can operate on vectorial inputs
such as the add primitive, as long as all inputs have the same width. The vector widths must be
consistent and are checked before the simulation starts.

### 6.3 Unsupported domains

#### 6.3.1 Synchronous Reactive (SR)

The Synchronous Reactive domain, created by Stephen Edwards [Edw92], is a new and very
experimental domain. The Synchronous Reactive domain is a statically-scheduled simulation
domain in MLDesigner designed for concurrent, control-dominated systems. To allow precise
control over timing, it adopts the synchronous model of time, which is logically equivalent to as-
suming that computation is instantaneous.

SR is similar to existing MLDesigner domains, but differs from them in several important ways.
Like Synchronous Data Flow (SDF), it is statically scheduled and deterministic, but it does not
have buffered communication or multi-rate behavior. SR is better for control-dominated systems
that need control over when things happen relative to each other; SDF is better for data-dominated
systems, especially those with multi-rate behavior.

SR also resembles the Discrete Event (DE) domain. Like DE, its communication channels transmit
events, but unlike DE, it is deterministic, statically scheduled, and allows zero-delay feedback
loops. DE is better for modeling the behavior of systems (i.e., to better understand their behavior),
whereas SR is better for specifying a system’s behavior (i.e., as a way to actually build it).

#### 6.3.2 Multidimensional Synchronous Data Flow (MDSDF)

The MDSDF domain was developed by Mike Chen and is still very experimental. This domain is
an extension of the Synchronous Data Flow model to multidimensional streams and is based on the
work of Edward Lee. MDSDF provides the ability to express a greater variety of data flow sched-
ules in a graphically compact way. It also allows nested reset-table loops and delays. Additionally,
MDSDF has the potential for revealing data parallelism in algorithms. The current implementa-
tion of the MDSDF domain only allows two dimensional streams, although we hope that many of
the ideas used in the development of the domain can be generalized to higher dimensions.
6.3.3 Code generation (CG)

The CG domain is the base from which all other code generation domains (such as CGC and CG56) are derived. This domain supports a general data flow model equivalent to the BDF and SDF models. The primitives in this domain do little more than generate comments when fired, but they can serve to demonstrate and test the features of scheduling algorithms. In this domain, you can build test systems, view the generated code (comments) for multiple processors, and display a Gantt chart for parallel schedules. In derived domains, real code is generated, compiled, downloaded and executed, all under control of the selected target. In MLDesigner, one serious weakness of the code generation domains is that they only support scalar data types (complex, floating-point, integer, and fixed-point) on the input and output ports.

6.3.4 Code generation in C (CGC)

The CGC domain uses Boolean-controlled data flow semantics, and has C as its target language. We have made every effort to name primitives and their parameters consistently so that it is easy to move from one domain to another. With a little effort, one could create CGC versions of all SDF primitives. If this were accomplished, then re-targeting from one domain to another would be a simple matter of changing domains and targets and running the system again.

The generated C code is statically scheduled, and the memory used to buffer data between primitives is statically allocated. Moreover, for many of the primitives, the code that is generated depends on the values of the parameters. One way to think of this is that the parameters of the primitive are evaluated at code generation time, so no run-time overhead is incurred from the added flexibility of parameterizing the primitive.

There are several targets to choose from in the CGC domain. The bdf-CGC target supports the boolean-controlled data flow model of computation. It must be used whenever primitives with BDF semantics are present in a program graph. The default-CGC target supports the SDF model of computation, so it can be used when the program graph contains only primitives with SDF semantics. The TclTk_Target target also supports SDF, and must be used whenever Tcl/Tk primitives are present in the program graph. The unixMulti_C target supports SDF and partitions the program graph for execution on multiple workstations on a network.

6.3.5 Code generation for the Motorola DSP56000 (CG56)

This domain synthesizes assembly code for the Motorola DSP56000 family. The code generation techniques that are used are described in [PHLB93]. They are derived from techniques used in Gabriel [BGH+90]. This domain has been used to generate real-time implementations of various modem standards, touch tone generators, and touch tone decoders.

6.3.6 Code generation in VHDL (VHDL, VHDLB)

This pair of domains is for generating code in VHDL (VHSIC Hardware Description Language). The VHDL domain supports functional models using the SDF model of computation, while
VHDLB supports behavioral models using the native VHDL discrete event model of computation. Since the VHDL domain is based on the SDF model, it is independent of any notion of time. The VHDLB domain supports time delays and time-dependent behavior of blocks. The VHDL domain is intended for modeling systems at the functional block level, as in DSP functions for filtering and transforms, or in digital logic functions, independent of implementation issues. The VHDLB domain is intended for modeling the behavior of components and their interactions in system designs at all levels of abstraction.

Within the VHDL domain there are a number of different Targets to choose from. The default target, default-VHDL, generates sequential VHDL code in a single process within a single entity, following the execution order from the SDF scheduler. This code is suitable for efficient simulation, since it does not generate events on signals. The SimVSS-VHDL target is derived from default-VHDL and it provides facilities for simulation using the SYNOPSYS VSS VHDL simulator. Communication actors and facilities in the SimVSS-VHDL target support code synthesis and co-simulation of heterogeneous CG systems under the CompileCGSubsystems target developed by José Luis Pino. There is also a SimMT-VHDL target for use with the Model Technology VHDL simulator. The struct-VHDL target generates VHDL code where individual actor firings are encapsulated in separate entities connected by VHDL signals. This target generates code which is intended for circuit synthesis. The Synth-VHDL target, derived from struct-VHDL, provides facilities for synthesizing circuit representations from the structural code using the SYNOPSYS Design Analyzer tool set. Because the VHDL domain uses SDF semantics, it supports re-targeting from other domains with SDF semantics (SDF, CGC, etc.) provided that the primitives in the original graph are available in the VHDL domain. As this experimental domain evolves, more options for VHDL code generation from data flow graphs will be provided. These options will include varying degrees of user control and automation depending on the target and the optimization goals of the code generation, particularly in VHDL circuit synthesis.

Unlike the VHDL domain, the older and less-developed VHDLB domain is much simpler in its operation. When a system in the VHDLB domain is run, the graph is traversed and a code file is generated in a pop-up window and in a subdirectory which reflects the topology and hierarchy of the graph. The generated VHDL code will reference VHDL entities which are expected to be included in other files. There is a VHDL code file in the $MLD/src/domains/vhdlb/lib directory for each VHDL primitive in the main primitives of the $MLD/src/domains/vhdlb/icons directory. Adding a new primitive is a matter of writing VHDL code for the entity and adding a primitive file in the primitives subdirectory of the VHDLB domain which reflects the inputs, outputs, and parameters of that primitive. The existing primitives should serve as examples for how new primitives can be written.

### 6.4 Summary of various domains

The following table shows a quick overview of all the domains currently available in MLDesigner. A '*' denotes domains which are experimental and therefore not supported by the MLDesigner support group.
<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous Data Flow (SDF)</td>
<td>Oldest and most mature domain; this is a sub-domain of DDF and BDF Domains  &lt;li&gt;Special case of data flow model of computation developed by Dennis [Den75]&lt;/li&gt;  &lt;li&gt;Flow is completely predictable at compile time thus allows for efficient scheduling&lt;/li&gt;  &lt;li&gt;Allows for static scheduling&lt;/li&gt;  &lt;li&gt;Good match for synchronous signal processing systems with sample rates that are rational multiples of one another&lt;/li&gt;  &lt;li&gt;Supports multi-rate applications and has a rich primitives library&lt;/li&gt;  &lt;li&gt;Range of applications is limited&lt;/li&gt;</td>
</tr>
<tr>
<td>Dynamic Data Flow (DDF)</td>
<td>Versatile model of computation as it supports conditionals, data-dependent iteration, and true recursion  &lt;li&gt;More general than SDF&lt;/li&gt;  &lt;li&gt;Uses dynamic (run-time) scheduling which is more time-intensive than static scheduling&lt;/li&gt;  &lt;li&gt;Good match for signal processing applications with a limited amount of run-time control&lt;/li&gt;</td>
</tr>
<tr>
<td>Boolean Data Flow (BDF)</td>
<td>Relatively new domain which supports run-time flow of control  &lt;li&gt;Attempts to construct a compile-time schedule to try and achieve efficiency of SDF with generality of DDF&lt;/li&gt;  &lt;li&gt;More limited than DDF&lt;/li&gt;  &lt;li&gt;Constructs an annotated schedule: execution of a task is annotated with a boolean condition&lt;/li&gt;</td>
</tr>
<tr>
<td>Discrete Event (DE)</td>
<td>Relatively mature domain which uses an event-driven model of computation  &lt;li&gt;Particles carry time-stamps which represent events that occur at arbitrary points in simulated time&lt;/li&gt;  &lt;li&gt;Events are processed in chronological order&lt;/li&gt;</td>
</tr>
</tbody>
</table>
### 6.4 Summary of various domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Finite State Machine (FSM)</strong></td>
<td>The new FSM supports an extended set of finite state machine elements, including</td>
</tr>
<tr>
<td></td>
<td>• events</td>
</tr>
<tr>
<td></td>
<td>• states</td>
</tr>
<tr>
<td></td>
<td>• transitions</td>
</tr>
<tr>
<td></td>
<td>• actions</td>
</tr>
<tr>
<td></td>
<td>• arguments</td>
</tr>
<tr>
<td></td>
<td>• histories</td>
</tr>
<tr>
<td><strong>Continuous Time/Discrete Event (CTDE)</strong></td>
<td>Combined Continuous Time and Discrete Event Model of computation</td>
</tr>
<tr>
<td></td>
<td>• The Discrete-Event scheduler processes all events that have a current time-stamp. The simulator then switches to the Continuous-Time scheduler.</td>
</tr>
<tr>
<td></td>
<td>• The trajectory of the continuous signal is calculated by a numerical ordinary differential equations (ODE) solver.</td>
</tr>
<tr>
<td></td>
<td>• Direct connections between ports of different types are not allowed. One has to insert a primitive to convert the signal.</td>
</tr>
<tr>
<td><strong>Higher Order Functions (HOF)</strong></td>
<td>Implements behavior of functions that may take a function and return a function</td>
</tr>
<tr>
<td></td>
<td>• HOF collection of primitives may be used in all other domains</td>
</tr>
<tr>
<td></td>
<td>• Intended to be included only as a sub-domain by other domains</td>
</tr>
<tr>
<td><strong>Integer and State Controlled Data Flow (STDF)</strong></td>
<td>Very new to MLDesigner and still experimental. This is an extension to BDF</td>
</tr>
<tr>
<td></td>
<td>• Realizes data flow control by integer control data and port statuses</td>
</tr>
<tr>
<td></td>
<td>• Scheduling is static and conditional like BDF</td>
</tr>
<tr>
<td></td>
<td>• It has user-defined evaluation functions</td>
</tr>
<tr>
<td><strong>Multidimensional Synchronous Data Flow (MDSDF)</strong></td>
<td>Relatively new and experimental</td>
</tr>
<tr>
<td></td>
<td>• Extends SDF to multidimensional streams</td>
</tr>
<tr>
<td></td>
<td>• Provides ability to express a greater variety of data flow schedules in a graphically compact way</td>
</tr>
<tr>
<td></td>
<td>• Currently only implements a two-dimensional stream</td>
</tr>
</tbody>
</table>
## Domain and Description

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous/Reactive (SR)*</td>
<td>Very new to MLDesigner and still experimental</td>
</tr>
<tr>
<td></td>
<td>• Implements model of computation based on model of time used in Esterel</td>
</tr>
<tr>
<td></td>
<td>• Good match for specifying discrete reactive controllers</td>
</tr>
<tr>
<td>Code Generation (CG)*</td>
<td>Base domain from which all code generation domains are derived</td>
</tr>
<tr>
<td></td>
<td>• Supports a data flow model that is equivalent to BDF and SDF semantics</td>
</tr>
<tr>
<td></td>
<td>• This domain only generates comments, allows viewing of the generated</td>
</tr>
<tr>
<td></td>
<td>comments, and displays a Gantt Chart for parallel schedules</td>
</tr>
<tr>
<td></td>
<td>• Can only support scalar data types on the input and output ports</td>
</tr>
<tr>
<td></td>
<td>• All derived domains obey SDF semantics</td>
</tr>
<tr>
<td></td>
<td>• Useful for testing and debugging schedulers</td>
</tr>
<tr>
<td></td>
<td>• Targets include bdf-CGC which supports BDF, default-CGC which supports</td>
</tr>
<tr>
<td></td>
<td>SDF semantics, TclTk_Target which supports SDF and must be used when</td>
</tr>
<tr>
<td></td>
<td>Tcl/Tk primitives are present, and unixMulti_C which supports SDF</td>
</tr>
<tr>
<td></td>
<td>semantics and partitions the graph for multiple workstations on a network</td>
</tr>
<tr>
<td>Code Generation in C (CGC)*</td>
<td>Uses data flow semantics and generates C code</td>
</tr>
<tr>
<td></td>
<td>• Generated C code is statically scheduled and memory used to buffer</td>
</tr>
<tr>
<td></td>
<td>data between primitives is statically allocated</td>
</tr>
<tr>
<td>Code Generation for the Motorola DSP 56000</td>
<td>Synthesizes assembly code for the Motorola DSP56000 family</td>
</tr>
<tr>
<td>(CG56)*</td>
<td>Relatively new and experimental</td>
</tr>
<tr>
<td></td>
<td>• Generates VHDL code</td>
</tr>
<tr>
<td></td>
<td>• VHDL domain supports SDF semantics whereas VHDLB supports behavioral</td>
</tr>
<tr>
<td></td>
<td>models using native VHDL discrete event model of computation</td>
</tr>
<tr>
<td></td>
<td>• Many targets to choose from</td>
</tr>
<tr>
<td></td>
<td>• VHDL domain is good for modeling systems at functional block level</td>
</tr>
<tr>
<td></td>
<td>whereas VHDLB is good for modeling behavior of components and their</td>
</tr>
<tr>
<td></td>
<td>interactions at all levels of abstraction.</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of various domains
6.5 Targets

A target coordinates the scheduling and implementation of algorithms described in a particular domain. As part of the coordination, a target may provide an interface to software (compiler, assembler, simulator, etc.) or hardware. A typical domain supports many different types of schedulers and many different implementation technologies. This is made possible by having many different target types for each domain.

In a simulation domain, the target selects the scheduler to use (there can be several schedulers in a single domain) and starts and stops a simulation. In a code generation domain, the target also selects the scheduler, but then also generates the code, compiles it, and runs it on a suitable platform. Targets can be defined hierarchically; for example, a multiprocessor target may consist of several, possibly heterogeneous execution platforms, each specifying itself as a target. In this example, the top level target might handle the partitioning and interprocessor communication, and the lower level targets might handle the code generation, compilation, and execution. Targets play a much bigger role in code generation domains than in simulation domains.

SDF Domain Targets

As is typical of simulation domains, the SDF domain does not have many targets. These targets are visible in the Property Editor under the Model Properties tab. The default-SDF target is normally selected by default.

6.5.0.1 Default SDF target

The default SDF target has a simple set of options: The SDF scheduler determines the order of execution of stars in a system at start time. It performs most of its computation during its setup() phase. If the loopScheduler target parameter is DEF, then we get a scheduler that exactly implements the method described in [LM87a] for sequential schedules. If there are sample rate changes in a program graph, some parts of the graph are executed multiple times. This scheduler does not attempt to generate loops; it simply generates a linear list of blocks to be executed. For example, if star A is executed 100 times, the generated schedule includes 100 instances of A. A loop scheduler will include in its "looped" schedule (where possible) only one instance of A and indicate the repetition count of A, as in (100 A). For simulation, a long unstructured list might be tolerable, but not in code generation. (The SDF schedulers are also used in the code generation for a single processor target).

Neglecting the overhead due to each loop, an optimally compact looped schedule is one that contains only one instance of each actor, and we refer to such schedules as single appearance schedules. For example, the looped schedule (3 A)(2 B), corresponding to the firing sequence AAABB, is a single appearance schedule, whereas the schedule AB(2 A)B is not.

By setting the loopScheduler target parameter to CLUST, we select a scheduler developed by Joe Buck. Before applying the non-looping scheduling algorithm, this algorithm collects actors into a hierarchy of clusters. This clustering algorithm consists of alternating a "merging" step and a
"looping" step until no further changes can be made. In the merging step, blocks connected together are merged into a cluster if there is no sample rate change between them and the merge will not introduce deadlock. In the looping step, a cluster is looped until it is possible to merge it with the neighbor blocks or clusters. Since this looping algorithm is conservative, some complicated looping possibilities are not always discovered. Hence, even if a graph has a single appearance schedule, this heuristic may not find it.

Setting the loopScheduler target parameter to ACYLOOP results in another loop scheduler being selected, this one developed by Praveen Murthy and Shuvra Bhattacharyya [Mur96][BML96]. This scheduler only tackles acyclic SDF graphs, and if it finds that the universe is not acyclic, it automatically resets the loopScheduler target parameter to CLUST. This scheduler is optimized for program as well as buffer memory. Basically, for a given SDF graph, there could be many different single appearance schedules. These are all optimally compact in terms of schedule length (or program memory in inline code generation). However, they will, in general, require differing amounts of buffering memory: the difference in the buffer memory requirement of an arbitrary single appearance schedule versus a single appearance schedule optimized for buffer memory usage can be dramatic. Again, in simulation this does not make that much difference (unless really large SDF graphs with large rate changes are being simulated of course), but in code generation it is very helpful. Note that acyclic SDF graphs always have single appearance schedules; hence, this scheduler will always give single appearance schedules. If the logFile target parameter is set, then a summary of internal scheduling steps will be written to that file. Essentially, two different heuristics are used by the ACYLOOP scheduler, called APGAN and RPMC, and the better one of the two is selected. The generated file will contain the schedule generated by each algorithm, the resulting buffer memory requirement, and a lower bound on the buffer memory requirement (called BMLB) over all possible single appearance schedules.

NOTE: The ACYLOOP scheduler modifies the System during its computations; hence, scripted runs that depend on the System remaining in the original state, cannot be used with this scheduler. Since the System reverts to its original state after a run sequence, the ACYLOOP scheduler will work fine in normal usage.

6.5.0.2 The loop-SDF target

An exact looping algorithm, available in an alternative target called the loop-SDF target, was developed by adding postprocessing steps to the CLUST loop scheduling algorithm. For lack of a better name, we call this technique "SJS scheduling", for the first initials of the designers (Shuvra Bhattacharyya, Joe Buck, and Soonhoi Ha). In the postprocessing, we attempt to decompose the graph into a hierarchy of acyclic graphs [BBHL93], for which a compact looped schedule can easily be constructed. Cyclic subgraphs that cannot be decomposed by this method, called tightly interdependent subgraphs, are expanded to acyclic precedence graphs in which looping structures are extracted by the techniques developed in [BL94a] and extensions to these techniques developed by Soonhoi Ha. This scheduling option is selected when the loopTarget is chosen instead of the default SDF target.

The target options are:

- logFile
6.5 Targets

- schedulePeriod

They have the same interpretation as for the default target, but in the loop-SDF target, schedulePeriod has an initial default of 10000.0. When there are sample rate changes in the program graph, the default SDF scheduler may be much slower than the loop schedulers, and in code generation, the resulting schedules may lead to unacceptably large code size. Buck’s scheduler provides a fast way to get compact looped schedules for many program graphs, although there are no guarantees of optimality. The somewhat slower SJS scheduler is guaranteed to find a single appearance schedule whenever one exists [BBHL95]. Furthermore, a schedule generated by the SJS scheduler contains only one instance of each actor that is not contained in a tightly interdependent subgraph. However, neither the SJS scheduler nor Buck’s scheduler will attempt to optimize for buffer memory usage; this need is met by the ACYLOOP scheduler chosen through the default-SDF target as described above, for acyclic graphs. Algorithms for generating single appearance schedules optimized for buffer memory systematically for graphs that may contain cycles have not yet been implemented.

The looped result can be seen by setting the logFile target parameter. That file will contain all the intermediate procedures of looping and the final scheduling result. The loop scheduling algorithms are usually used in code generation domains, not in the simulation SDF domain. Refer to the Code Generation domain documentation for a detailed discussion to the section on “Schedulers”.

<table>
<thead>
<tr>
<th>Target</th>
<th>Target Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>logFile</td>
<td>(STRING) Default = The name of a file into which the scheduler will write the final schedule. The initial default is the empty string.</td>
</tr>
<tr>
<td>loopScheduler</td>
<td>(STRING) Default = DEF A String specifying whether to attempt to compact the schedule for forming looping structure (see below). Choices are DEF, CLUST, ACYLOOP. The case does not matter: DEF, def, Def are all the same. For backward compatibility, ”0” or ”NO”, and ”1” or ”YES” are also recognized, with ”0” or ”NO” being DEF, and ”1” or ”YES” being CLUST.</td>
</tr>
<tr>
<td>schedulePeriod</td>
<td>(FLOAT) Default = 0.0 A floating-point number defining the time taken by one iteration through the schedule. This is not needed for pure SDF systems, but if SDF systems are mixed with timed domains, such as DE, then this will determine the amount of simulated time taken by one iteration.</td>
</tr>
</tbody>
</table>

Table 6.2: SDF Targets and Target Parameters

6.5.0.3 SDF to PTCL target

The SDF-to-PTCL target was introduced in Ptolemy 0.6. This target is substantially incomplete, we give a rough outline below. We hope to complete work on the SDF-to-PTCL target in a later
release. The SDF-to-PTCL target uses CGMultiInOut stars to generate abstract ptcl graphs which capture the SDF semantics of a simulation SDF universe. These abstract graphs can then be used to test SDF schedulers.

The ptcl output filename will use the universe name as a prefix, and append .pt to the name (e.g., the ptcl output for the butterfly demo would be in butterfly.pt). Currently the directory that will contain the ptcl output is hardwired to ~/PTOLEMY_SYSTEMS/ptcl/. You may need to create this directory by hand. The most interesting aspect about the target is that it collects statistics on the execution time of each star. This is valuable for seeing the relative run-times of the various stars which can be used in code generation. It collects statistics by running the scheduled universe, accumulating elapsed CPU time totals for each star. This new target does not call the wrapup methods of the stars, so you will not see XGraph outputs.
Chapter 7
Simulation with MLDesigner

The **Switch to Simulation Mode** icon is visible on the toolbar when a **System** is open in the Model Editor Window. Click the icon to switch to simulation mode. The simulation run control icons are now visible on the toolbar. These are, from left to right:

![Simulation Icons on the Toolbar](image)

**Figure 7.1: Simulation Icons on the Toolbar**

1. **Edit Mode**. The run controls apply to the system that was active when you clicked the **Simulate** icon. If you want to run another system without physically closing the active system, first click the **Switch to Edit Mode** icon before selecting a new system to simulate.

2. **Go**. To start the simulation click this icon. Notice the Progress tab at the bottom of the GUI showing the progress of the simulation.

3. **Step Into** - The simulation is advanced to the next block. If the next block in the system is a hierarchical module, the first block within the module is highlighted. In the example in fig. 7.2. The firing order is Step 1, Step 2.1, Step 2.2, Step 2.3, Step3, Step 4.

4. **Step Over** - The simulation is advanced to the next block on the same hierarchical level in the system. The firing order is Step 1, Step 2, Step 3, Step 4. Clicking Step Over when Step 2 is highlighted causes Step 2.1, Step 2.2 and Step 2.3 to execute and the simulation pauses at Step 3.

5. **Finish** - If the simulation has advanced to an instance within a module it is possible to execute all events in the module and advance to the next model instance outside the module. Clicking **Finish** while Step 2.1 of the hierarchical model Step 2 is highlighted advances the simulation to Step 3. Clicking **Finish** while step 3 is highlighted causes the simulation to run through to the end.

6. **Early End** Calls the wrap up command and all graphs are produced if enough information was available for the end conditions to be met.

7. **Abort**. The simulation is halted and reset.

8. **Graphical Animation**. Model instances and ports are highlighted as they fire during the simulation. This slows the simulation down a lot but is useful in connection with Breakpoints for debugging systems.

**NOTE:** The formal ports of the modules are not highlighted because they do not exist in...
7 Simulation with MLDesigner

kernel during simulation. A system containing no wormholes is, after compilation, nothing else than a plain network of primitive instances.

9. **Textual Animation.** Opens the Animation console. A textual report of each port firing or model instance firing is printed to the Animation console. This is useful for checking firing order of model elements and wormholes.

10. The **Debug Mode** icon activates/deactivates all breakpoints in the active system.

11. **Generate Extern.** A single click on the icon executes the simulation using the method chosen in the **Settings** configuration of the main menu. The default is **Generate C++.** The files are created in the target directory but are not compiled (see sec. 7.1.1 for more information about compiling and running external simulations.) To generate a PTcl file for simulation, click and hold the mouse over the **Generate Extern** icon for more than 1 second. A submenu displays the two options **Generate C++** and **Generate PTcl.** The default simulation method can be changed under the **Run control** item in the Settings menu. Check the appropriate radio button under the **Preferred external simulation method.**

12. **Generate & Run Extern.** Generates all object files needed to run a simulation extern (independent of the MLDesigner GUI), compile and execute the simulation once. All files are written to \$MLD_USER/SYSTEMS/Library Name/System Name/.. unless otherwise specified in the **Target Directory** property of the **Simulation Properties** window.

---

**Figure 7.2:** Step Into Example

**Figure 7.3:** Step Over Example

### 7.1 Generate Extern

The following options are available when executing a simulation Extern.
7.1 Generate Extern

7.1.1 Generate C++

The Generate C++ option on the toolbar starts the process of generating all files needed to run a simulation independent of the MLDesigner GUI. The simulation will, however, not run Extern unless MLDesigner is installed on the machine where you execute the system; a result of the dynamic linking mechanism.

All files are written to "TargetDirectory"/"LibraryName"/"SystemName"/..., where "TargetDirectory" means the content of the same named property of the "Simulation Properties", which is set to $MLD_USER/SYSTEMS/ as default. Here is a list of the files which are generated from MLD.

NOTE: Note that "SystemName" stands for the according system name, e.g. "butterfly".

Here is a list of the files which are generated from MLD.
Before the system can be executed, it must be compiled. In the directory where the files are stored type make, or better, start the script ending with "-build". If compilation is successful, the directory contains several more files listed below.

NOTE: Note that $MLDARCH means the output of "mldarch" on your site.

As you can see, now there is a file called "SystemName". To give the simulation a try, you can start it by typing "./SystemName". If your system uses X, Tk, Qt or similar primitives, you have to start the simulation with "./SystemName -x", which is the common case. If you are unsure about this, use -x in every case. The file with the extension .param is parsed when the simulation is executed. This file can be edited manually and the simulation can be rerun independent of the MLDesigner GUI. The simulation parameters for each simulation run are stored in a file with the extension .param.* where * indicates a unique simulation number.

7.1.2 Generate PTcl Extern

The Generate PTcl option on the toolbar opens a dialog where you can create a location to save the .ptcl file in. It is possible to create a new directory and/or change the default name of the
file | role
---|---
SystemName | script for compiling with a lockfile mechanism, to avoid parallel compilation of the same architecture.
SystemName-mldstart | script used by mld to start distributed external simulations, do not use it.
*.cpp, *.cc, *.h | source code files, where for each module one cc file is generated
SystemName.params | parameter file
makefile | makefile
version | file containing the version number under which these files are generated. To avoid conflicts, MLDesigner will generate new source files if the version number found here does not match the version number of MLDesigner itself
obj/* | object files of the user primitives

Table 7.1: generated files

<table>
<thead>
<tr>
<th>file</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemName</td>
<td>script for starting the simulation</td>
</tr>
</tbody>
</table>
SystemName.$MLDARCH | simulation binary file
*.$MLDARCH.mo | module object files
SystemName.moc | moc file

Table 7.2: files in extern directory

The two files *.params and *.ptcl are generated in the specified location. The *.params file contains the default settings for all visible parameters (parameters that are visible in the Simulation Properties window). After every simulation a *.params.* file is created containing the settings for the simulation. The filename is extended by a unique number indicating the simulation number. This feature means it is possible to execute a number of simulations with different settings while keeping a record of the parameter settings for each simulation. It makes sense to save the results of each simulation with the name extended by the simulation number.

### 7.1.2.1 Run PTcl Extern

- Via the MLDesigner Console
In the MLDesigner Command Console cd to the directory where the files are stored and type:

```bash
execute <systemname.ptcl>
```

- **Via the Shell**
  An MLDesigner PTcl shell makes it possible to execute PTcl simulations independent of the MLDesigner GUI. To execute the simulation type:

```bash
$MLD/ptclsh -x SystemName.ptcl
```

It is also possible to run PTcl simulations in batch mode using the `at` command followed by `$MLD/ptclsh -x SystemName.ptcl`.

### 7.1.2.2 Parameter Values

The `.params` file can contain a list of numbers separated by semi colons or can be a list of numbers defined using the following syntax:

```
for -1.0 to 1.0 step 0.1
```

which would be equivalent to a list of numbers separated by semicolons such as

```
-1.0; -0.90; -0.80; ....to 1.0.
```

These values represent the initial particle value per iteration of the relevant Parameter Set. A typical file of a simulation with three parameter sets looks like this:

```
Parameter_Set {
    GlobalSeed    : 1234567890
    PTclScript    :
    RunLength     : 1000
    new_parameter_1: for -1.0 to 1.0 step .1
}

Parameter_Set {
    GlobalSeed    : 1234567890
    PTclScript    :
    RunLength     : 1000
    new_parameter_1: for 2 to 10 step 2
}

Parameter_Set {
    GlobalSeed    : 1234567890
    PTclScript    :
    RunLength     : 1000
}
```
new_parameter_1: 3; 30; 300
}

It is not possible to create new parameters by editing the file: it is only possible to edit the values of parameters in the .params file.

### 7.1.3 Execute on other Platforms

It is possible to execute systems designed under one operating system such as Linux on another operating system such as Solaris, but only if the system DOES NOT contain user-defined primitives. User-defined primitives that are compiled for external simulation on one system will not be compatible with another operating system. If the system contains user defined primitives they are exported to the $MLD/SYSTEMS/.. directory as .o object files. The primitive itself is not exported. The simulation can only run if the user-defined primitive has been compiled on the machine where you want to execute the system. To do so, export the system using the Export Library feature available via the context menu in the tree view window. For more information on exporting libraries see sec. 2.4.2.

### 7.1.4 Environment Variables

Extern files are sensitive to MLDesigner environmental variable settings. Changes to the default settings, if made incorrectly, could lead to unsuccessful compilation of extern systems.

The environment variables `SIM SYSTEM` and `SIM EXTERN` are set to the URL of the simulated system and the directory where the code of the external simulation is generated (specified by the Target Parameter property in the Simulation Properties window).

### 7.2 Generate & Run Extern

Generate & Run Extern creates all files and compiles them. Once all files are generated the system is executed once in the background. The option to generate and run using PTcl or C++ can be selected. To select a preferred simulation method, click and hold the left mouse button with the cursor placed over the Generate & Run Extern icon. Select one of the methods available.

A single click on the icon will run the simulation in default mode. The default setting can be set via the Settings configuration. Under the Run Control item the option Preferred external simulation method.

**NOTE:** With external simulations, model source code is regenerated only if a model has changed. The .cpp file must be updated in this case. As a result the compile process is a lot faster when recompiling after minor changes are made.
7.2 Generate & Run Extern

7.2.1 External Parameters

A number of [P]arameters can be defined for a system. These parameters are a class of arguments used to control the functions of a system. Parameters on system level are not used to control the model component on embedding, but to control its behavior during simulation. You can set simulation parameters in system models and can change them while in simulation mode.

Parameter definitions on system model level have an additional property called Scope. This property is used to specify whether the parameter is shown as a user interface control element in the Simulation Properties window. The system model defines the default values of the parameters. These values can be changed anytime in the simulation control window without any modification of the system model. To change the default settings, click the Switch to Edit Mode icon before changing the parameter values. Save the model and return to simulation mode by clicking the relevant icon.

7.2.2 Example

Continued from page 3-21

Let us come back to the example system model we developed in ch. 3. The system model Sine Modulator System defines the two parameters to set the frequencies for the sine wave generators. Follow the steps below to make these parameters visible within the simulation control window.

- Select the Model Editor Window that contains the Sine Modulator System model.
- Click on the Sine Modulator System model background.
- Click on the plus sign to the left of the parameter SignalFrequency.
- Set the Scope attribute to true.
- Click on the Sine Modulator System model background.
- Click on the plus sign next to the parameter CarrierFrequency.
- Set the Scope attribute to true.
- Save the model using the tool button Save Model.

The simulation control buttons located on the toolbar perform the following actions.

The GO button starts the simulation run.

The PAUSE button merely interrupts the simulation.

Use the Step Forward button to continue the simulation run.

Every time the STEP Forward button is pressed, the simulation continues for one step. That is, the scheduler that controls the simulation fires the next block (it calls the go method of the next scheduler). The Step Increment parameter in the Simulation Properties window allows you to step through the simulation in larger steps, i.e., setting this value to 5 will advance the simulation by five firings of the actual ports represented on both the system level as well as the subsystem.
level. This applies to both homogenous modules and wormholes.

With the **ABORT** button, the simulation is stopped and re-initialized. If needed you can click **Clear** in the Textual Animation window to clear the results of the aborted simulation. The **Early End** button initializes the wrapup script. All relevant graphs associated with the simulation display the results of the simulation if enough information was available when the simulation was ended.

The **Switch to Edit Mode** icon closes the **Simulation Properties** window but leaves the Ptolemy Xgraph windows open.

You have the following options:

- Save EPS (saves the results in Postscript format)
- Print (prints the graph directly to your default printer)
- Zoom In- this is done by left click on the mouse and dragging a rectangle, from left to right, over the area you want to examine more closely.
- Zoom Out- is achieved by dragging a rectangle over the area you want to expand, from right to left.
- Fill (resets the zoom level so the whole graph is visible in the display window after you have zoomed in on a section of the graph)
- Close (closes the relevant graph)
- Close All (closes all graphs resulting from the simulation)

If the model uses Tk primitives the Tk user interface elements appear within **tclRunControl** windows (see fig. 7.5). For details on customizing the GUI see sec. 16.2.

![Figure 7.5: Additional controls in TclScript panel](image)

Figure 7.6 and fig. 7.7 show the simulation control window for the SDF example system MLDLibraries/DEMO/SDF Demo/Tcl/Tk/animatedLMS. Using these Tk elements, the model parameters can be controlled during the simulation run.

### 7.3 Debug Mode

The debug features of MLDesigner are only available via the context menu when MLDesigner is in simulation mode. This is an interactive tool for locating and diagnosing inconsistencies or bugs in your modules and systems. With breakpoints, you can specify where and when you want a system to pause during a simulation.
7.3 Debug Mode

Figure 7.6: Simulation control window with Tk slider elements

Figure 7.7: Simulation control window with Tk slider elements
7.3.1 Place a Breakpoint

To place a breakpoint you need to switch to simulation mode. Click the **Switch to simulation Mode** icon on the toolbar and activate the context menu over the model instance where you would like to place the breakpoint. Select **Add Breakpoint** from the context menu. If there are existing breakpoints on the model instance you chose, the new breakpoint will get a unique identifier. You have the option of selecting and enabling multiple breakpoints but this only makes sense if you have conditional breakpoints and you want to stop a simulation if one or another condition is met on a model instance. At the moment conditional breakpoint are not working but they will be implemented soon.

![Breakpoint Properties Window](image)

Figure 7.8: Breakpoint Properties Window

7.3.2 Unconditional Breakpoints

It is now possible to place unconditional breakpoints. The simulation pauses as soon as a model instance containing a breakpoint fires or an event is produced at the specified point in the simulation. You can adjust these breakpoint to ignore a certain number of firings or events before pausing the simulation by setting the value in the **Ignore Count** input field.

7.3.2.1 Ignore Count Breakpoints

The input field **Ignore Count** sets the amount of times a model instance or port fires before stopping the simulation. Once the instance has fired the specified number of times, the breakpoint becomes an unconditional breakpoint and stops the simulation every time it fires. Enter a numerical value in the **Ignore Count** input field.

7.3.3 Module Breakpoints

If a module is instantiated more than once in a system the option of placing a module breakpoint is available. In the top of the **Add New Breakpoint** dialog a list of all modules is displayed. It is possible to select one or more modules from this list by mouse-click. To select all in the list select the check box **Module Breakpoint**. When a selected module fires the simulation will pause.
In short, breakpoints are useful when you know that at a certain point in your program, or, when a certain condition occurs, a problem exists. By defining an appropriate breakpoint, you can stop a simulation at the point in the simulation where the problem appears.

In simulation mode the tool tip text facility no longer displays a name and description for the model instance but rather displays values and parameters of the model instance or input/output port. There are a number of types of breakpoints depending on the type of model instance. The following types are implemented in MLDesigner:

- Instance Breakpoint,
- Port Breakpoint,
- Memory Breakpoint,
- Event Breakpoint,
- Resource Breakpoint,
- FSM Transition Breakpoint,
- FSM State Breakpoint, and
- FSM History Breakpoint.

7.3.3.1 Simulate from Command line

It is possible to run PTcl systems or C++ systems from the command line once all files have been generated. To run a C++ simulation external from the command line enter

```bash
./SystemName
```

at the `~/MLD/SYSTEMS/directory` prompt. The simulation will run in the background as normal.

To revert back to opening MLDesigner without the debugger type one of the following lines:

```bash
export MLD_PREBIN=
or
unsetenv MLD_PREBIN
```

Once again the first line is for `bash` or `shell` command lines and the second line is used for `tcsh` and `csh` type command lines.

7.4 Simulation with Parameter Sets

It is possible to run simulation permutations using a combination of multiple iterations and a number of parameter sets. The two examples covered here are relatively simple and are intended to show the order of simulation achieved by using more than one set of parameters in systems. This option can also save you a lot of time in terms of running a simulation once in order to display a variety of simulation results. The best way to explain this is by way of example.

We need to create a simple system using a Ramp primitive and an XGraph. Proceed as follows:

- Click the new model icon on the toolbar.
Choose **System** from the **Type of Model** drop-down menu.

In the **Logical Name** field enter **SimpleParamSet**.

Select a Library where you will easily find your new system.

In the **File View** of the tree view window go to `MLD Libraries/SDF/Sources/` and click and drag the **Ramp** primitive into your blank system window.

Press Control and click the **Ramp** primitive with the left mouse button. The cursor changes to a plus sign when you move the mouse indicating you have a copy of the Ramp primitive. Drag the copied primitive to the appropriate position as shown in fig. 7.9. Release the mouse button and thereafter the control key.

Open the **Sinks** library in the **SDF** directory. Click and drag the **XYgraph** into your system.

Connect the ports as shown in the diagram.

---

![Diagram of System](image)

Figure 7.9: Data Type Hierarchy

You are now ready to create new parameters for your first parameter set.

Click on the background of the Model Editor Window to deselect any model instances in the system. Activate the context menu in the **Name** column of the **Model Properties** window. Choose **New Parameter** from the context menu. A new parameter is created with the name **Parameter1**. Repeat the procedure to create a second field called **Parameter2**.

The next step is to give values to the parameters:

- Scroll down to the fields **Parameter1** in the Model Properties window. Enter the Values **1; 2; 3** separated with semicolons.
- In the **Parameter2** input field type **4; 5; 6** in the input field.

You now need to link the parameters **Step** and **Value** of the model instances **Ramp#1** and **Ramp#2** to the **Parameter1** and **Parameter2**:
Click on the **Ramp#1** primitive model instance in the Model Editor Window.

Activate the context menu of the **Step** parameter in the **Instance Properties** window. Choose **Link To** and select **Parameter2** from the sub-menu.

Activate the context menu for the parameter **Value** and select **Parameter1** from the sub-menu.

Click on the **Ramp#2** primitive model instance in the Model Editor Window.

Activate the context menu of the **Step** parameter in the **Instance Properties** window. Choose **Link To** and select **Parameter2** from the sub-menu.

Activate the context menu for the parameter **Value** and select **Parameter1** from the sub-menu.

Click the **Switch to Simulation Mode** icon. In the Simulation Properties window set the **Run Length** to 2 to see the results of the simulations clearly. Large simulations often produce many graphs that all appear the same at first glance but are in fact all very different. This is a result of the scale factor of the XYgraph. You will often only see the differences when you use the Zoom function (click and drag the mouse over a section of the graph).

The sets of simulations carried out as a result of these settings are in table **7.3**:

<table>
<thead>
<tr>
<th>Graph Number</th>
<th>Start Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 7.3: Permutation of Simulation with Three Step Parameters and Three Start Value

The next step is to create a second Parameter Set and see what results or permutations are achieved. Between the tree view window and the properties editor is a small window titled **Parameter Set 1**. Activate the context menu in this window and choose **New Parameter Set**. All default values from Parameter Set 1 are set as default value in parameter set 2. For this example proceed as follows:

- Choose **Parameter Set 1** from the Parameter Set window drop-down menu.
- Set the Value of **Parameter1** to 1; 2; 3 and the Value of **Parameter2** to 4.
- Choose **Parameter Set 2** from the Parameter Set window drop-down menu.
- Set the Value of **Parameter1** to 3 and the Value of **Parameter2** to 5; 6.
7 Simulation with MLDesigner

<table>
<thead>
<tr>
<th>Graph Number</th>
<th>Start Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 7.4: Permutation of Simulation with Two Parameter Sets and Three Start Value

7.5 Saving Simulation Results

You may wish to write the simulation results to a file where you can analyze the data later or use the data as input for other software tools. There are a number of ways to do this.

7.5.1 Write Simulation Results to the Console

Start MLDesigner from the Command Shell with the parameter \(-c\) as appendage.

\[
\text{mld2 } -c
\]

A system containing a \texttt{Print} primitive will print simulation results to the command shell. An example is found in Library View/Demos and is called \texttt{MinDirectivity}. Here the default [P]arameter Value for \texttt{Printer.input=1#1} are set as \langle \texttt{stdout} \rangle. The results of the simulation are printed to the console.

7.5.2 Write Simulation Results to File

With the previous MinDirectivity example, it is also possible to write the simulation results to a file. The MLD Libraries directory is, however, write protected so you must first copy the system to your own library folder. There you will be able to edit certain parameters. Activate the context menu and select \textbf{Save As} to create a copy of MinDirectivity in your library. The new copy is automatically opened and active in the design area. Proceed as follows:

- Click on the \texttt{Printer.input=1#1} primitive in the Design Window. The Property Editor will display the \textbf{Instance Properties} for the primitive.
- Click in the [P]arameter Field \texttt{fileName} and click the folder icon to open the directory explorer.
- Select a folder and open the folder using the explorer. Type a name for the file. The file will be created automatically if it does not exist, unless you do not have write permissions to that folder.
- Run the system to write the results to the file.
The same is also true for printing the results of a XGraph to a file. A list of co-ordinates are saved in this case.

Using a program such as MLDesigner’s pxgraph you can view the file later. pxgraph is shipped with MLDesigner and can be found in the MLDesigner installation directory.

From the Command Shell you can open the results of a simulation and view them as a graph. Enter the command to call the pxgraph program followed by the full path of the file you wish to view in a 2 dimensional format.

    mldpxgraph ~/MLD/testgraph =800x400

The image is displayed using mldpxgraph. As you can see, it is possible to define how you would like the graph to appear using Ptcl script as in the =800x400 (see sec. 8.3).

Another way to open the simulation results dumped to a file is to select the instance that produced the output file, then right click on its parameter of type filename in the Property Editor Window and choose the Open option from the context menu.

It is possible to use the simulation and iteration number in the expression that defines the filename to generate distinct files for different simulations/iterations. If the filename is given by a parameter, you can specify an expression that combines the simulation and iteration number, e.g.:

    $MLD_USER/results.$absSimul.$absIter.txt
    $MLD_USER/results_${SIMNO}_${SIMIT}.txt

If the filename is generated in the primitive’s code, during initialization or when the wrapup is called, you can use the environment variables in conjunction with expandPathName function:

    char* tFileName = expandPathName("results.$SIMNO.$SIMIT");
    // open and write to file
    delete []tFileName;

Also you can use in your code the static methods of the SimControl class:

    int getCurParamSet ();
    int getAbsSimul ();
    int getAbsIter ();
    int getCurIter ();

### 7.6 Distributed External Simulations

The ability to run a simulation extern on a number of computers simultaneously is useful where complex simulations with numerous iterations result in long simulation run times.

**NOTE:** The computers used for the distributed simulation must all have the same architecture and the same version of MLDesigner otherwise the simulation will not run.
The first step towards running a distributed simulation is to ensure that all workstations have MLDesigner installed in a shared environment. The $MLD_USER environment variable should also be set to a shared environment i.e., the command mld2 should start the same MLDesigner on all computers. The computers to be used must also be registered in the /etc/hosts.equiv file.

<table>
<thead>
<tr>
<th>Workstation</th>
<th>/etc/hosts.equiv entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer 1</td>
<td>Computer2</td>
</tr>
<tr>
<td></td>
<td>Computer3</td>
</tr>
<tr>
<td>Computer 2</td>
<td>Computer1</td>
</tr>
<tr>
<td>Computer 3</td>
<td>Computer1</td>
</tr>
</tbody>
</table>

Table 7.5: Host registry for Distributed Simulations

On computer 2 and 3 the rshdeamon must be running. To test if this is the case type:

```
rsh Computer2 rsh Computer1 hostname
```

This command should return Computer1.

In the Used Computers property of the simulation Properties window shown in fig. 7.10 enter the computers you wish to use. If this field is empty the simulation runs only on localhost. A list of servers can be added to the Used Computers property by editing the ~/.mld/.mldrc file. By default localhost is entered in the <SHARED SERVERS> environment. To add servers to this list, type the server name next to or below the localhost entry. Save the file and restart MLDesigner. The servers entered now appear in the Used Computers list when MLDesigner is in simulation mode.

The simulation can now be run. Note that the rshscheduler runs each iteration parallel although only one iteration per computer can be run at a given time. If the Progress bar is visible it is possible to observe the simulation as it is executed on all computers.

### 7.7 Simulation Statistics

It is possible to print the simulation statistics to the Command console. This is a useful feature if you want to see how much processor time primitives or modules need in simulation and how many times each instance fires. This function is activated by the PTCI command stats on. Simulation time is increased by between 15 and 30 percent depending on the complexity of the model when statistics collection is active. We recommend you keep the stats turned off when possible.

The following TCL commands are valid:

```
stats on
stats off
```
To execute the simulation from the MLDdesigner command console:

   execute MySimulation.ptcl

You can write all the statistics from a given simulation to a file. First you must run the simulation with `stats on` and then type the following in the MLDdesigner Command console:

   set fd [open "MySimulation.stats" w]
   puts $fd [stats recursive]

creates the file specified with the `.stats` extension

   close $fd

Closes the file with the statistics from the last simulation.

The file now holds all the relevant information regarding the simulation and can be viewed using a text editor. In this case you should have a file called `testPacket.stats` in your working directory.

Module testPacket:
Primitive Clock#1 0.0s 52
Primitive Ramp#1 0.0s 51
Primitive Packetize#1 0.0s 51
Primitive ITerminator#12 0.0s 0
Primitive UnPacketize#1 0.0s 57
Primitive VarServer#1 0.0s 10
Primitive XMgraph.input=2#1 0.0s 102
Primitive RanGen.distribution=e#1 0.0s 10
Primitive ExecuteInOrder.Out=2#1 0.0s 10
Primitive auto-fork-node73 0.0s 51
Primitive auto-fork-node74 0.0s 51
sum Module testPacket: 0.0s

The file contains four columns:

- Column 1 - The type of model instance
- Column 2 - The Name and instance number of the model instance
- Column 3 - The processor time needed in seconds and tenths of a second.
- Column 4 - The total number of times the model instance fired.

In this case the total processor time is measured in 100ths of a second and the .stats file shows 10ths of a second.

**NOTE:** The simulation must run for more than one tenth of a second processor time.
Chapter 8

Plots, Graphs and Animation

8.1 Animation Using Tk Primitives

MLDesigner provides you with a number of primitive components that can be used to create complex animations. All these primitives are based on execution of Tcl/Tk scripts that realize Tk widgets. Tk widgets are realized by the Tk toolkit associated with the Tcl language.

![Figure 8.1: Sink primitive model components for animation](image)

Any number of animated displays can be created using Tcl/tk primitives. Some of them are opened in separate windows, others can be embedded in the simulation control window. The animation can be textual or graphical. The Tk animation primitives can be found within the Tcl/Tk library of the according domain, e.g., MLD Libraries/SDF Domain/TclTk. The most important primitives, shown in fig. 8.1, are:

- **TkPlot**: Plot \( Y \) input(s) vs. time with dynamic updating. Two styles are currently supported: *dot* causes individual points to be plotted, whereas *connect* causes connected lines to be plotted. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out and for resizing the box to fit the data in view.

- **TkXYPlot**: Plot \( Y \) input(s) vs. "X" input(s) with dynamic updating. Two styles are currently supported: *dot* causes points to be plotted, whereas *connect* causes connected lines to be plotted. Drawing a box in the plot will
reset the plot area to that outlined by the box. There are also buttons for zooming in and out and for resizing the box to fit the data in view.

| TkShowValues | Display the values of the inputs in textual form. The print method of the input particles is used, so any data type can be handled, although the space allocated on the screen may need to be adjusted. |
| TkBarGraph | Dynamically display the value of any number of input signals in bar-chart form. The first 12 input signals will be assigned distinct colors. After that, the colors are repeated. The colors can be controlled using X resources. |
| TkMeter | Dynamically display the value of any number of input signals on a set of bar meters. |
| TkShowBooleans | Display input Booleans using color to highlight their value. |
| TkText | Display the values of the inputs in a separate window, keeping a specified number of past values in view. The print method of the input particles is used, so any data type can be handled. |

The simplest animation is to print the output of ports to text widgets using a TkText primitive connected to the output port. For detailed information of the functionality of the animation primitives, please refer the online documentation of the primitives.

**Example**

To demonstrate the using of Tk primitive modules we will modify the Sine Modulator model. Follow the steps below to modify the Sine Modulator module as shown in fig. 8.2.

- Select the Model Editor Window that contains the Sine Modulator module.
- Select the Library View plane in the tree view.
- Expand the library MLD Libraries/SDF/TclTK.
- Drag item TkBarGraph into the Model Editor Window. From the Select Special Primitive dialog select TkBarGraph.input=1.
- Expand the library MLD Libraries/SDF/Control.
- Drag item Fork into the Model Editor Window. From the Select Special Primitive dialog select Fork.output=2.
- Connect the model instances as shown in fig. 8.2.
- Click on the TkBarGraph.input=2 model instance.
- Click on the value field of the parameter number_of_bars.
- Set the value to 32.
- Save the model using the tool button Save Model.

Click the Switch to Simulation Mode. Click Go to run the simulation. A Bar Chart Display appears when the simulation is complete (see fig. 8.3). Also displayed are two Ptolemy Xgraphs.
8.1 Animation Using Tk Primitives

Figure 8.2: Model

Figure 8.3: Animation
8 Plots, Graphs and Animation

8.2 Visualization Using 2D Plotting System

The 2D visual representation of simulation results, usually in the form of a set of graphs, are realized through primitives working as sinks. These primitives generate different types of graphs such as Line plots, Histograms a.s.o. All these primitives use the internal plotting system derived from the pxgraph program. The pxgraph program belongs to the original Ptolemy distribution. These primitives can be found within the Sink library of the according domain, e.g., MLD Libraries/-SDF Domain/Sinks.lib.

These primitives produce the following results:

- **XMgraph**: Generate a generic multi-signal plot.
- **XYgraph**: Generate an XY plot using the 2D plotting system. The X data is on xInput and the Y data is on input.
- **Xscope**: Generate a multi-trace plot using the 2D plotting system. Successive traces are overlaid on one another.
- **Xhistogram**: Generate a histogram using the 2D plotting system. The plot is successive. The parameter binWidth determines the bin width.
- **Waterfall**: Plot a series of traces in the style of a waterfall plot. This is a type of three-dimensional plot used to show the evolution of signals or spectra. Optionally, each plot can be made opaque, so that lines appearing behind the plot are eliminated.
- **Printer**: Print out one sample from each input port per line. The fileName parameter specifies the file to be written; the special names stdout and cout specify the standard output stream, as well as stderr and cerr specify the error output stream.

8.3 Xgraph Configuration

The sink primitives Xgraph, XMgraph and XYgraph are parameterized so you can configure them to display a variety of graph types. They can display up to 64 independent datasets using different colors and/or line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels, and a legend. There are options to control the appearance of most components of the graph. Use the parameter [P]options field in the Instance Properties window to control the appearance, size, and position of your simulation output graphs. Click the XMgraph primitive in the System Design Window. The Instance Properties tab is activated. You will now see the default values of the [P]arameter options. Click the icon to the right of the input field to activate the Property Text Editor. The commands seen in fig. 8.4 will produce an XMgraph with a bright green background, no grid lines, each data point marked by a pixel sized dot. The first dataset is called data out in the legend, and the second dataset, data lost. The graph is 800 X 150 pixels in size and is positioned on the left of the screen 600 pixels from the top.

If you have an XMgraph with three input ports and you would like to define the label for each input port, select the primitive in the Design window, with a single mouse-click. The Instance Properties tab is activated in the Properties Editor window. The syntax for the graph input label is:
-0 label1 -1 label2 -2 label3

where -0 indicates the index of the first set of output data consumed by the XMgraph.

Figure 8.4: XMgraph Configuration Syntax

The following commands can be entered in the Options field:

- \texttt{=wxh+x+y} Specifies the initial size and location of the plot window. Co ordinate values \( x \) and \( y \) are optional. (+0+0 indicates the top left corner of the screen).

- \langle\texttt{digit}\rangle\langle\texttt{name}\rangle The legend title for the corresponding dataset. The digit can range be- tween 0 and 63 with -0 being the first dataset and -1 being the second. The \texttt{name} will be seen in the legend.

- \texttt{-tk} Removes the light grey grid pattern from the graph background.

- \texttt{-bar} Specifies that vertical bars should be drawn from the data points to a base point, which can be specified with \texttt{-brb}. Usually, the \texttt{-nl} flag is used with this option. The point itself is located at the center of the bar.

- \texttt{-bb} In previous versions this was used to draw a bounding box around the data region. This option is not supported anymore. All graphs have bounding boxes.

- \texttt{-bg \langle\texttt{color}\rangle} Background color of the plot window. Color is given by name.

- \texttt{-binary} This specifies that the input is a binary file rather than an ASCII file.

- \texttt{-brw \langle\texttt{width}\rangle} This specifies the width of bars in a bar graph. The amount is specified in the user's units. By default, a bar is drawn with one pixel width.

- \texttt{-fg \langle\texttt{color}\rangle} Foreground color. This color is used to draw all the text and the normal grid lines in the window. Color is given by name.

- \texttt{-lf \langle\texttt{fontname}\rangle} Label font. All axis labels and grid labels are drawn using this font. A font name may be specified exactly (e.g. 9x15 or \texttt{-*-courier-bold-r-normal-*-140-*)} or in an abbreviated form: \langle\texttt{family}\rangle-\langle\texttt{size}\rangle. The family is the family name (like Helvetica) and the size is the font size in points (like 12). The default value for this parameter is Helvetica-12.

- \texttt{-lnx} Specifies a logarithmic x axis. Grid labels represent powers of ten.
-lny
Specifies a logarithmic y axis. Grid labels represent powers of ten.

-1x \langle x_l,x_h \rangle
This option limits the range of the x axis to the specified interval. This (along with -ly) can be used to “zoom in” to a particularly interesting portion of a larger graph.

-ly \langle y_l,y_h \rangle
This option limits the range of the y axis to the specified interval.

-m
Mark each data point with a distinctive marker. There are eight distinctive markers used by the plotting system. These markers are assigned uniquely to each different line styles on black and white machines and varies with each color on color machines.

-M
Similar to -m but markers are assigned uniquely to each of the eight consecutive datasets (this corresponds to each different line style on color machines).

-nl
Turn off drawing lines. When used with -m, -M, -p, or -P this can be used to produce scatter plots. When used with -bar, it can be used to produce standard bar graphs.

-p
Marks each data point with a small marker (pixel sized). This is usually needed with the -nl option for scatter plots.

-P
Similar to -p but marks each pixel with a large dot.

-rv
Reverse video. On black and white displays, this will invert the foreground and background colors. The behavior on color displays is undefined.

-t \langle \text{string} \rangle
Title of the plot. This string is centered at the top of the graph.

-tf \langle \text{fontname} \rangle
Title font. This is the name of the font to use for the graph title. The usage is the same as for the -lf option.

-treasure
Redundant option. Replaced by the Parameter ’Cumulation’.

-x \langle \text{unitname} \rangle
This is the unit name for the X axis. Its default is ”X”.

-y \langle \text{unitname} \rangle
This is the unit name for the Y axis. Its default is ”Y”.

Figure 8.5 shows the output of the XMGraph.input=1 instance in the Sine Modulator System model.

The following options are available in the Xgraph window:

Save EPS
To export the graph as vector graphic in EPS format. Select a location from the Save as dialog or create a new folder.

Print
Prints the graph direct to your default printer.

Fill
After using the cursor to zoom in on an area of a graph, click Fill to revert to normal view where the entire result of the simulation is visible.

Close
Closes the Result window.

Close All
Closes all result windows resulting from the simulation.
Figure 8.5: XMGraph example
Chapter 9

Modeling Using PTCL - The Ptolemy TCL Interpreter

9.1 Introduction

There are a few ways to work with MLDesigner: Firstly using the graphical user interface (GUI), and secondly using the built-in interpreter. This combination allows you to interact with MLDesigner using both graphical elements and textual commands. The MLDesigner GUI is described in sec. 2.2. The MLDesigner built-in interpreter is called PTCL, which stands for Ptolemy Tool Command Language. It conveniently operates within the Command console window of MLDesigner. This chapter holds information about using the Command console.

The Ptolemy interpreter, PTCL, accepts input commands from the keyboard, from a file, or from combinations thereof. It allows you to set up a new simulation by creating instances of blocks (primitives, modules, or wormholes), connecting them together, setting the initial values of parameters, and running/restarting the simulation. It allows simulations to be run in batch mode. PTCL is based on John Ousterhout’s tool command language TCL, pronounced ”tickle”, which is an extensible interpreted language. All the commands of Tcl are available in PTCL. PTCL extends the Tcl interpreter language by adding new commands. The underlying grammar and control structure of Tcl are not altered. Commands in Tcl are string based with relatively few constructs and a simple syntax: the basic syntax is

\[
\text{command arg1 arg2 arg3 \ldots}
\]

This chapter describes only the extensions to Tcl made by PTCL. Two excellent references on Tcl are books by Ousterhout [Ous94] and Welch [Wel97].

9.2 Global information

The interpreter has a list containing all the classes of primitives and modules it currently knows about. New primitives can be added to the list at run time by using the incremental linking facility, but this has restrictions (see sec. 9.8.4).

NOTE: Remember that the Ptolemy vocabulary for models and parameter differs from that in MLDesigner: star refers to primitive, galaxy refers to modules, universe refers to system.
9 Modeling Using PTCL - The Ptolemy TCL Interpreter

and state refers to parameter, see sec. 2. Therefore, some PTCL commands still contain the Ptolemy vocabulary.

## 9.3 Commands for Defining Simulation

This section describes using the built-in PTCL interpreter as well as the commands to build simulations, add primitives, modules, parameters, and the connections between them. These commands are entered at the prompt in the Command Console window. The commands are summarized in table 9.1.

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>galport b1 p1</td>
<td>Connect a module port galport to the port p1 of block b1.</td>
<td>9-8</td>
</tr>
<tr>
<td>animation</td>
<td>[on / off]</td>
<td>Enable or disable printing of primitive names as they fire.</td>
<td>9-14</td>
</tr>
<tr>
<td>busconnect</td>
<td>b1 p1 b2 p2 w</td>
<td>Form a bus connection of width w between the two multiportholes p1 and p2 that belongs to the blocks b1 and b2.</td>
<td>9-8</td>
</tr>
<tr>
<td>cancelAction</td>
<td>action_handle</td>
<td>Cancel an action previously registered using registerAction.</td>
<td>9-19</td>
</tr>
<tr>
<td>cd</td>
<td>[directory]</td>
<td>Change the current directory to the one given by argument directory.</td>
<td>9-17</td>
</tr>
<tr>
<td>connect</td>
<td>b1 p1 b2 p2 [delay]</td>
<td>Form a connection between the two portholes p1 and p2 that belongs to the blocks b1 and b2.</td>
<td>9-7</td>
</tr>
<tr>
<td>cont</td>
<td>num</td>
<td>Continue executing the current system num times (default: 1).</td>
<td>9-13</td>
</tr>
<tr>
<td>cursystem</td>
<td>[name]</td>
<td>Print or set the name of the current system to name.</td>
<td>9-6</td>
</tr>
<tr>
<td>defmodule</td>
<td>name { body }</td>
<td>Define a new module class with name.</td>
<td>9-10</td>
</tr>
<tr>
<td>delds</td>
<td>ds_name</td>
<td>Delete the data structure or enumeration ds_name.</td>
<td>9-26</td>
</tr>
<tr>
<td>delnode</td>
<td>name</td>
<td>Delete the node with name from the current module.</td>
<td>9-15</td>
</tr>
<tr>
<td>delprimitive</td>
<td>name</td>
<td>Delete the primitive with name from the current module.</td>
<td>9-14</td>
</tr>
<tr>
<td>delsystem</td>
<td>[name]</td>
<td>Delete the current system or the system with name.</td>
<td>9-6</td>
</tr>
</tbody>
</table>
### 9.3 Commands for Defining Simulation

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>descriptor</td>
<td>[block]</td>
<td>Return the descriptor of block (default: current module).</td>
<td>9-12</td>
</tr>
<tr>
<td>disconnect</td>
<td>b1˜p1</td>
<td>Remove the connection going to/from the specified port.</td>
<td>9-15</td>
</tr>
<tr>
<td>domain</td>
<td>[name]</td>
<td>Set the domain name as current, or print the name of the current domain.</td>
<td>9-7</td>
</tr>
<tr>
<td>domains</td>
<td></td>
<td>List the known domains.</td>
<td>9-7</td>
</tr>
<tr>
<td>execute</td>
<td>filename</td>
<td>Reads the file filename.ptcl and runs the simulation.</td>
<td>9-12</td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td>Exit MLDesigner.</td>
<td>9-18</td>
</tr>
<tr>
<td>halt</td>
<td></td>
<td>Request that the current simulation stop.</td>
<td>9-13</td>
</tr>
<tr>
<td>help</td>
<td>[command]</td>
<td>Print a short description of command, or help on help, if the argument is omitted.</td>
<td>9-18</td>
</tr>
<tr>
<td>knownlist</td>
<td>[domain]</td>
<td>List the known blocks of domain (default: current domain).</td>
<td>9-11</td>
</tr>
<tr>
<td>link</td>
<td>objfile</td>
<td>Incrementally link objfile into MLDesigner.</td>
<td>9-17</td>
</tr>
<tr>
<td>listobjs</td>
<td>class [name]</td>
<td>List parameters, ports, multiports, event, memories, and resources in the block name (default: current module).</td>
<td>9-12</td>
</tr>
<tr>
<td>matlab</td>
<td>command [arg1] [arg2]</td>
<td>Manage a Matlab process and evaluate Matlab commands.</td>
<td>9-19</td>
</tr>
<tr>
<td>mathematica</td>
<td>command [arg1] [arg2]</td>
<td>Manage a MATHEMATICA process and evaluate commands.</td>
<td>9-19</td>
</tr>
<tr>
<td>multilink</td>
<td>linker_args code.o</td>
<td>Link arbitrary code into the interpreter.</td>
<td>9-17</td>
</tr>
<tr>
<td>newds</td>
<td>ds_name ds_base</td>
<td>Define a new data structure ds_name with parent type ds_base.</td>
<td>9-24</td>
</tr>
<tr>
<td>newdsmember</td>
<td>ds_name member_name member_type [default] [subrange]</td>
<td>Define a new member member_name with type member_type for data structure ds_name.</td>
<td>9-24</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>newenum</td>
<td><code>enum_name enum_base</code></td>
<td>Define a new enumeration <code>enum_name</code> with parent type <code>enum_base</code>.</td>
<td>9-25</td>
</tr>
<tr>
<td>newenummember</td>
<td><code>enum_name</code></td>
<td>Define a new member with value <code>member_value</code> and index <code>member_index</code> for enumeration <code>enum_name</code>.</td>
<td>9-25</td>
</tr>
<tr>
<td>newevent</td>
<td><code>name scope type value</code></td>
<td>Define an event <code>name</code> for the current module with <code>type</code> and a default value.</td>
<td>9-23</td>
</tr>
<tr>
<td>newmemory</td>
<td><code>name scope type value</code></td>
<td>Define a memory <code>name</code> for the current module with <code>type</code> and a default value.</td>
<td>9-22</td>
</tr>
<tr>
<td>newparam</td>
<td><code>name type value</code></td>
<td>Define a parameter <code>name</code> for the current module with <code>type</code> and a default value.</td>
<td>9-9</td>
</tr>
<tr>
<td>newquantity</td>
<td><code>name scope</code></td>
<td>Define a quantity resource <code>name</code> for the current module with <code>type</code> and a default value.</td>
<td>9-23</td>
</tr>
<tr>
<td>newserver</td>
<td><code>name scope</code></td>
<td>Define a server resource <code>name</code> for the current module with <code>type</code> and a default value.</td>
<td>9-23</td>
</tr>
<tr>
<td>newsystem</td>
<td><code>[name] [domain]</code></td>
<td>Create a new empty system with <code>name</code> in <code>domain</code> (defaults: <code>main</code> and the current domain).</td>
<td>9-6</td>
</tr>
<tr>
<td>node</td>
<td><code>name</code></td>
<td>Create a node with <code>name</code> for use by nodeconnect.</td>
<td>9-8</td>
</tr>
<tr>
<td>nodeconnect</td>
<td><code>b1 p1 node</code></td>
<td>Connect a porthole given by block argument <code>b1</code> and port argument <code>p1</code> to a specified node with <code>name</code>.</td>
<td>9-8</td>
</tr>
<tr>
<td>numports</td>
<td><code>b1 p1 number</code></td>
<td>Force a multiporthole to have a given number of portholes.</td>
<td>9-10</td>
</tr>
<tr>
<td>paramvalue</td>
<td><code>b1 name</code></td>
<td>Print the current or initial value of parameter <code>name</code> in block <code>b1</code>.</td>
<td>9-18</td>
</tr>
<tr>
<td>paramvalue</td>
<td>`[current</td>
<td>initial]`</td>
<td></td>
</tr>
<tr>
<td>permlink</td>
<td><code>linker_args code.o</code></td>
<td>Link arbitrary code into MLDesigner permanently.</td>
<td>9-17</td>
</tr>
<tr>
<td>pragma</td>
<td><code>b1 b2 name value</code></td>
<td>Set pragma <code>name</code> to <code>value</code> for block <code>b2</code> in parent <code>b1</code>.</td>
<td>9-16</td>
</tr>
</tbody>
</table>
## 9.3 Commands for Defining Simulation

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pragmaDefaults</td>
<td>target</td>
<td>Print default values of the pragmas for the target</td>
<td>9-16</td>
</tr>
<tr>
<td>primitive</td>
<td>name class</td>
<td>Create an instance with name of a primitive from the given class.</td>
<td>9-7</td>
</tr>
<tr>
<td>print</td>
<td>[name]</td>
<td>Print a description of block name or block class (or the current module)</td>
<td>9-12</td>
</tr>
<tr>
<td>printds</td>
<td>ds_name</td>
<td>Print out the default value of the data structure or enumeration ds_name.</td>
<td>9-26</td>
</tr>
<tr>
<td>printdsnames</td>
<td></td>
<td>Lists all existing data structure and enumeration names.</td>
<td>9-26</td>
</tr>
<tr>
<td>registerAction</td>
<td>pre/post command</td>
<td>Register a Tcl command to be executed before or after primitives are fired.</td>
<td>9-19</td>
</tr>
<tr>
<td>renamesystem</td>
<td>[oldname]</td>
<td>Rename a system (default: current system).</td>
<td>9-7</td>
</tr>
<tr>
<td></td>
<td>newname</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset</td>
<td>[name]</td>
<td>Empty a system with name (default: main).</td>
<td>9-14</td>
</tr>
<tr>
<td>run</td>
<td>[num]</td>
<td>Run the current system num times (default: 1).</td>
<td>9-13</td>
</tr>
<tr>
<td>schedtime</td>
<td>[actual]</td>
<td>Print the normalized (default) or un-normalized current scheduler time.</td>
<td>9-14</td>
</tr>
<tr>
<td>schedule</td>
<td></td>
<td>Generate and print a schedule (only valid for some domains).</td>
<td>9-12</td>
</tr>
<tr>
<td>seed</td>
<td>number</td>
<td>Change or print the random number seed.</td>
<td>9-16</td>
</tr>
<tr>
<td>setmemory</td>
<td>block name value</td>
<td>Change the value of memory name of a block block to value.</td>
<td>9-22</td>
</tr>
<tr>
<td>setevent</td>
<td>block name value</td>
<td>Change the value of event name of a block block to value.</td>
<td>9-23</td>
</tr>
<tr>
<td>setparam</td>
<td>block name value</td>
<td>Change the parameter name of a block block to value.</td>
<td>9-9</td>
</tr>
<tr>
<td>setquantity</td>
<td>block name dimension capacity occupancy blocking discipline reject fit</td>
<td>Change the properties of quantity resource name of a block block to value.</td>
<td>9-23</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>setserver</td>
<td>block name dimension servers rate mechanism occupancy overhead preempt discipline reject</td>
<td>Change the properties of server resource name of a block block to value.</td>
<td>9-24</td>
</tr>
<tr>
<td>source</td>
<td>filename</td>
<td>No longer valid. See execute.</td>
<td>9-16</td>
</tr>
<tr>
<td>stoptime</td>
<td></td>
<td>Return the stop time of the current run.</td>
<td>9-13</td>
</tr>
<tr>
<td>systemlist</td>
<td></td>
<td>List the names of all defined systems.</td>
<td>9-6</td>
</tr>
<tr>
<td>target</td>
<td>[newtarget]</td>
<td>Change or display the name of the current target.</td>
<td>9-15</td>
</tr>
<tr>
<td>targetparam</td>
<td>name [value]</td>
<td>Change or display the value of a target parameter name.</td>
<td>9-15</td>
</tr>
<tr>
<td>targets</td>
<td>[domain]</td>
<td>List targets usable with domain (default: current domain).</td>
<td>9-15</td>
</tr>
<tr>
<td>topblocks</td>
<td>[name]</td>
<td>List top-level blocks of the block name (default: current module).</td>
<td>9-18</td>
</tr>
<tr>
<td>wrapup</td>
<td></td>
<td>Invoke the wrapup method of all the blocks.</td>
<td>9-13</td>
</tr>
</tbody>
</table>

Table 9.1: Summary of PTCL commands

### 9.3.1 Creating and deleting Systems

The command

```
systemlist
```

will return the list of names of systems that currently exist. The command

```
newsystem [<name>] [<dom>]
```

creates a new, empty system named name (default is main) and makes it the current system with domain dom (default is current domain). Both arguments may be omitted. If there was previously a system with the same name, it will be overwritten. The previous actual system will not be affected, unless it had the same name as the new one and will therefore be overwritten. To remove a system, simply issue the command:

```
delsystem [<name>]
```

If no argument is given, this will delete the current system. After this, the current system will be main. To find out what the current system is, issue the command:
cursystem [<name>]

With no arguments the name of the current system is returned. With one argument the current system will become name.
A system can be renamed using the syntax

renamesystem [<oldname>] <newname>

With one argument newname, renamesystem renames the current system to newname. With two arguments, the system named oldname is renamed to newname. Note that any existing system named newname is deleted.

9.3.2 Setting the domain

MLDesigner supports multiple simulation domains. Before creating a simulation environment and running it, it is necessary to establish the domain. The interpreter has a current domain which is initially the default domain SDF. The command

domain [<name>]

changes the current domain. This is only legal when the current module is empty. The argument name must be the name of a known domain. If the argument name is omitted, the command domain returns the current domain. It is possible to create wormhole interfaces between domains by including a domain command inside a module definition. The command

domains

lists the domains that are currently linked to the interpreter.

9.3.3 Creating instances of primitives and modules

The first step towards creating a system is to define the blocks - primitives and modules - to be used in the system. The command

primitive <name> <class>

creates a new instance of a primitive or module of class class, names it name, and inserts it into the current module. Any parameters in the primitive or module are created with their default values. While it is not enforced, the normal naming convention is that names begin with a lower case letter and classes begin with an upper case letter. This makes it easy to distinguish instances of a class from the class itself.

9.3.4 Connecting primitives and modules

The next step is to connect the blocks so that they can pass data among themselves using the connect command. This forms a connection between two primitives or modules by connecting their portholes. A porthole is specified by giving the primitive or module name followed by the port name within the primitive. The first porthole must be an output porthole and the second must be an input porthole. For example:
connect <block1> <port1> <block2> <port2> [<delay>]

The connect command accepts an optional integer delay parameter. For example:

connect myprimitive output yourprimitive input 1

This specifies one delay on the connection. The delay parameter makes sense only for domains that support it. The delay argument may be an integer expression with variables referring to module parameters as well.

One or both of the portholes may really be a MultiPortHole. If so, the effect of doing the connect is to create a new porthole within the MultiPortHole and connect to that (see also the numports command).

### 9.3.4.1 Netlist-style connections

As an alternative to issuing connect commands, which specify point-to-point connections, you may specify connections in a netlist style. This syntax is used to connect an output to more than one input. This is called auto-forking. Two commands are provided for this purpose. The node command creates a node with name:

node <name>

The nodeconnect command connects a porthole port of block block to a node name:

nodeconnect <block> <port> <nodename> [<delay>]

Any number of portholes may be connected to a node, but only one of them can be an output node.

### 9.3.4.2 Bus connections between multiple portholes

A pair of multiportholes can be connected using a bus connection, meaning that each multiporthole has \( N \) portholes and they all connect in parallel to the corresponding port in the other multiporthole. The syntax for creating such connections is

busconnect <block1> <port1> <block2> <port2> <width> [<delay>]

Here, width is an expression specifying the width of the bus (how many portholes in the multiporthole); and delay is an optional expression giving the delay on each connection. The other arguments are identical to those of the connect command.

### 9.3.5 Connecting internal primitives and modules to the outside

When you define a new module there are typically external connections to that module that need to be connected through to internal blocks. The alias command is used to add a porthole newport to the current module, and associate it with an input or output porthole blockport of one of the contained block within the module. The syntax is:

alias <newport> <block> <blockport>

This also works if blockport is a MultiPortHole- the module will then appear to have a multiporthole as well.
9.3 Commands for Defining Simulation

Defining parameters for a module

A parameter is a piece of data that is assigned to a module and can be used to affect its behavior. Typically the value of a parameter is coupled to the parameter of blocks within the module, allowing you to customize the behavior of blocks within the module. The newparam command adds a parameter to the current module. The form of the command is

```
newparam <name> <type> <value>
```

The name argument is the name to be given to the parameter. The type argument is the type of parameter. All standard types are supported, see table 3.4 on page 3-47. The value argument is the default value to be given to the parameter, if the programmer of the module does not change it using the setparam command described below. The default value specifies the initial value of the parameter, and can be an arbitrary expression involving constant values and other parameter names. This expression is evaluated when the simulation starts. The following parameter names are predefined: YES, NO, TRUE, FALSE, and PI. YES and TRUE have a value of 1; NO and FALSE have a value of 0; PI has the value 3.14159... Some examples are:

```
newparam count int 3
newparam level float 1.0
newparam title string "This is a title"
newparam myfreq float modulefreq
newparam angularFreq float "2*PI*$freq"
```

The full syntax of parameter initial value strings depends on the type of parameter, and is explained in sec. 3.5 on page 3-11.

9.3.6 Setting the value of parameters

The setparam command is used to change the value of a parameter. It can be used in three contexts:

- Change the value of a parameter for a primitive within the current module.
- Change the value of a parameter for a module within the current module.
- Change the value of a parameter within the current module.

The latter would normally be used when you want to perform multiple simulations using different parameter values. The syntax for setparam is:

```
setparam <block> <parameter> <value>
```

- argument block is either the name of a primitive or a module in the current module. It is the block for which the value of the parameter is to be changed. It can also be this meaning a parameter belonging to the active module itself.
- argument parameter is the name of the parameter you wish to change.
- argument value is the new value for the parameter. The syntax for value is the same as described in the newparam command. However, the expression for value may refer to the name of one or more parameters in the current module or an ancestor of the current module.

An example of the use of setparam is given in the section describing the defmodule command below.
9.3.7 Setting the number of ports in a primitive

Some primitives in MLDesigner are defined with an unspecified number of multiple ports. The number of connections is defined by the user of the primitive rather than the primitive itself. The `numports` command applies to primitives that contain such `MultiPortHoles`. It causes a specified number of `PortHoles` to be created within the `MultiPortHole`. The syntax is

```
numports <block> <port> <n>
```

where `block` is the name of a primitive within the current module, `port` is the name of a `MultiPortHole` in the primitive, and `n` is an integer, representing the number of `PortHoles` to be created. After the portholes are created, they may be referred to by appending `#i`, where `i` is an integer, to the multiporthole name, and enclosing the resulting name in quotes. The main reason for using this command is to allow the portholes to be connected in random order. Here is an example:

```
primitive summer Add
numports summer input 2
alias galInput summer "input#1"
connect foo output summer "input#2"
```

9.3.8 Defining new modules

The `defmodule` command allows you to define a new class of modules. The syntax is

```
defmodule <name> {
    command
    command
    ...
}
```

Here `name` is the name of the module type you are creating. While it is not required, we suggest that you have the name begin with a capital letter in accordance with our standard naming convention that class names begin with capital letters. The command lines may be any of the commands described before, such as `primitive`, `connect`, `busconnect`, `node`, `nodeconnect`, `numports`, `newparam`, `setparam`, or `alias`. The defined class is added to the known list, and you can then create instances of it and add them to other modules. An example is:

```
reset
domain SDF
defmodule SinGen {
    domain SDF
    # The frequency of the sine wave is a module parameter
    newparam freq float "0.05"
    # Create a primitive instance of class "Ramp" named "ramp"
```
9.4 Showing the Current Status

primitive ramp Ramp
# The ramp advances by 2*pi each sample
setparam ramp step "6.283185307179586"
# Multiply the ramp by a value, setting the frequency
primitive gain Gain
# The multiplier is set to "freq"
setparam gain gain "freq"
# Finally the sine generator
primitive sin Sin
connect ramp output gain input
connect gain output sin input
# The output of "sin" becomes the modules output
alias output sin output
}

In this example, note the use of parameters to allow the frequency of the sine wave generator to be changed. For example, we could now run the sine generator, changing its frequency to 0.02, with the interpreter input:

    primitive generator SinGen
    setparam generator freq "0.02"
    primitive printer Printer
    connect generator output printer input
    run 100

You may include a domain command within a defmodule command. If the inside domain is different to the outside domain, an object known as a Wormhole is created. This is an interface between two domains and is described in a later section.

9.4 Showing the Current Status

The following commands display information about the current state of the interpreter.

9.4.1 Displaying the known classes

The knownlist command returns a list of known classes of primitives and modules that are usable in the current domain. The syntax is

    knownlist

It is also possible to ask for a list of objects available in other domains, e.g., the command

    knownlist DE

displays objects available in the DE domain.
9.4.2 Displaying information on a the current module or other class

If invoked without an argument, the print command displays information on the current module. If invoked with an argument, the argument is either the name of a primitive or module contained in the current module, or the name of a class on the known list. The information is shown about that primitive or module. The syntax is

```
print [<name>]
```

The command

```
descriptor [name]
```

will print a short description of a block in the current module or on the known list, or of the current module if name is omitted. The commands

```
listobjs stars [<name>]
listobjs ports [<name>]
listobjs multiports [<name>]
```

will list the names of the parameters, ports, or multiports associated with the named primitive or module.

9.5 Running the Simulation

Once a simulation has been constructed using the commands previously described you can use these commands to run a simulation. (See also the execute command on page 9-16).

In the MLDesigner Command Console cd to the directory where the files are stored and type:

```
execute <systemname.ptcl>
```

Via the Shell

An MLDesigner PTcl shell makes it possible to execute PTcl simulations independent of the MLDesigner GUI. To execute the simulation type:

```
$MLD/ptclsh -x Systemname.ptcl
```

It is also possible to run PTcl simulations in batch mode using the at command followed by $MLD/ptclsh -x SystemName.ptcl.

9.5.1 Creating a schedule

The schedule command generates and returns the order in which primitives are invoked. For domains such as DE, this command returns a not-implemented message, since there is no “compile time” schedule for the DE domains. The syntax is:

```
schedule
```


9.5.2 Run Length

The `run` command generates the schedule and runs it \( n \) times, where \( n \) is the argument. The argument may be omitted, its default value is then 1. For the DE interpreter, this command runs the simulation for \( n \) time units, and \( n \) may be a floating point number. If omitted the default value is 1.0. If this command is repeated while the simulation is running the simulation is restarted from the beginning. If animation is enabled, the full name of each primitive will be printed to the standard output when the primitive fires. The syntax of the command is:

\[
\text{run } [\langle n \rangle]
\]

9.5.3 Continuing a simulation

The `cont` command continues the simulation for \( n \) additional steps, or time units. If the argument is omitted, the default value of the argument is the value of the last argument given to a `run` or `cont` command or 1.0 if no argument was given. The syntax is

\[
\text{cont } [\langle n \rangle]
\]

Wrapping up a simulation

The `wrapup` command calls the `wrapup` method of the current target, which in turn calls the `wrapup` method of each primitive, signaling the end of the simulation run. The syntax is

\[
\text{wrapup}
\]

9.5.3.1 Interrupting a simulation

The command

\[
\text{halt}
\]

requests a halt of the current simulation. Note that the halt does not occur immediately. This merely registers the request within the scheduler and depends on the type of domain where the simulation is being executed. The scheduler will halt the simulation on completion of a cycle. This is especially useful within Tcl primitives.

9.5.3.2 Obtaining the stop time of the current run

The command

\[
\text{stoptime}
\]

returns the time until which the current simulation will run. Tcl/Tk primitives can use this command in their `setup` or `go` methods to find out the stop time of the current run.
9 Modeling Using PTCL - The Ptolemy TCL Interpreter

9.5.3.3 Obtaining time information from the scheduler

The command

\[ \text{schedtime [actual]} \]

returns the current time from the top-level scheduler of the current system. If the target has a parameter named \text{schedulePeriod}, then the returned time is divided by this value. Using the \text{schedtime} command with argument \text{actual} returns the scheduler time without dividing by \text{schedulePeriod}.

In SDF domains, \text{schedtime actual} should return the number of iterations. In SDF domain, \text{schedulePeriod} is usually set to 0, since SDF has no notion of time, and to a timed domain, such as DE, SDF systems appear to fire instantaneously.

9.5.3.4 Animating a simulation

The \text{animation} command can be used to display, on the standard output, the name of each primitive as it runs. The syntax

\[ \text{animation [on|off]} \]

Typing \text{animation on} enables animation, while \text{animation off} disables it. Typing \text{animation} returns whether animation is on or off.

9.6 Undo Commands

The commands in this section remove part or all of the structure you have built with previous commands.

9.6.1 Resetting the interpreter

The \text{reset} command replaces the system \text{main} or a named system by an empty system. Any \text{defmodule} definitions you have made are still remembered. The syntax is

\[ \text{reset [<system>]} \]

9.6.2 Removing a primitive

The \text{delprimitive} command removes the named primitive from the current module. The syntax is

\[ \text{delprimitive [<system>]} \]

where name is the name of the primitive.
9.6.3 Removing a connection

The `disconnect` command reverses the effect of a previous `connect` or `nodeconnect` command. The syntax is:

```
    disconnect <block> <port>
```

where `block` and `port`, taken together, specify one of the two connected portholes. Note that you can disconnect by specifying either end of a porthole for a point-to-point connection.

9.6.4 Removing a node

The `delnode` command removes a node from the current module. The syntax is:

```
    delnode <node>
```

9.7 Targets

MLDesigner uses a structure called a `target` to control the execution of a simulation, or to control code generation, compilation, and execution. If you issue no target commands, your target will have the name `default-XXX`, where `XXX` is replaced by the name of the current domain. Alternative targets for simulation can be used to specify different behavior, for example, to use a different scheduler or to analyze a schematic rather than running a simulation. For code generation, the target contains information about the target of compilation, and has methods for downloading code and starting execution.

9.7.1 Available targets

The command

```
    targets [<domain>]
```

returns the list of targets available for the current domain or the domain given by the optional argument.

9.7.2 Changing the target

Using command

```
    target [<name>]
```

without argument `name` displays the target for the current system or current module, together with its parameters. Specifying an argument, changes the target to that one specified by the argument.

9.7.3 Changing target parameters

Target parameters may be queried or changed with the `targetparam` command. The syntax is

```
    targetparam <name> [<value>]
```
9.7.4 Pragmas

MLDesigner can use target pragmas as a generalization of the attribute mechanism to inform the target of your wishes. The Dynamic Data Flow (DDF) domain uses pragmas to specify the number of firings of a primitive required in one iteration. The C Code Generation (CGC) domain uses pragmas to identify any parameters that you would like to change on the command line. Using command

```
pragma <block1> <block2> <name> <value>
```

you can set pragma name to value for block block2 in parent block1. The command

```
pragmaDefaults target
```

prints the default values of the pragmas for the target.

9.8 Miscellaneous Commands

This section describes the remaining interpreter commands.

9.8.1 Loading commands from a file

For complicated simulations it is better to store your interpreter commands, at least those defining the simulation connectivity, in a file rather than typing them into the interpreter directly. This way you can run your favorite editor in one window and run the interpreter in another window, easily modifying the simulation and also keeping a permanent record. Two exceptions to this are changing parameters using the `setparam` command and running and continuing the simulation using `run` and `cont`, this is normally done interactively with the interpreter.

The command

```
execute <filename>
```

reads interpreter commands from the given filename, until the end of the file or until an error occurs. The # character indicates that the rest of the line is a comment. By convention, files meant to be read by the `execute` command should have the extension `.ptcl`

```
execute "testfile.ptcl"
```

Using the tilde notation (~) for home directories is allowed within filenames.

9.8.2 Changing the seed of random number generation

The `seed` command changes the seed of the random number generation. The default value is 1. The syntax is

```
seed <n>
```

where n is an unsigned integer.
9.8.3 Changing the current directory

The `cd` command changes the current directory. The syntax is:

```
  cd [<name>]
```

where `name` specifies the directory. If the argument is omitted the command `cd` changes the current directory to user's home directory. For example,

```
  cd "$MLD/demo/ptcl/sdf/basic" execute "butterfly.pt"
```

will load the same file as the example in the previous section. Again, we have assumed that the environment variable `$MLD` contains the installation directory of MLDesigner. To see what the interpreters current directory is, you can type

```
pwd
```

9.8.4 Dynamically linking new primitives

The interpreter has the ability to extend itself by linking in outside object files. The object files in question must define single primitives, they will have the right format if they are produced from preprocessor input. Unlike using MLDesigner’s graphical interface, the interpreter will not automatically run the preprocessor and compiler. It expects to be given object files that have already been compiled. The syntax is

```
  link <objfile>
```

Building object files for linking into MLDesigner can be tricky since the command line arguments to produce the object file depend on the operating system, the compiler, and whether or not shared libraries are used. `$PTOLEMY/mk/userstars.mk` includes rules to build the proper object file for a primitive.

It is also possible to link in several object files at once, or pull in functions from libraries by use of the `multilink` command. The syntax is

```
  multilink <opt1> <opt2> <opt3> ...
```

where the options may be the names of object files or linker options such as `-L` or `-l` switches, etc. These arguments are supplied to the Unix linker along with whatever options are needed to completely specify the incremental link.

When the above linker commands are used, the linked code has temporary status. Symbols for it are not entered into the symbol table, meaning that the code cannot be linked against by future incremental links, and it can be replaced, for example, an error in the loaded modules could be corrected and the `link` or `multilink` command could be repeated. There is an alternative linking command that specifies that the new code is to be considered permanent. It causes a new symbol table to be produced for use in future links. See the Ptolemy language keyword `derivedfrom` item in the MLDesigner Programming Guide on page 13-10 for more information. Such code cannot be replaced, but it can be linked against by future incremental `link` commands. The syntax is
permlink <opt1> <opt2> <opt3> ...

where the options are the same as for the multilink command.

### 9.8.5 Top-level blocks

The command

```tcl
topblocks [<name>]
```

returns the list of top-level blocks in the named block, or in the current module or system, if the argument is omitted.

### 9.8.6 Examining parameters

The `paramvalue` command takes the form

```tcl
paramvalue <block> <parameter> [<current|initial>]
```

and returns the current value of the parameter within the block. The command takes an optional third argument, which may be either `current` to specify that the current value should be returned (the default), or `initial` to specify that the initial value (the parameter value) should be returned.

### 9.8.7 Quitting the Interpreter

The `exit` command exits the interpreter. The syntax is

```tcl
exit
```

### 9.8.8 Getting help

The `help` command implements a simple help system describing the commands available and their syntax. It does not provide help for the standard Tcl functions. The syntax is

```tcl
help [<topic>]
```

or

```tcl
help ?
```

for a list of topics. If the argument is omitted, a short "help on help" is printed.
9.8.9 Registering actions

It is possible to associate a Tcl action with the firing of any primitive. The `registerAction` command does this. The syntax is

\[
\text{registerAction } <\text{pre}|\text{post}> \text{ command}
\]

The first argument specifies whether the action should occur before or after the firing of a primitive. The second argument is a string giving the first part of a tcl command. Before this command is invoked, the name of the primitive that triggered the action will be appended as an argument. For example:

\[
\text{registerAction pre puts}
\]

will result in the name of a primitive being printed on the standard output before it is fired. A typical action resulting from this command would be

\[
\text{puts system_name.module_name.primitive_name}
\]

The value returned by `registerAction` is an action handle, which must be used to cancel the action using `cancelAction`. The syntax is:

\[
\text{set action_handle [registerAction pre tcl_command]}
\]
\[
\text{cancelAction action_handle}
\]

9.9 The Interface to MATLAB and MATHEMATICA

`PTCL` can control MATLAB [HL96] and MATHEMATICA [Wol91] processes by means of the `matlab` and `mathematica` commands. The commands have a similar syntax:

\[
\text{matlab } <\text{command}> [<\text{arg1}>] [<\text{arg2}>]
\]
\[
\text{mathematica } <\text{command}> [<\text{arg1}>] [<\text{arg2}>]
\]

The `matlab` command controls the interaction with a shared MATLAB process. The possible commands and arguments are summarized in table 9.2.

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>end</td>
<td></td>
<td>terminate a session with MATLAB</td>
</tr>
<tr>
<td>eval</td>
<td>script</td>
<td>evaluate a MATLAB script and print the result</td>
</tr>
<tr>
<td>get</td>
<td>name script</td>
<td>evaluate a MATLAB script and get the named MATLAB matrix as Tcl lists of numbers</td>
</tr>
<tr>
<td>getpairs</td>
<td>name script</td>
<td>evaluate a MATLAB script and get the named MATLAB matrix as ordered pairs of numbers</td>
</tr>
<tr>
<td>send</td>
<td>script</td>
<td>evaluate a MATLAB script and suppress the output</td>
</tr>
<tr>
<td>set</td>
<td>name rows cols real imag</td>
<td>set the named MATLAB matrix with real and imaginary values</td>
</tr>
</tbody>
</table>
### Table 9.2: Commands for the MATLAB interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td></td>
<td>start a new MATLAB session</td>
</tr>
<tr>
<td>status</td>
<td></td>
<td>return the status of the Tcl/MATLAB connection (0 means connected, -1 means not initialized, and 1 means error)</td>
</tr>
<tr>
<td>unset</td>
<td>name</td>
<td>unset the named MATLAB matrix</td>
</tr>
</tbody>
</table>

The **mathematica** command controls the interaction with a shared MATHEMATICA process. The possible commands and arguments are summarized in table 9.3.

### Table 9.3: Commands for the MATHEMATICA interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>end</td>
<td></td>
<td>terminate a session with MATHEMATICA</td>
</tr>
<tr>
<td>eval</td>
<td>script</td>
<td>evaluate a MATHEMATICA script and print the result</td>
</tr>
<tr>
<td>get</td>
<td>name script</td>
<td>evaluate a MATHEMATICA script and get the named MATHEMATICA variable as a Tcl string</td>
</tr>
<tr>
<td>send</td>
<td>script</td>
<td>evaluate a MATHEMATICA script and suppress the output</td>
</tr>
<tr>
<td>start</td>
<td></td>
<td>start a new MATHEMATICA session</td>
</tr>
<tr>
<td>status</td>
<td></td>
<td>return the status of the Tcl/MATHEMATICA connection (0 means connected, -1 means not initialized, and 1 means error)</td>
</tr>
</tbody>
</table>

To initiate a connection to a MATLAB and MATHEMATICA process, use

```bash
matlab start
mathematica start
```

To generate a simple plot of a straight line in MATLAB and MATHEMATICA, use

```bash
matlab send { plot([0 1 2 3])}
mathematica send { Plot[x, {x, 0, 3} ] }
```

The `plsend` command suppresses the output normally returned by interacting with the program using the command interface. The `eval` command, on the other hand, returns the dialog with the console interface:

```bash
mathematica eval { Plot[x, {x, 0, 3} ] }
```

-Graphics-
To terminate the connection, use

matlab end
mathematica end

One can work with matrices as Tcl lists or in MATLAB format. To create a new MATLAB matrix \( \mathbf{x} \) that has two rows and three columns, type:

```
matlab set x 2 3 "1 2 3 4 5 6" "1 1 1 1 1 1"
```

We can retrieve this MATLAB matrix in the same format:

```
matlab get x
2 3 {1.0 2.0 3.0 4.0 5.0 6.0} {1.0 1.0 1.0 1.0 1.0 1.0}
```

We can also retrieve the matrix elements as a Tcl list of complex numbers in an ordered-pair format:

```
matlab getpairs x
(1.0,1.0) (2.0,1.0) (3.0,1.0) (4.0,1.0) (5.0,1.0) (6.0,1.0)
```

Now, matrices can be manipulated in both Tcl and MATLAB. For example, the following code creates a Tcl list and sends it to MATLAB as a 2x2 matrix, calculates the inverse in MATLAB and retrieves it back to Tcl as a list and/or pairs. Note that string `prompt` in the code is only a placeholder that will be replaced with current working directory name in your MLDesigner console window.

```
prompt> matlab start
prompt> set a 1
1
prompt> set b 2
2
prompt> set c 3
3
prompt> set d 4
4
prompt> set e [expr "\{a b c d\}"]
1 2 3 4
prompt> set f [expr "\{a b c d\}"]
1 2 3 4
prompt> matlab set matrix $b $b $e $f
prompt> matlab eval {matrix(1,1)}
>>
ans =

1.0000 + 1.0000i
```
prompt> set inv_matrix [matlab get inverse {inverse = inv(matrix)}]
2 2 {-1.0 0.5 0.75 -0.25} {1.0 -0.5 -0.75 0.25}
prompt> set inv_matrix [matlab getpairs inverse {inverse = inv(matrix)}]
(-1.0,1.0) (0.5,-0.5) (0.75,-0.75) (-0.25,0.25)
prompt> set new $inv_matrix
(-1.0,1.0) (0.5,-0.5) (0.75,-0.75) (-0.25,0.25)
prompt> lindex $new 0
(-1.0,1.0)
prompt> matlab unset matrix
prompt> matlab eval {matrix(1,1)}
prompt> matlab end

For other examples of the use of the \texttt{matlab} and \texttt{mathematica} PTCL commands, see sec. 3.7.8 on page 3-26.

\textbf{Limitations of the Interpreter}

There should be many more commands returning information on the simulation, to permit better exploitation of the full power of the Tcl language.

\section{9.10 Definition of Shared Elements}

With MLDesigner you can use a number of so-called shared elements to model the functionality of a component (see ch. 10). Such elements are called shared elements, since they are used to share information without exchanging data. For that purpose they can be linked over different model hierarchy levels. Linking means that different model components use the same model element to manipulate the data. Shared elements are

- memories
- events, and
- resources.

\subsection{9.10.1 Defining Memories}

The command

\begin{verbatim}
newmemory <name> <scope> <type> <value>
\end{verbatim}

adds a memory of given type to the current module. The type can be any data structure derived from \texttt{Root} excluding Ptolemy base types. The scope defines whether the memory is \texttt{internal} or \texttt{external}. The memory may have an initial value. To set the value of a memory, issue the command:

\begin{verbatim}
setmemory <block> <name> <value> [link]
\end{verbatim}
Using this command, you change the initial value of the external memory `name` of the block `block` within the current module to the new value `value`. In this case you can specify the link `link` as the name of a memory of same or derived data structure type to which the memory is linked.

The command

```
setmemory this <name> <value>
```

changes the initial value of the memory `name` of the current module.

### 9.10.2 Defining Events

The command

```
newevent <name> <scope> <type> <value>
```

adds an event of given type to the current module. The type can be any data structure derived from `Root` excluding Ptolemy base types. The scope defines whether the event is `internal` or `external`. The event can have a initial value. To change the value of an event, issue the command:

```
setevent <block> <name> <value> [link]
```

Using this command, you change the initial value of the external event `name` of the block `block` within the current module to the new value `value`. The command

```
setevent this <name> <value>
```

changes the initial value of the event `name` of the current module.

### 9.10.3 Defining Resources

The command

```
newquantity <name> <scope> <indexed|non-indexed>
```

adds a new quantity resource to the current module. For quantity resources, you have to define the addressing mode which can be `indexed` or `non-indexed`. The scope defines whether the quantity resource is `internal` or `external`. Using command

```
setquantity <block> <name> <dimension> <capacity> <occupancy> <blocking> <discipline> <reject> <fit>
```

you can change the properties of the external quantity resource `name` of block `block`. You can use `this` as block name to change the properties of the quantity resource `name` of the current module. Using

```
setquantity <block> <name> <link>
```

you can link the quantity resource `name` of block `block` to a quantity resource of the current module with same name.

The command

```
setquantity <block> <name> <link>
```
newserver <name> <scope>

adds a new server resource to the current module. The scope defines whether the server resource is internal or external.

Using command

setserver <block> <name> <dimension> <servers> <rate>
  <mechanism> <occupancy> <overhead> <preempt>
  <discipline> <reject>

you can change the properties of the external server resource name of block block. You can use this as block name to change the properties of the server resource name of the current module.

Using command

setquantity <block> <name> <link>

you can link the server resource name of block block to a server resource of the current module with same name.

9.11 Definition of Data Structure Types

MLDesigner supports the definition of complex data types. The difference between MLDesigner data types and the original base types is that you can define arbitrary structure data types as well as certain special data types, such as enumeration, that were not defined by Ptolemy. For more detailed information about the structure and the programming of MLDesigner data types, refer to ch. 14.

NOTE: To distinguish between Ptolemy original types and MLDesigner data types, the latter are referred to as “data structures” throughout this document.

9.11.1 Defining Composite Data Structures

Use the command

newds <ds_name> <ds_base>

to create a new data structure of type name derived from the data structure ds_base. Parameter ds_name specifies the type name as well the library in which the new data structure is stored. Library and data type name are separated by a colon. If the library name is omitted, the new data structure is created in the library of the parent data structure. For example, the command

newds MyLib:MyPacket Root.Packet

creates a data structure in MyLib that is derived from Root.Packet. The unique name of this data structure is then MyLib:Root.Packet.MyPacket.

To create a new member of a data structure, use command
9.11 Definition of Data Structure Types

newdsmember <ds_name> <member_name> <member_type> [default] [subrange]

Using this command creates a new member with name member_name of type member_type in the data structure given by ds_name. It is possible to specify a default value. If the data structure member is of numerical type, a subrange of values can also be defined with open and closed intervals. The complete syntax of this command is

newdsmember <ds_name> <member_name> <member_type>
newdsmember <ds_name> <member_name> <member_type> <default>
newdsmember <ds_name> <member_name> <member_type> <default> 
'['or'(' <min>,<max> ')'or']'

For example

newdsmember MyLib:Root.Packet.MyPacket Byte1 Root.Integer 0 [0,256)

creates a new member Byte1 in data structure MyLib:Root.Packet.MyPacket which can have values between 0 and 255. (Note the '[' symbol includes the 0 and the ']' symbol excludes 256).

9.11.2 Defining Enumerations

You can use the command

newenum <enum_name> <enum_base>

to create a new enumeration type with name name that is derived from the enumeration type enum_base. Parameter enum_name specifies the enumeration name as well the library in which the new enumeration type is stored. Library and type name are separated by a colon. If the library name is omitted, the new enumeration type is stored in the library of parent enumeration type. For example

newds MyLib:MyEnum Root.ENUM

creates an enumeration in MyEnum that is derived from Root.ENUM. The unique name of this enumeration type then is MyLib:Root.ENUM.MyEnum.

To create a new member of a data structure, use command

newenummember <enum_name> <member_value> [member_index]

Using this command, you can create a new enumeration value with string value member_value and an according index member_index. If member_index is omitted, the next available index is used. For example

newenummember MyLib:Root.ENUM.MyEnum Member1 0

creates a new enumeration value for MyLib:Root.ENUM.MyEnum with value Member1 and member index 0.
9.11.3 Handling Data Structures

Using command

\[ \text{delds } \text{<ds\_name>} \]

deletes the data structure or enumeration type with unique name \text{ds\_name}. A unique name consists of the library name and the full type name separated by a colon. For example

\[ \text{delds MyLib:Root.ENUM.MyEnum} \]

deletes the \text{MyEnum} enumeration type in library \text{MyLib}. Using \* for parameter \text{ds\_name}, you can delete all data structures and enumeration types that were created by you.

\[ \text{delds } \ast \]

You can use the command

\[ \text{printdsnames} \]

to list the names of all existing data structure types including data structure types that are defined by the standard type library. Use the command

\[ \text{printds } \text{<ds\_name>} \]

to print out the default value of a data structure given by the unique name parameter \text{ds\_name}.

9.12 A Wormhole Example

Here is an example of a simulation that contains both an SDF part and a DE part. In this example, a Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples. This demo shows how easy it is to use the SDF primitives to perform computation on DE particles. The overall delay of the SDF wormhole is zero, so the result is simply Poisson arrivals of Gaussian noise samples.

A wormhole has an outer domain and an inner domain. The outer domain is determined by the current domain at the time you start the \text{defmodule} command to create the wormhole. The inner domain is determined by the \text{domain} command that appears inside the module definition.

\[ \text{proc trysetparam } \{ \text{block param value} \} \{ \]
\[ \text{if [catch \{paramvalue } \text{block } \text{param initial} \text{]} err] then} \]
\[ \{ \]
\[ \text{puts stdout "Warning: $err\n"} \]
\[ \text{else} \]
\[ \{ \]
\[ \text{setparam } \text{block } \text{param } \text{value} \]
\[ \} \]
\[ \text{proc trytargetparam } \{ \text{param value} \} \]
```haskell
9.12 A Wormhole Example

```{ catch {targetparam $param $value} err} then
    puts stdout "Warning: $err\n"
} else
    targetparam $param $value

reset
domain DE
defmodule wormGuts {
    domain SDF

    # definition of model parameters
    newparam stopTime float {$stopTime}
    newparam curIter int {$curIter}
    newparam absIter int {$absIter}
    newparam absSimul int {$absSimul}

    # definition of model memories

    # definition of model events

    # definition of model resources

    # definition of instances and their properties
    instance Add.input=2#1 Add
    numports Add.input=2#1 input 2
    instance Random#1 Random
trysetparam Random#1 {Distribution} {2}
trysetparam Random#1 {MinOrMeanOrTrials} {0}
trysetparam Random#1 {MaxOrVarianceOrProbability} {1}
trysetparam Random#1 {Seed} {-1}

    # define the connections
    alias out Add.input=2#1 output
    alias in Add.input=2#1 input#1
    connect Random#1 Output Add.input=2#1 input#2

    #create sources and sinks for autoterminated ports
    )
defsystem worm
domain DE
# define the target and set the target parameters
target default-DE
trytargetparam timeScale {1.0}
trytargetparam syncMode {YES}
trytargetparam usedScheduler {0}

# definition of model parameters
newparam {GlobalSeed} {int} {1234567890}
newparam {PTCLScript} {string} {}
newparam {RunLength} {int} {40}

# definition of model memories

# definition of model events

# definition of model resources

# definition of instances and their properties
instance wormGuts#1 wormGuts SDF
instance Poisson#1 Poisson
trysetparam Poisson#1 {meanTime} {1.0}
trysetparam Poisson#1 {magnitude} {0.0}
instance XMgraph.input=2#1 XMgraph
numports XMgraph.input=2#1 input 2
trysetparam XMgraph.input=2#1 {title} {Noisy Poisson Process}
trysetparam XMgraph.input=2#1 {saveFile} {}
trysetparam XMgraph.input=2#1 {options}
   { -P -o original -1 noisy =800x300+0+0}
trysetparam XMgraph.input=2#1 {Cumulation} {0}

# define the connections
node node1
nodeconnect Poisson#1 output node1
nodeconnect wormGuts#1 in node1
nodeconnect XMgraph.input=2#1 input#1 node1
connect wormGuts#1 out XMgraph.input=2#1 input#2

# create sources and sinks for autotermiated ports

# execution of iterations
if {{info exists parameterfile] && \
   [info exists parameterset ] && \
   [info exists parameteriter]} {
simulate $parameterfile $parameterset $parameteriter
} else {
    puts {Cannot execute simulation without}
    puts {specification of the parameter file. }
    puts {One possible reason is that you tried }
    puts {to simulate the system using "source" }
    puts {command. Use "execute" command instead }
    puts {to run simulations, see "help execute" }
}
delsystem worm

The result of simulation is shown in fig. 9.2. Figure 9.1 shows the model representation within the MLDesigner GUI.

![Noisy Poisson process](image1)

(a) worm system

![wormGuts module](image2)

(b) wormGuts module

Figure 9.1: Model of the the wormhole example worm

![Simulation result](image3)

Figure 9.2: Simulation result of the wormhole example
9.13 PTCL as Simulation Control Language

9.13.1 Creation of PTCL Scripts

You can use MLDesigner to create an appropriate PTCL script that describes a system model using commands. For example, if you would like to create a PTCL file that constructs the system model sinMod, see fig. 9.3, you must do the following:

- **select Library View** in the MLDesigner tree view
- **select system item** MLD Libraries/DEMO/SDF DEMO/Basic/sinMod,
- **Save this system to** $MLD_USER We first want to export some instance parameters to the system level so that these parameters can be altered in the sinMod.param file. Click on the singen#1 model instance.
- **The three Instance Properties** RunLength, frequency and phase.in_radians must be exported to the system level. In the Instance Properties window choose Export from the context menu.
- **Click the Switch to Simulation Mode** icon on the toolbar.
- **Click and hold the Generate Extern** icon and select Generate PTcl from the expanded menu.
- **Enter a filename for the PTCL script file in the Save As dialog.**
- **Click OK to generate the PTCL script file**

MLDesigner creates the following PTCL file with a .ptcl extension as well as a .param file with the parameters of the System.

```tcl
# This file was generated by MLDesigner version 2.3.r03

proc trysetparam {block param value} {
    if [catch {paramvalue $block $param initial} err] then {
        puts stdout "Warning: $err\n"
    } else {
        setparam $block $param $value
    }
}
```

Figure 9.3: Model of the example sinMod

(a) sinMod system (b) modulator module
proc trytargetparam {param value} {
    if [catch {targetparam $param $value} err] then {
        puts stdout "Warning: $err\n"
    } else {
        targetparam $param $value
    }
}

reset
domain SDF

defmodule singen {
    domain SDF

    # definition of model parameters
    newparam stopTime float {$stopTime}
    newparam curIter int {$curIter}
    newparam absIter int {$absIter}
    newparam absSimul int {$absSimul}
    newparam {sample_rate} {float} {2*PI}
    newparam {frequency} {float} {PI/50}
    newparam {phase_in_radians} {float} {0.0}

    # definition of model memories
    # definition of model events
    # definition of model resources

    # definition of instances and their properties
    instance Sin#1 Sin
    instance Ramp#1 Ramp
    trysetparam Ramp#1 {step} {2*PI*frequency/sample_rate}
    trysetparam Ramp#1 {value} {phase_in_radians}

    # define the connections
    connect Ramp#1 output Sin#1 input
    alias out Sin#1 output

    # create sources and sinks for autotermminated ports
}

defmodule modulator {
domain SDF

# definition of model parameters
newparam stopTime float {$stopTime}
newparam curIter int {$curIter}
newparam absIter int {$absIter}
newparam absSimul int {$absSimul}
newparam {freq} {string} {0.062832}

# definition of model memories

# definition of model events

# definition of model resources

# definition of instances and their properties
instance Mpy.input=2#1 Mpy
numports Mpy.input=2#1 input 2
instance singen#1 singen
trysetparam singen#1 {sample_rate} {2*PI}
trysetparam singen#1 {frequency} {$freq}
trysetparam singen#1 {phase_in_radians} {0.0}

# define the connections
alias in Mpy.input=2#1 input#1
alias out Mpy.input=2#1 output
connect singen#1 out Mpy.input=2#1 input#2

# create sources and sinks for autoterminated ports
}

defsystem sinMod
domain SDF
# define the target and set the target parameters
target default-SDF
trytargetparam logFile {}
trytargetparam loopScheduler {DEF #choices: DEF, CLUST,ACYLOOP}
trytargetparam schedulePeriod {0.0}

# definition of model parameters
newparam {GlobalSeed} {int} {1234567890}
newparam {PTclScript} {string} {}
newparam {RunLength} {int} {400}

# definition of model memories
# definition of model events

# definition of model resources

# definition of instances and their properties
instance modulator#1 modulator
trysetparam modulator#1 {freq} {0.2*PI}
instance singen#1 singen
trysetparam singen#1 {sample_rate} {2*PI}
trysetparam singen#1 {frequency} {PI/100}
trysetparam singen#1 {phase_in_radians} {0.0}
instance XMgraph.input=1#1 XMgraph
numports XMgraph.input=1#1 input 1
trysetparam XMgraph.input=1#1 {title} {A modulator demo}
trysetparam XMgraph.input=1#1 {saveFile} {}
trysetparam XMgraph.input=1#1 {options} {=800x400+0+0 -0 x}
trysetparam XMgraph.input=1#1 {ignore} {0}
trysetparam XMgraph.input=1#1 {xUnits} {1.0}
trysetparam XMgraph.input=1#1 {xInit} {0.0}
trysetparam XMgraph.input=1#1 {EndCondition} {FALSE}
trysetparam XMgraph.input=1#1 {NumberOfItems} {1}

# define the connections
connect singen#1 out modulator#1 in
connect modulator#1 out XMgraph.input=1#1 input#1

#create sources and sinks for autoterminated ports

# execution of iterations
if {([info exists parameterfile] && 
  [info exists parameterset ] && 
  [info exists parameteriter])} {
  simulate $parameterfile $parameterset $parameteriter
} else {
  puts {Cannot execute simulation without}
  puts {specification of the parameter}
  puts {file. One of possible reasons is}
  puts {that you tried to simulate the}
  puts {system using "source" command.}
  puts {Use the "execute" command instead to}
  puts {run simulations, see "help execute"}
}
delsystem sinMod
This is a PTCL representation of the MLDesigner sinMod model.

### 9.13.2 Execute the Simulation

You can use the MLDesigner console window to execute the PTCL script using the command

```
execute <directory>/sinMod.ptcl
```

where directory specifies the directory where you stored the sinMod PTCL file. If this is a subdirectory, enter the full path to your file.

Another option is to run the simulation using the built-in ptclsh by entering the following command in the shell.

```
$MLD/ptclsh <directory>/sinMod.ptcl
```

An MLDesigner PTcl shell makes it possible to execute PTcl simulations independent of the MLDesigner GUI. To execute the simulation type:

```
$MLD/ptclsh -x Systemname.ptcl
```

It is also possible to run PTcl simulations in batch mode using the at command followed by $MLD/ptclsh -x SystemName.ptcl.

You can use the systemlist and cursystem command to get the list of known system models as well as the current system model.

```
prompt> systemlist
main sinMod
prompt> cursystem
sinMod
```

The prompt indicates the current working directory.

To run a simulation of sinMod system model with different waveform frequencies use a text editor to open the file

```
<directory>/sinMod.param
```

Change the file so it looks like:

```
PARAMETER_SET
{
   GlobalSeed : 1234567890
   PTclScript :
   RunLength : 400
   frequency : PI/100
   phase_in_radians: for -1.0 to 1.0 step 0.1
}
```
You can now execute the simulation by entering the following command in the commander window.

```
execute <directory>/<sinMod.ptcl>
```

The result will be 21 frequency modulation output graphs.

**NOTE:** Every PTcl simulation produces a `.param.simulationNumber` file with the simulation parameters for the specific simulation. This file is for record purposes and is not parsed when the simulation is rerun.

The combination of PTCL and the GUI of MLDesigner is very powerful. The examples shown above are just some hints on how they can be used together.
Chapter 10

Shared Model Elements

10.1 Introduction

In this chapter all existing shared elements are explained in detail. For information on how to instantiate and how to use these shared elements, refer to sec. 3.13.

With MLDesigner you can use a number of so-called shared elements to model the functionality of a component. Such elements are called shared elements as they are used to share information without exchanging data. For that purpose they can be linked over different model hierarchy levels. Linking means that different model components use the same model element to manipulate the data. Shared elements are

- Memories
- Events, and
- Resources.

These elements are instantiated via the context menu in the module or system Model Editor Window or by clicking the relevant icons on the toolbar. The advantages of using shared elements is that it is not necessary to connect modules in order to share the information and it is possible to share information over different modeling levels.

10.2 Memories

A memory shared element identifies a location in the memory, used to share information between MLDesigner modules. The information is sent as a Data Structure and consequently every memory has an associated type.

A memory element with an internal scope is the defining memory. This is where the space for the data structure is really allocated. An internal memory can have an initial value which is set to conform with the Data Structure string representation syntax. (See chapter Modeling using Data Structures). This default value is assigned during the simulation initialization phase, before any events are executed.

An external block memory element must be linked to a memory element of the same type in the module which uses the block. External memory shared elements become block memory ele-
ments when a module is used as a block. Memories cannot be External in a system module. For every chain of memories linked together (i.e. sharing the same memory) there is a single memory that has an Internal scope. The value stored in a memory can change during the course of a simulation.

Memories can be manipulated using primitives from MemoryAccess libraries in the DE domain. Generally, memory is modified with ReadMemory and WriteMemory blocks.

Memories having Vector types (VECTOR, IntVector and FloatVector) are initialized by specifying the initial value and the length of the vector.

### 10.2.1 Memory Modules

Several primitives are supplied to handle memories. Most of them are located in MemoryAccess library. There are two types of memories: Local Memory and Shared Memory - memories that are shared by linking them over several levels of modules.

#### 10.2.2 Local Memory

The memory location used by local memory blocks is accessible only through that instance of the block. Local memory primitives have one input which writes to the memory when enabled, and another input as Trigger which causes the memory to be read and its content to be placed on the output port. Primitives that provide access to the local memory are: LocalMemory, LocalMemoryInt and LocalMemoryFloat.

#### 10.2.3 Shared Memory

Shared Memory allows access to a single copy of a data structure (a memory location) from many blocks at arbitrary points in the hierarchical block diagram. Access to the memory value is typically through the ReadMemory primitive. The WriteMemory primitive allows you to change the contents of the memory. MLD provides also modules to access memories that operate specifically on vectors. Some of these modules are MemoryAccessVector, MemorySetVector, MemoryAccessIVector, MemorySetIVector, etc. The WriteMemory primitive also allows access or modification to elements of VECTOR, IntVector and FloatVector base types data structures stored in memory.

Each of these modules has an external memory argument. The way this shared element is linked to others affects the way blocks share memory. To allow several instances of a memory block to share the same memory location, link the memory element of all instances to the same memory. If the memory they are linked to has a scope of Internal, the memory is not visible at higher levels in the block diagram. That means that all Read and Write Memory modules whose memories are explicitly linked together will have access to the same memory. Memory elements are explicitly linked together when you select link for a memory in the property editor.

Exporting memories allows other modules in the hierarchy to access the memory. However, localizing memory allocates sufficient space to hold a data structure of the specified type and limits
the scope of the memory to the block diagram where it is localized. Localizing memories involves providing an initial value which is placed in the memory upon its creation.

Non-local memory is a powerful feature within MLDesigner. Non local memories permit one (or several) modules to modify the state of another module in non-obvious way (by modifying non-local memory). To minimize this effect localize memory at the lowest possible level in the block diagram hierarchy. This restricts the number of modules which can access the non-local memory.

### 10.2.4 Global Memory

MLDesigner provides two modules `GlobalMemoryWrite` and `GlobalMemoryRead` to access the global memories. The only parameter of this modules is a String that indicates the name of the global memory on which the block operates. All global memory modules which reference to the same String name will operate on the same global memory.

**NOTE:** The String parameters need not be linked together; they must, however, be spelt the same.

### 10.3 Events

An *event shared element* is used to indicate that a primitive module can accept asynchronous events. The *Event* is used by the timer modules to specify which timer blocks in a system interact with one another. Events provide links to primitives or modules which manage timing events, such as `StartTimer`, `ServiceTimer` and `CancelTimer`. Events can be used only in timed domains, such as DE domain and in FSM modules.

To add an event to the model use the tool button **Add Event**, the menu item **Edit - Add Event** or the hot key **E**. The following elements can be defined for an event:

- the event’s name
- the scope of the event
- the type of the data sent through an event
- the default value of the data (only for *Internal* events)
- the event’s description.

The scope of an event can be *External* or *Internal*. An *Event* cannot be *External* in a system. Primitives which have an event element (either internal or external) have an event entry point associated with the event. When an event occurs, the code associated with this event is executed. If you want specific code to run when an event occurs, then you have to put the code in the `defevent` item in the primitive source code. A `defevent` item is generated for every event element defined in a primitive:

```plaintext
defevent
{
    name { EventName }
    type { Root }
```
The type of the event is a data structure (for more information about data structures see Data Structures section). You can link an event from a primitive to an event from a module or system only if the type of the first event is the same as (or a base type for) the type of the second event. For an external event you can set the default value, and for an internal event you can set the initial value. The following methods can be used to access the data associated with an event:

```c++
Type* readData () const;
void writeData (Type* pValue);
void writeData (int pValue);
void writeData (float pValue);
```

The events can be used to send data between primitives. In this case you have to set the data using one of the writeData(...) methods before schedule the event. To schedule an event the method

```c++
void emitEvent (double pTime, int pEventID);
```

should be called, where the arguments represent the time at which the event will occur and the event ID, respectively. The event ID is used to identify the event, for example you can cancel an event with a specific ID using the method

```c++
void cancel (int pEventID);
```

which removes, from the queue of events, all entries with this ID. The queue of events contains the scheduled times at which various events occur and the location in the block diagram where the execution resumes once the event occurs. Entries in the queue of events are stored in ascending order by event time, so the first entry in the queue is the next event to occur. Whenever an event is to be processed, the simulation clock is advanced to the time of occurrence of the event. Using the method

```c++
double getResidualTime (int pEventID);
```

you get the difference between the time at which the event with this ID should occur and the current time of the simulation. If there is no entry in the queue of events with this ID, the method returns -1.

## 10.4 Resources

### 10.4.1 Introduction

A Resource can be used to simulate an item that is shared, such as main memory in a computer or the processing power of a CPU. In MLDesigner, resources are divided into Quantity Resources and Server Resources.
10.4 Resources

10.4.1 CPU Demo

An excellent example of how Resources work in MLDesigner can be found in $MLD/Examples/CPU Demo and is called CpuSystem. In this example packets are competing for both Memory and CPU resources. A packet must first obtain Memory, before it goes to the CPU module where it is processed. After being processed, the packet releases its memory, which can be allocated to pending packets. The Memory is modeled with a Quantity Resource, called "MemoryResource”. The CPU is modeled with a Server Resource, called "CPUResource”.

The example shown in fig. 10.1 shows a linked resource over three levels in a hierarchical model. See table 10.1 and fig. 10.1.

<table>
<thead>
<tr>
<th>resource Name</th>
<th>Primitive/Block</th>
<th>Linked To</th>
<th>scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>Allocate</td>
<td>Resource</td>
<td>External</td>
</tr>
<tr>
<td>resource</td>
<td>Allocate#1</td>
<td>CpuMemory</td>
<td>Internal</td>
</tr>
<tr>
<td>Resource</td>
<td>AllocateBasic#1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: The chain of the linked resources in CPU Demo

The Resource of the Allocate#1 primitive named resource with its scope set to external. When the model instance Allocate#1 is selected, the Instance Properties editor is active indicating a link to the resource named Resource of the AllocateBasic#1 module. This resource also has its scope set to External. When the AllocateBasic#1 instance is selected in the system, the Instance Properties editor shows the resource named Resource is linked to the internal resource named MemoryResource. In this case the shared model element was instantiated in each level of the hierarchy.

A quicker way to achieve the same result would be to Export the resource to the next level of the hierarchy (from Version 2.4 upwards). To export a resource select the instantiated model instance whose resource needs to be shared with other levels of the system or module and activate the context menu over the resource property in the Instance Properties editor. Choose Export or Export As if a more descriptive name is required for the resource.

10.4.1.2 Brief Explanation

The packet that leaves the PacketSource has its memory and CPU requirements fields set. Then, the packet enters the AllocateBasic block to obtain the memory requirement (packet length). When the required memory is allocated, the packet moves to the CPU block. The packet is processed by the CPU block based on the priority and CPU requirement. Preemptive is allowed with a switching overhead of 0.1. There are 10 servers active in the CPU. The mode of operation is based on dedicated server.

Once the packet processing is completed by the CPU block, the packet invokes FreeBasic block to return the allocated memory back to the memory pool.
Figure 10.1: CPU Demo Showing Linking of Resource Elements
10.4 Resources

StatisticReporter collects and plots various different statistics for the system.

10.4.1.3 Defining Resource Elements

To add a resource to the model use: the tool button Add Resource, the menu item Edit - Add Resource or the hot key R. Set the field Scope to Internal. In the Resource Properties window. The following elements can be defined for a resource:

- the resource’s name
- the type of the resource (Quantity or Server)
- the scope of the resource
- the resource’s description
- the attributes of the resource (see table 10.2 and table 10.3).

Resource attributes are settings which affect the overall operation of the resource. They are specified for the resource whose scope is set to Internal. A Resource cannot be External in a System. For every chain of Resources linked together (i.e. sharing the same information) there is a single memory that has an Internal scope.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Dimensions</td>
<td>1</td>
</tr>
<tr>
<td>Initial Capacity</td>
<td>1</td>
</tr>
<tr>
<td>MaximumQueueOccupancy</td>
<td>100</td>
</tr>
<tr>
<td>Blocking Mechanism</td>
<td>Wait_for_Resource</td>
</tr>
<tr>
<td>Queue Discipline</td>
<td>First_In_First_Out</td>
</tr>
<tr>
<td>Queue Reject Mechanism</td>
<td>Incoming_DS_Rejected</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>NonIndexed</td>
</tr>
<tr>
<td>Addressed Fit Policy</td>
<td>FirstFit</td>
</tr>
</tbody>
</table>

Table 10.2: Quantity Resource Attributes

Some resource attributes can be varied during simulation. For a Quantity Resource, the number of resource units can be changed. For a Server Resource, the processing rate multiplier (the processing power of each processor) can be modified during simulation. Special blocks allow the capacity of a Quantity Resource and the processing rate multiplier of a Server Resource to be changed during execution of a model. The resource attributes for these two items specify the capacity and service rate multiplier at the beginning of execution. Resources can be used only in timed domains.

The Quantity Resource represents elements which must be possessed by a transaction (a data structure). After a transaction receives resource units, it continues through the model, hold-
### 10.4.2 Quantity Resources

A Quantity Resource represents discrete elements which are stored in a resource pool when not in use, and are allocated to MLDesigner data structures as they move through a system. Typically, a data structure will enter an Allocate block to request resource units. Any quantity of units can be requested. The Allocate block provides queuing if the resource units cannot be allocated immediately. When the resource units can be allocated, the data structure exits the Allocate block and continues its journey through the model. At some later point in the processing of the data structure (and usually at a later time), the resource units may be returned to the resource pool for allocation to other requests.

When using non-addressed resources, units can be identical and indistinguishable, in which case the resource represents a token pool. On the other hand, the addressed resources are distinguish-
able and each has an integer address. Requests for an addressed Quantity Resource require a contiguous block of units, which is allocated using either a first-fit or best-fit policy.

The two fundamental blocks for manipulation of quantity resources are the Allocate and Free blocks, shown in fig. 10.2. The Allocate block is used to make a request for resource units. When the units are available, the output ports on the right side of the Allocate block are enabled to indicate that the resource units have been granted. The Free block is used to release resource units when they are no longer needed.

![Allocate and Free blocks](image)

Figure 10.2: Allocate and Free blocks

### 10.4.2.1 General operation

A quantity resource begins a simulation with an initial capacity, the number of resource units in the pool at the beginning of the simulation. The initial capacity is specified with the Initial Capacity resource attribute. During simulation, requests for quantities of the resource are made by enabling all of the inputs of an Allocate block. Input values, which includes one data structure of any type, are collectively called the transaction. The other inputs specify the attributes of the transaction. One attribute of the transaction is the number of units of the resource requested. If there is an unused quantity of the resource greater than or equal to the number of units requested and there are no transactions with a higher priority waiting in the queue, that number of units of the resource will be removed from the pool and granted to the transaction.

Any time resource units are granted to a transaction, the outputs of the Allocate block where the request was made are enabled. One of the outputs is the data structure which is associated with the transaction. The other outputs are the number of units granted and the starting index if the resource is addressed. Resource units can then be held for an arbitrary amount of time (as the data structure is delayed by other blocks in your model). The resource units can be freed by enabling the inputs of a Free block. One of the inputs specifies the number of units of the resource to free. Normally, you will need a field in your data structure to store the number of resource units held by each resource that can be possessed. If you use addressed resources, you may also need a field to store the starting index.

### 10.4.2.2 Queuing, blocking and rejecting

There are a few options if the requested quantity of the resource cannot be allocated immediately when the request is made. The transaction can be put into a queue to wait for the resource according to a specified queuing discipline (either FIFO or LIFO), it can be sent out the reject output without seizing any resource units, or it can take the available resource units. The first possibility is called Wait_for_Resource, the second Exit_without_any_Resource_Units, the third
Shared Model Elements

Seize_Available_Units. These are chosen with the Blocking Mechanism resource attribute, which has these three possible values. The default value is Wait_for_Resource.

If the Blocking Mechanism is Seize_Available_Units, a transaction seizes its requested number of units or the amount in the pool, whichever is smaller. The Units Granted output can (and should) be used to determine the number of resource units that were seized. If the number of units in the pool is zero and the blocking mechanism is Seize_Available_Units, the transaction is instead rejected, and the arbitrary data structure associated with the transaction is sent out the Reject output.

Setting the Blocking Mechanism to Exit_without_any_Resource_Units could be used to model "blocked calls cleared", as is the case with the common Erlang B model used in telephony. Setting the blocking mechanism to this value could also be used to divert transactions to a second resource when the first one is not immediately available, rather than having them queue and wait for the first resource to become available.

If the Blocking Mechanism is Wait_for_Resource, a transaction simply waits in the queue according to the specified queuing discipline until the resources become available. If the queue is full and the Queue Reject Mechanism is Incoming_DS_Rejected, then this transaction exits on the Reject output port. Else, if the Queue Reject Mechanism is Lowest_Priority_DS_Rejected and there is another transaction in the queue with lower priority, then the incoming transaction gets to take its place, otherwise the incoming transaction is rejected.

There is a single queue maintained for the entire resource (actually one for each dimension as explained in Multiple Dimensions section), which is ordered first by priority and, within each priority, either FIFO or LIFO, according to the Queue Discipline resource attribute. When a transaction is queued, the Allocate block from which the transaction entered the queue is remembered, so the transaction can be sent out the same block when it is granted resource units or rejected from the queue. Priorities of transactions are greater than or equal to zero, with zero being the highest priority.

10.4.2.3 Addressing mode

A quantity resource has two fundamental modes of operation specified by the Addressing Mechanism resource attribute. The default value is NonIndexed, which makes resource units indistinguishable from one another. Resource units are viewed simply as tokens. The other possible setting for this attribute is Indexed. In this mode, each unit of the resource is assigned an integer index, beginning with zero. The status of each resource unit is maintained and can be either in use (not in the pool) or idle (in the pool). A request for resource units is then a request for a contiguous block of resource units. An integer vector, called the Indexed Usage Vector, is maintained to store the state of each resource unit. In this vector, a 1 indicates that a particular resource unit is in use, while a 0 means that the resource unit is not in use.
10.4.2.4  Fit policy

When the Addressing Mode is Indexed, the Addressed Fit Policy resource attribute controls how the Indexed Usage Vector is searched when a transaction enters an Allocate block. The possible values for this attribute are FirstFit and Best-Fit. When set to the default value, FirstFit, the Allocate module begins searching the Indexed Usage Vector at index zero and stops when it finds a contiguous block of units at least as large as the quantity requested. If the addressed fit policy is BestFit, the indexed usage vector is searched for the smallest unused block of units large enough to satisfy the request.

Setting the addressed fit policy to BestFit may cause longer simulation run times, especially if the number of resource units is large.

If the Addressing Mode resource attribute is NonIndexed, the value of the Addressed Fit Policy resource attribute is not used and has no effect on the operation of the resource.

10.4.2.5  Uses of the indexed addressing mode

The indexed addressing mode is useful in a model where the resource units represent items that are distinguishable. For instance, if the resource represents pages of memory and transactions need contiguous pages of memory, the indexed addressing mode is appropriate.

Another use of a quantity resource with indexed addressing mode would be for allocation of virtual circuit numbers in a network. In this case, requests always ask for a single resource unit, and the index of that unit is the virtual circuit number that can be used.

10.4.2.6  Caveats

When you use the indexed addressing mode, one of the outputs of the Allocate block is the starting index for the resource units. Store this value (usually in a field of your data structure) and pass it to the Free block when you release the resource units. The Free block must be told the number of units to be freed and the starting index of those units if the addressing mechanism is indexed.

MLDesigner does not limit the number of resource units that can be represented with an indexed quantity resource. Remember that the state of every resource unit is kept with an integer vector having one element for each resource unit, so simulating an indexed resource with a large number of resource units takes longer to execute than if the addressing mode is NonIndexed.

10.4.2.7  Modifying capacity during a simulation

If the Addressing Mechanism is NonIndexed, the capacity of the resource may be increased or decreased during the simulation by using the Change Capacity block. If the capacity is increased, new resource units are immediately placed in the resource pool, and the resource queue is checked. If any waiting transactions can be granted their requested number of units, they are immediately given the requested units.
If the capacity is decreased, the resource units are removed from the pool, provided the pool contains at least that number of units. If the pool does not contain the number to be removed, all unused units in the pool are removed, and the remaining units are removed when they become freed. The capacity of the resource cannot be less than zero. If you attempt to set the capacity below zero, the capacity of the resource is not changed, the output of the Change Capacity block is not enabled, but the simulation continues executing.

### 10.4.2.8 Multiple dimensions

One of the resource attributes, Number of Dimensions, allow a single set of resource blocks arguments to represent a collection of identical resources distinguished by an integer index, called the dimension. The Allocate, Free, and other resource models have an input (or parameter) which is the dimension of the resource to be accessed. The dimension is zero-based, with valid values between zero, inclusive, and the number of dimensions attribute of the resource, exclusive.

When you use multiple dimensions, separate queues and resource pools are created for each dimension. The resource attributes are the same for every dimension.

### 10.4.2.9 Quantity resource attributes and possible values

The meaning of each resource attribute and the possible values are listed below.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description and Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Dimensions</td>
<td>The number of dimensions which can be modeled with each grouping of resource blocks and arguments.</td>
</tr>
<tr>
<td>Initial Capacity</td>
<td>The number of resource units in the pool when the simulation starts.</td>
</tr>
<tr>
<td>Blocking Mechanism</td>
<td>The action to be taken if a transaction cannot be granted the requested number of resource units when it enters the Allocate block. This attribute must be set to Wait for Resource for the values of the next three resource attributes to be used (Maximum Occupancy, Queue Discipline and Queue Reject Mechanism). Possible values: Wait_for_Resource, Exit_without_any_Resource_Units, Seize_Available_Units.</td>
</tr>
<tr>
<td>Maximum Occupancy</td>
<td>The maximum number of transactions which can wait in the queue at any one time.</td>
</tr>
<tr>
<td>Queue Discipline</td>
<td>The ordering of transactions in the queue which have the same priority. Possible values: First_In_First_Out, Last_In_First_Out.</td>
</tr>
<tr>
<td>Queue Reject Mechanism</td>
<td>The way a transaction is chosen for rejection when a new transaction would cause the overall occupancy to exceed the Maximum Occupancy. Possible values: Incoming_DS_Rejected, Lowest_Priority_Rejected.</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>This attribute determines whether each resource unit has a unique integer identifier (Indexed) or if resource units are indistinguishable. Possible values: Indexed, NonIndexed.</td>
</tr>
</tbody>
</table>
10.4 Resources

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description and Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressed Fit Policy</td>
<td>The policy by which available resource units are searched for when the Addressing Mode is Indexed. The value of this attribute is ignored if the Addressing Mode is NonIndexed. Possible values: FirstFit, BestFit.</td>
</tr>
</tbody>
</table>

Table 10.4: Quantity Resource Attributes

10.4.3 Server Resources

A Server Resource is logically the combination of two fundamental components: a queue and a bank of one or more servers. A data structure enters a Service block to request processing from the resource. The data structure and several attributes, such as the requested service time, are together called a transaction. A transaction immediately moves to a server if one is idle, or may optionally preempt lower priority transactions. While being processed by a server, transactions accumulate service time and they exit the Service block when they received the required amount of service time. Accumulation of service time is affected by several resource attributes.

If a new transaction cannot immediately move into a server, it is placed in a queue where it waits for a server to become available. The queue is ordered first by priority, then according to the queuing discipline attribute: either first in first out (FIFO) or last in first out (LIFO).

Each server in a Server Resource is capable of distributing its processing power to more than one transaction. Server sharing methods are either processor sharing or round robin. When a server is shared using processor sharing, each transaction receives service at a reduced rate, where the reduction is proportionate to the number of transactions sharing the server. With round robin server sharing, each transaction sharing the server receives the full power of the server for a (usually small) amount of time, its time-slice. Then, the next transaction receives its time-slice, and so on. Each time a transaction receives its time-slice, its remaining time is reduced by that amount. When a transaction has received all of its service time, it exits the Service block.

The primary block for accessing a server resource is the Service block, shown in fig. 10.3. The inputs to this block are the transaction attributes, such as service time needed, priority, and so on. When a transaction completes service, the data structure which entered the arbitrary data structure input is placed on the arbitrary data out output port. There are versions of the Service block which have fewer inputs, where some of the transaction attributes are parameters instead of inputs.

10.4.3.1 General operation

For every resource of type Server Resource and scope Internal, there is one queue and a bank of servers for every dimension of the resource. The number of servers in the server bank is specified by the Number of Servers attribute of the resource.

A transaction is a request for some amount of time in a server. Each transaction has attributes such as its priority, service time, and other attributes needed by the resource. When all inputs of a
Service block are enabled, the combination of all input values makes up a new transaction. One of the inputs (arbitrary data structure) is a data structure of any type that is kept with the transaction. When a transaction has received its requested service time, this data structure is enabled on the output of the Service block where the transaction originated. If for some reason the transaction is rejected from the resource, this data structure is instead placed on the reject output of the block where the transaction originated.

When a new transaction enters the resource, it goes to an idle server or preempts a server currently processing a lower priority transaction. If it cannot do either, it checks to see if there is a transaction of equal or higher priority waiting in the queue. If so, it is enqueued based on its priority. If no equal or higher priority transactions are waiting in the queue, the new transaction sees if it can share a server with other transactions of the same priority. If a server cannot be shared, the transaction is queued.

If a new transaction preempts a server, the transaction(s) that were being processed by that server first attempt to share another server. If another server cannot be found that can be shared, the transaction(s) that were being processed by the preempted server are said to be preempted, and they are handled according to the value of their Preempt Response attribute. Preempted transactions that are to be resumed or restarted move back to the queue and wait until they receive processing from a server. A preempted transaction can also be discarded, in which case its arbitrary data structure is sent out the reject output of the Service block where the transaction originated.

When a transaction completes its service, its associated data structure is sent out the arbitrary data output port at the Service block where the transaction originated. Prior to sending out the data structure, the server that had been processing the transaction, if it was not being shared, checks the queue for pending transactions. If there are any, it begins processing the transaction at the head of the queue, and all others in the queue having the same priority if server sharing is enabled.

10.4.3.2 Queuing

Transactions in the queue are ordered first by priority and then within each priority according to the Queuing Discipline, which is a resource attribute. Any time a transaction is preempted, it is placed in the queue and ordered based on its priority and the time it entered the resource according to the Queuing Discipline. Priorities are greater than or equal to zero, with zero being the highest priority.
10.4 Resources

10.4.3.3 Server Sharing

The Server Mechanism is the resource attribute that determines whether transactions can share a server. It has three possible values: Dedicated Server, Round Robin and Processor Sharing. If this is set to Dedicated Server, each server can process only a single transaction at a time, and server sharing is not enabled.

When the Server Mechanism is set to Round Robin, each server can process several transactions at once, but does so by dedicating service in a round robin fashion. In round robin, each transaction sharing the server gets a time slice of service each time the server processes it. The time slice given to each eligible transaction is an attribute of the transaction, and can vary from one transaction to another.

If the Server Mechanism is set to Processor Sharing, all transactions sharing the server accumulate service time at a rate inversely proportional to the number of transactions sharing the server and proportional to the number of servers which they are sharing. If the time slice quantum is small compared to the service time of transactions and the time slice is the same for every transaction, Processor Sharing is a good approximation to Round Robin.

A server can be shared only by transactions with equal priorities. Also, a number of servers can be shared by a number of transactions, as long as all transactions sharing the servers have equal priority.

10.4.3.4 Processor Sharing

If the Server Sharing resource attribute is set to Processor Sharing, processor sharing begins at the point where the number of transactions in the server bank of equal priority becomes greater than the number of servers available to process that priority.

For example, consider a server resource with two servers. If all transactions that request the resource have the same priority, then when a third transaction enters the resource while two others are being processed, the three transactions then share the two servers. If no other transactions arrive before one of them completes service, the completed transaction exits the resource, and the two remaining transactions no longer have to share the servers because they each have their own. If a fourth transaction had entered the resource before one completed, the two servers would then be shared by four transactions.

When transactions are sharing a server or servers, the rate at which they receive service time is proportional to the number of servers being shared and inversely proportional to the number of transactions sharing the server(s).

Going back to the example in which the resource has two servers, at the time the two servers become shared by three transactions, each transaction is receiving two-thirds the service time it would if the server were not shared. If four transactions are sharing two servers, the accumulation of service time is one-half (2/4) the rate if each transaction had a dedicated server.

If the Context Switching Overhead resource attribute is nonzero, it is applied to each transaction when it begins being processed by a server. There is no extra service time lost due to
processor sharing.

### 10.4.3.5 Round Robin

If the `Server Sharing` resource attribute is set to `Round_Robin`, round robin server sharing begins when the number of transactions in the server bank of equal priority becomes larger than the number of servers available to process that priority. At that time, the new transaction is placed in a round robin wait list (there is a separate list for each priority that is sharing servers round robin). Then, the transactions in shared servers are checked to see if they have received their time slice since they most recently received the server. A transaction that had been processed by a dedicated server that crosses the threshold to round robin sharing is allowed to continue service until its time slice has completed. If it has received more than its time slice of service time since it received the server, it is put at the end of the round robin wait list. Then the transaction at the head of the round robin wait list is moved to that server and remains in the server until it completes its time slice or its service, whichever comes first. This continues for all shared servers. If a new transaction is to share a set of servers which are already shared round robin, the transaction is placed at the end of the round robin wait list.

When a transaction completes its service, the transaction at the head of the round robin wait list is moved to the server just freed by the completed transaction. If no more transactions are in the round robin wait list, the server(s) is no longer shared and crosses the threshold back to dedicated server.

If the `Context Switching Overhead` resource attribute is nonzero, it is applied each time a transaction gets a server. Specifically, the context switching overhead is applied at the beginning of the time slice.

### 10.4.3.6 Preemption

The `Preemption Discipline` resource attribute specifies whether a higher priority transaction can preempt a server processing lower priority transactions. This resource attribute has two possible values: `Allow_Preemption` and `Dont_Preempt`. If preemption is enabled and the `Queue Discipline` resource attribute is set to `Last_In_First_Out` (LIFO), then transactions can preempt other transactions of equal or lower priority. When the `Queue Discipline` is `First_In_First_Out` (FIFO), equal priority transactions cannot be preempted, only strictly lower priorities will be preempted.

If the number of servers is greater than one, the servers are searched to determine which is processing the lowest priority transaction. If there is only one, it is preempted. If there are more than one, one server is preempted. When transactions are preempted, if the `Server Mechanism` is either `Round_Robin` or `Processor_Sharing` and another server is processing the same priority as the server that was preempted, the transactions that were being processed by the preempted server will share the other server(s) processing the same priority.

If preempted transaction(s) cannot share another server, they are dealt with based on their `Preempt Response`. The `Preempt Response` is a transaction attribute that can be `Resume`, `Restart`, etc.
10.4 Resources

or Discard. Transactions with their preempt response set to Resume are placed back in the queue with their remaining service time set to their original service time required minus the service time they accumulated prior to being preempted plus the value of the Resume Overhead transaction attribute. The Resume Overhead specifies the wasted service time when a preempted transaction resumes service. If a preempted transaction’s preempt response attribute is Restart, the transaction is placed back in the queue, and its remaining service time is set equal to the original service time required by the transaction. Finally, if the preempt response is Reject, a rejected transaction’s associated data structure (from the arbitrary data structure input) is sent out the rejected output at the Service block it originally entered.

10.4.3.7 Accumulation of service time

When a transaction enters the resource, a service time is specified for the transaction. This is the amount of time needed in a server given that the transaction is the only transaction in the server and the Service Rate Multiplier (SRM) is 1.0. The Service Rate Multiplier is a resource attribute that specifies the relative rate at which the servers process transactions. A higher SRM causes servers to process transactions faster while a lower value makes a server take a longer amount of time for a transaction to accumulate the same amount of service time. The actual elapsed time in a server (independent of queuing delay) is dependent on the Server Mechanism for the resource. The three possibilities for Server Mechanism and the way service time is computed for each is listed below. Note that the SRM affects all processing times, including time slice, context switching overhead and resume overhead.

Server Mechanism = Dedicated_Server

There is no sharing of servers. Therefore, the elapsed time in the server for a transaction is:

\[
\frac{Service\ Time}{Service\ Rate\ Multiplier}
\]

Server Mechanism = Processor_Sharing

If the Server Mechanism is set to Processor_Sharing, the elapsed time for each eligible transaction in the server is equal to:

\[
\frac{Service\ Time \ast Transactions}{Service\ Rate\ Multiplier \ast Servers}
\]

where Transactions is the number of transactions sharing the server(s) and Servers is the number of servers being shared. Each time the number of transactions sharing a server changes, or the number of servers being shared changes, the accumulated service time is computed for each transaction and then the remaining elapsed time is recomputed.

Server Mechanism = Round_Robin

If the Server Mechanism is set to Round_Robin, a server being shared by several transactions processes them one at a time in succession, giving each its slice of service time before moving to the next. The time period allotted to each transaction is dependent on the time slice
attribute of the transaction, which may vary from one transaction to another. The accumulated service time for a transaction each time it receives the server is equal to its time slice attribute. The elapsed time for this to occur is:

\[
\begin{array}{c|c}
\text{Time Slice} & \text{Service Rate Multiplier} \\
\end{array}
\]

10.4.3.8 Context switching overhead

There is an optional overhead associated with switching from one transaction to another. Context Switching Overhead is a resource attribute that specifies the amount of service time wasted when a server switches from one transaction to another. Except during processor sharing, the elapsed time wasted is the Context Switching Overhead divided by the Service Rate Multiplier. During processor sharing, the Context Switching Overhead is assessed to each transaction at the beginning of its service. The elapsed time for the Context Switching Overhead to occur during processor sharing when \( T \) transactions are sharing \( S \) servers is:

\[
\frac{\text{Context Switching Overhead} \times T}{\text{Service Rate Multiplier} \times S}
\]

The Context Switching Overhead is also applied to each transaction when it begins service after being preempted. If a server begins processing a transaction but is preempted before the context switching overhead has completed, the remaining service time of the transaction remains unchanged.

10.4.3.9 Maximum Occupancy and Queue Reject Mechanism.

The Maximum Occupancy attribute specifies the maximum number of transactions that can be anywhere in the resource. This includes those waiting in the queue, transactions in a server and transactions in a round robin wait list. Whenever accepting a new transaction would cause the occupancy of the resource to exceed this value, either the new transaction or the transaction at the tail of the queue is rejected. The method of determining which transaction is rejected is specified by the Queue Reject Mechanism, which has two possible values: Incoming_DS_Rejected or Lowest_Priority_Rejected. If it is set to Incoming_DS_Rejected, the new transaction is rejected whenever the maximum occupancy would be exceeded. If it is set to Lowest_Priority_Rejected and there is at least one transaction in the wait queue, the new transaction is accepted and processed in the normal manner, and then the transaction at the tail of the queue is removed from the queue and its arbitrary data structure sent out the reject output. If the wait queue is empty, the incoming request is always rejected.

10.4.3.10 Multiple dimensions

One of the resource attributes, Number of Dimensions, allows a single grouping of resource models and resource elements to represent a collection of independent and identical resources which are distinguished by an integer index, called the dimension. Each dimension is isolated from all other dimensions. The dimension values are zero based, with valid values between zero, inclusive, and the number of dimensions attribute of the resource, exclusive.
10.4 Resources

10.4.3.11 Modifying the Service Rate Multiplier during simulation execution

The Service Rate Multiplier (SRM) can be varied during the course of simulation. The Modify SRM block can be used to set the service rate multiplier for a given dimension of the resource. This block is illustrated in fig. 10.4.

![Modify SRM block](image)

Figure 10.4: Modify SRM block

The Modify SRM block has two inputs: the new SRM and the dimension. When both inputs are enabled, this block executes, checks for valid input values and sets the SRM for the given dimension of the resource to the specified value. If there are no errors, the output port is enabled. This block allows you to set the SRM differently for each dimension of the resource.

10.4.3.12 Server resource attributes and possible values

The meaning of each resource attribute and the possible values are listed below.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description and Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Dimensions</td>
<td>The number of dimensions which can be modeled with each grouping of resource models and shared elements.</td>
</tr>
<tr>
<td>Initial Number of Servers</td>
<td>The number of servers in the server bank. This value cannot be changed during simulation.</td>
</tr>
<tr>
<td>Initial Service Rate Multiplier</td>
<td>The Service Rate Multiplier (SRM) at the beginning of the simulation.</td>
</tr>
<tr>
<td>Server Mechanism</td>
<td>The method by which a server can be shared by several transactions of the same priority. Possible values: Dedicated_Server, Round_Robin, Processor_Sharing.</td>
</tr>
<tr>
<td>Maximum Occupancy</td>
<td>The maximum number of transactions which may be in the resource at a single time. This includes transactions which are in the queue, a server, and in a round robin wait list.</td>
</tr>
<tr>
<td>Context Switching Overhead</td>
<td>The service time which is wasted each time a server switches from processing one transaction to another.</td>
</tr>
<tr>
<td>Preempt Discipline</td>
<td>Controls whenever a transaction can preempt another transaction to get a server. Possible values: Dont_Preempt, Allow_Preemption.</td>
</tr>
<tr>
<td>Queue Discipline</td>
<td>The ordering of transactions in the queue which have the same priority. Possible values: First_In_First_Out, Last_In_First_Out.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description and Possible Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Queue Reject Mechanism</td>
<td>The way a transaction is chosen for rejection when a new transaction would cause the overall occupancy to exceed the Maximum Occupancy. Possible values: Incoming_DS_Rejected, Lowest_Priority_Rejected.</td>
</tr>
</tbody>
</table>

Table 10.5: Server Resource Attributes
Chapter 11

Import/Conversion of Models

11.1 Converting OCT Models

11.1.1 Supported Oct types

If you have existing Oct libraries you wish to convert to XML, you can locate them by expanding the Root directory in the Tree View of the MLDesigner GUI.

NOTE: You must have write permission to convert Oct Libraries to XML.

11.1.2 How to start conversion

The converter is developed to do its work recursively making it possible to convert an entire Oct library, module, or system. To do so select the directory you wish to convert in the tree view window and activate the context menu by clicking the right mouse. Select Convert to Xml to begin conversion. This option is only available in the File View.

11.1.3 Estimated time vs. estimated number of models

After selecting Convert to Xml the converter counts all systems, modules and models in the directory you wish to convert. If you select a single primitive, counting and conversion will last one or two seconds. If you chose a system containing a lot of primitives and modules, counting can take a while. Thereafter the conversion starts and a dialog will display the path and filename of the Oct model currently being converted. A progress bar indicates how long the conversion will take.

11.1.4 Models that will be converted

Oct to XML conversions work recursively meaning that all sub-libraries and models that are contained in the chosen Oct library or module will also be converted. If any item in the selected library or module has been converted previously, and therefore has an associated .mml file, the item will not be converted. The context menu option Convert to XML will not be available for models, modules, or libraries with associated .mml files as these have already been converted.
11 Import/Conversion of Models

11.1.5 Converting or not

At any time the conversion can be interrupted by pressing the cancel button. If the converter has already finished counting all models in the library, a skeleton .mml file is created for all models and primitives. In this case you must delete the .mml files of models or primitives before attempting to convert them again. The presence of the .mml file in the library leads the interpreter to believe that the Oct model has already been converted to XML.

11.1.6 Layout of converted models

Coordinates in Oct and Xml

One of the most obvious things you will see after conversion is that your converted model is smaller. The reason for this is the different coordinate system in Xml.

Alignment of ports

With Oct, there is no way to set the alignment of ports. This results in problems with port alignment in the XML model. An algorithm was found that works in most cases but still has some limitations. It is possible that ports of instances may have changed position.

This will happen if your instance, in the original Oct system, was rotated 90 degrees and mirrored. While this does not affect the functionality of the module or system it creates a confusing block diagram. Solving this in larger modules or systems can be time consuming.

11.1.7 Changes

There are some syntax changes for parameters that cannot be evaluated during conversion. This affects especially linked parameters and parameters using tcl expressions.

Linked parameters

Because of differences in syntax it is necessary to redo the links for all instances after conversion to XML. This is because the parameter linked parameter has not been converted to the new syntax for XML. You need to change the parameter to $linked parameter in the Property Editor. An alternative is to select the option link to from the context menu in the Property Editor.

Expressions

The 2nd difference concerning parameter syntax is a change in evaluating tcl expressions and formulas to compute numeric values. These changes are

- The variables get a $ and always loose their curly braces, except in special function calls for matlab.
- The exclamation mark (!) has been replaced by “tcl” e.g., !"expression” has changed to tcl(expression).
11.1 Converting OCT Models

- System wide constants such as PI do not need a $ if they are used in a formula that computes a numeric type value (e.g. float, int). A $ is only necessary to identify such a constant inside a string expression.

Old Syntax:

i) " expr { variable_1 } * PI "
   or
ii) { variable }

New Syntax:

i) tcl( expr\$variable_1 * $PI )
   or
ii) {$variable}

11.1.8 Parameter list

You will find some new parameters in Xml modules and systems, such as new target parameters, and some parameters of primitives will no longer appear in the Property Editor. Parameters that are not editable for example in their .pl files will not be displayed in the property editor.

11.1.9 Inconsistencies in Oct

Oct connections or a net of connections sometimes have inconsistencies, but there is no way to exactly specify where they can be found in your design. In most cases conversion works but you probably cannot open the model. This is most likely a problem with edges of the net object in Oct. During conversion, some of these edges could not be determined and some values in the .mml file probably contain faulty data. In this case you have to redraw the connections in the special Oct model and convert again. Remember to delete the .mml file before converting a second time.

11.1.10 Missing interface facets of modules

It is possible that some of the models you like to convert do not have an interface facet file. You can identify these models by their grey icon in the tree view of MLDesigner. After conversion these models will no longer have a gray icon because missing interface information port definitions of modules is allowed in XML libraries. This, however, means that output ports of a module will be changed to input ports by default. All other parameters stay the same but you must change the port type from input to output or inmulti to outmulti after conversion.

11.1.11 New library structure in MLDesigner

Some modules and primitives in the standard library, such as the DE domain, have been moved to a different location inside this domain, e.g. some number generators, sources and older versions of vector operation primitives. It is possible that you will see red bounding boxes, indicating broken references, if you open an Oct models created with an older version of MLDesigner.
During conversion a mapper file is used to redefine the new location and will correct this in the newly created Xml model. That is why all missing instances in your Oct model can be found in the Xml again.
11.2 Converting BONeS Models

11.2.1 Conversion Conventions

General Conventions

The BONeS Model Conversion Tool in MLDesigner has been designed to convert BONeS Version 4.0 models or later. If you wish to convert models from earlier versions you must first convert them to Version 4.* using BONeS Designer before attempting to convert to MLDesigner.

This chapter contains:

- Conversion conventions
- How to Convert BONeS models to MLDesigner types
- Tips on troubleshooting
- Error messages, their causes and possible solutions
- Two tables containing lists of BONeS primitives with their MLDesigner equivalents. The first table is sorted by category and the second alphabetically.

There are some general conversion conventions to be noted:

- Due to the different naming conventions of MLDesigner and BONeS Designer some changes have to be made to the names of the models: Every character or whitespace which is not a letter or number is replaced by an underscore "_". The original name is stored as the "Logical Name".
- Descriptions are imported without restrictions.
- The logical structure of the BONeS library (grouping) is not imported (see sec. 11.2.5).

11.2.2 Conversion Conventions For Models

General Conventions

A model is converted to a system, a module or a primitive depending on the simulation flag and the chosen implementation. Where the simulation flag is set to System, a system is generated in MLDesigner. There is no implementation allowed for systems so the only way to convert a system is as a model. Where the simulation flag is set to “Component” a module or a primitive is generated depending on the implementation. Where the implementation is set to Standard Primitive, a primitive is generated and where None is set, a Module is generated.

FSMs, input requirements for modules and primitives, symbols, and backgrounds are not imported.

Conversion Conventions For Items Of A Model

- Instances are placed in the same positions as in the model of the BONeS Designer. They get an internal instance name which is generated automatically. The instance labels and the port alignments are set to the values of the BONeS Designer. Some changes are made to the values of the instance parameters. In front of references to other parameters a $n is placed.
The graphical representations of the instances normally differ in size to those of BONEs Designer. This is especially noticeable when an instance in the BONEs Designer model had a different symbol to the normal block. Because of this the relations between ports may loose their orthogonal style.

- Ports are placed at the same positions as in the model of the BONEs Designer. Instance labels, alignments, and auto terminations have the same values as in BONEs Designer. Bidirectional ports are split into one input and one output port. Data flows and deferred properties are not imported.
- Arguments are imported as parameters, memories, events and resources according to their "Class" properties. The data types for TRIGGER, ROOT-OBJECT and COMPOSITE are set to Root. Others data types are set to the equivalent type as in MLDesigner. Scopes and subranges are set to the values as in BONEs Designer. For parameters the scope is stored in the Attributes property (internal = A_NONSETTABLE, external = A_SETTABLE). Grouping elements are not imported.
- Text nodes are placed at the same positions as in the model of the BONEs Designer. The size is not imported.

### 11.2.3 Conversion Conventions For Data Structures

Some changes are made to the parent data structures since some types do not exist in MLDesigner. So the data structures TRIGGER, ROOT-OBJECT and COMPOSITE are mapped to the Root data structure of MLDesigner. The fields are converted to data structure members. The naming conventions are the same as for models. The data types are subject to the conditions of the parent data structures. Scopes and subranges are set to the values as in BONEs Designer. The members are imported as enumeration values. The single values are subject to other naming conditions. Whitespaces are replaced with underscore "_" and any other character which is not a letter or a number is deleted.

### 11.2.4 BONEs Conversion Assistant

To convert a BONEs library (or parts of it) into MLDesigner:

- Select the menu entry Convert BONEs... from the File menu. Click the folder icon to the right of the text field in the Convert Bones Model dialog.
- Select the directory containing the BONEs library you wish to convert, or type the path in the BONEs Library Path field. The directory is scanned and a dialog is displayed showing all convertible models (systems, modules, primitives and data structures). An error will be displayed if an incompatible directory is chosen.
- Next you must determine the MLDesigner library directory where you would like to save the BONEs library. You can either use the default directory which is generated automatically $MLD_USER/LibraryName or choose a new one (either per dialog by clicking on the button to the right of the MLDesigner Library Path text field or typing the path directly into the field).
- Select the models you want to convert by mouse click. The dependencies between the system, models, and data structures are observed. All modules and data structures contained
11.2 Converting BONeS Models

in the selected system are selected with a single mouse-click.

- You can select or deselect components by using the relevant buttons or by using a combination of \textit{(Shift)} and \textit{Mouse-click}.

- The two radio buttons \textit{Existing Models} are set to \textit{Keep} by default. This means that all existing BONeS Modules or Data Structures with the same name as those about to be converted, will not be Overwritten.

- Click the \textit{Convert} button at the bottom of the dialog. A progress bar indicates the approximate duration of the convert process as a percentage. After completing the conversion a warning/error summary is displayed. You can leave this dialog open while checking the results of the conversion.

- You can cancel the conversion at any stage by clicking the \textit{Cancel} button. The conversion process is stopped. All components converted up to this point are saved in the directory you selected.

![Figure 11.1: Convert BONeS Model Dialog](image)

**Figure 11.1: Convert BONeS Model Dialog**

11.2.5 Troubleshooting

- References to other user libraries:
  If the library to be imported uses models or data structures from other user libraries, you will get a "Missing dependency to ..." error in the final warning dialog of the BONeS Converter. One solution would be to merge the libraries in the BONeS Designer as it is not possible to import more than one library at a time.

- Relations between bidirectional ports:
  Bidirectional ports are split to one input and one output port. In the BONeS Designer there is only one connection between bidirectional ports. So the input and the output port are connected together. This leads to completely different simulation results. The Solution is to delete the connections between these ports and connect them again in the appropriate way.
11 Import/Conversion of Models

- Simulations:
  Since only the associated systems and not the simulation itself is imported the system parameters are not set for a simulation. You must set them manually before you can run the simulation. You must also make some changes in order to see the results of the simulation. Because we have no probing mechanism you must connect output models to the ports from which you want to see results. These models you can find in the Sinks or TclTk category of the DE domain of the MLDesigner libraries.

- Primitives:
  For user defined primitives a .pl file is generated. Vectors, base type data structures, strings, ports, and memories are converted to MLDesigner types. Where problems could arise is in primitives containing composite data structures and where resources or events are present.

11.2.6 Error Messages

The following tables show common error messages encountered after conversion of BONeS models and their possible causes with solutions.

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>... cannot contain any formal ports/connections</td>
<td>The conversion utility found model with formal ports or connections that could not be converted for MLDesigner. (e.g. in BONeS Designer it is possible to add ports to a system and save it without verification. After trying to convert such a system you will get this error message).</td>
<td>All components of the model were imported except for the formal ports. To avoid this message you can set the simulation flag to ‘Component’ in BONeS Designer or remove the ports completely and then convert the model again.</td>
</tr>
<tr>
<td>... has an unknown position/label format.</td>
<td>Despite the error message, the model has been converted properly. Only some graphical information was lost. For example, an object was moved to position (0, 0) because the original position could not be read or the instance label was lost. You will see the instance name instead of this.</td>
<td>The system will run properly in spite of this. You can move the model instance or set the label manually.</td>
</tr>
<tr>
<td>Error Dialog</td>
<td>Cause</td>
<td>Solution</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Bi-Directional port ... in ... is split into one input and one output port</td>
<td>This is only a warning message that is self explanatory.</td>
<td>Hand correction is needed to properly convert bi-directional ports from BONeS Designer files. Because MLDesigner does not currently support bi-directional ports, these connections must be converted to two unidirectional port connections. Today this must be done manually; future versions of the conversion utility will do this automatically.</td>
</tr>
<tr>
<td>Component found in ... has no entry in association table of the library</td>
<td>An entry in the association table of the library was not found. This happened while attempting to create an instance and searching for the model entry in the association table of BONeS Designer. This is a very unusual case because every time you add something from another library an entry is created in the association table. You must add the instance manually to the association table, set the argument values and connect the ports or make sure that every model is saved and verified in the BONeS Designer and convert the model again.</td>
<td></td>
</tr>
<tr>
<td>Conflicts in association tables of two libraries...</td>
<td>This message is a warning and rarely leads to problems during conversion. You will get more errors/warnings beside this one. Possible causes for this error message:</td>
<td>You must upgrade all libraries to at least BONeS Designer version 4.0.</td>
</tr>
<tr>
<td></td>
<td>• you are trying to convert more than one library which were created with different versions of BONeS Designer or • one of the converted libraries was created with an older version of BONeS Designer and then touched again with a newer version.</td>
<td></td>
</tr>
</tbody>
</table>
## Import/Conversion of Models

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Could not access ... file...</td>
<td>A file needed for the conversion of a model could not be found. May happen during model conversion from older versions of BONeS Designer, if the file architecture was different to version 4.*.</td>
<td>You must upgrade your libraries to at least BONeS Designer version 4.0.</td>
</tr>
<tr>
<td>Could not add instance of type ...</td>
<td>The converter tried to create an instance of a model and an error occurred during the operation in MLDesigner. The model for the instance was found, but the instance could not be added. Different from the error message 'Could not find an MLDesigner primitive/module...'. There is no special reason for this case because you can create an instance of every model and so this error should normally not appear.</td>
<td>You must add the instance manually, set the argument values and connect the ports in the MLDesigner.</td>
</tr>
<tr>
<td>Could not connect a port of ...</td>
<td>A port which was part of a connection was not found in MLDesigner. This can be a formal port or a port of an instance. Possible causes for this error message are:</td>
<td>You must create the missing port manually and connect it in the instance.</td>
</tr>
<tr>
<td></td>
<td>- the model of the instance is a user defined model and there were problems creating the missing port. In this case you will get another error message for the port that could not be created or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- the instance belongs to a model which had conflicts in the association table. As a result, there were two different models with the same internal id but different interfaces.</td>
<td></td>
</tr>
<tr>
<td>Error Dialog</td>
<td>Cause</td>
<td>Solution</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Could not find ... in instance ...</td>
<td>A parameter, memory, event, resource or port was not found in an instance even if it is referenced in the model definition file. Possible cases for this error message are:</td>
<td>You must create the missing argument or port manually and set the value of the instance or connect the port in the instance.</td>
</tr>
<tr>
<td></td>
<td>• the model of the instance is a user defined model and there were problems creating the missing argument or port. In this case you will get another error message for the argument or port that could not be created, or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• the instance belongs to a model which had conflicts in the association table. So there were two different models with the same internal id and they had a different interface.</td>
<td></td>
</tr>
<tr>
<td>Could not find a member of ...</td>
<td>A member of a the composite data structure was not found during the attempt to set it’s value (e.g. in an internal memory).The reason could be a data structure member of a user-defined data structure could not be created. There will be another error message for this member in this case.</td>
<td>The user has to add the missing data structure member and set the value after this.</td>
</tr>
</tbody>
</table>
### Error Dialog

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Could not find an MLDesigner primitive/module for the BONeS primitive/module ... in ... | The conversion utility could not find an MLDesigner model corresponding to an entry in the association table of the library. This may happen in two cases:  
  - there were dependencies to other libraries and the path of at least one of these libraries was not set properly so the models there could not be found or  
  - a primitive or module from the BONeS core primitives or modules was used which is not now implemented in MLDesigner.  
  This error occurs if the converter tries to create an instance of this primitive or module. | For the first case you must convert again with properly set library paths. In the second case you must convert this core primitive or module directly from the particular BONeS core library, then add the instance manually, set the argument values and connect the instance ports. |
<p>| Could not find parent data structure... | The parent of a data structure was not found during the creation of a child data structure. This might indicate that conversion of the required parent data structure was unsuccessful. (In this case, there will be another error message about the parent.) | |
| Could not find type of... | The type of a port, memory, data structure member, parameter... could not be found in MLDesigner. This can happen if a data structure could not be created and the particular element is of this type. | The user has to set the type of the particular element manually in the MLDesigner after creating the data structure. |
| Could not read source file of primitive ... | The primitive source file (.c or .cc) of the BONeS Designer primitive could not be found. This may happen with older versions of the BONeS Designer. | You must make sure that the primitive source files are in the same directory as the model definition files (as is normal in BONeS Designer) or upgrade your libraries to at least BONeS Designer version 4.0. |</p>
<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Could not set parameter ... of internal resource ... | A parameter of an internal resource could not be found or the value of it could not be set. This may happen in two cases:
  - the parameter was not found in the resource (maybe in older versions of BONeS Designer the resources had different definitions) or
  - the value specified for this parameter in the BONeS Designer is not valid. | The best solution for the first case is to upgrade the models to at least BONeS Designer 4.0 and convert the models again. For the second case make sure that the value specified in the BONeS Designer is valid and convert again or change the value manually in the MLDesigner. |
| Could not set value of data structure member ... - ... is not in Enum ... | The default value for a data structure member of type enumeration could not be set because the specified value was not found in enumeration. A wrong default value in the BONeS Designer should be the reason. | You must set the default value manually or correct the BONeS Designer data structure and convert again. |
| Data structure ... in library ... can not contain any enumeration values | This error message appears if there is a composite data structure with an enumeration value as member. This is normally not possible in BONeS Designer so this error message should not appear. Normally MLDesigner decides from the parent of the data structure if you can add members or enumeration values. If both are present this error message appears. | You must decide whether you want a composite or an enumeration data structure. Correct the BONeS Designer model and convert again or change the data structure in MLDesigner. |
| Error in reading pl file of primitive ... | The converted primitive had already a pl file which contained errors and so it could not be parsed by the MLDesigner. | You must correct the existing pl file of the primitive and convert the BONeS Designer primitive again or convert the code manually. |
| FSM primitives (...) are not converted (until now) | Primitives implemented with an FSM are not converted until now. Anyway the interface is created with an empty implementation and instances of the model are created. | No solution. |
## 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect syntax of...</td>
<td>A section of code in the .mdef file cannot be parsed because of unrecognized syntax. Different error messages result depending on where the incorrect code is located. This should only happen when converting models created with BONeS Designer older than version 4.0.</td>
<td>You must upgrade all libraries to at least version 4.0 of BONeS Designer. If this is not possible you must correct the converted models in MLDesigner. The error message will indicate in which part of the code the error is located so you can correct it.</td>
</tr>
<tr>
<td>Missing ... before/after</td>
<td>See Unexpected/missing...</td>
<td>The primitive source file may not be up to date. In this case you must modify and save the model. This should create the event entries. Rerun the file conversion utility on the revised source file.</td>
</tr>
<tr>
<td>No event entries found in pl file of primitive ...</td>
<td>There may be events in the primitive interface definition, but there are no entries for them in the primitive source file. They are used to place code there if the event is received.</td>
<td>The user has to close all edit windows that belong to models he wants to convert. You must correct the types of the output ports where necessary.</td>
</tr>
<tr>
<td>One or more of the the following models have an open edit window...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Only one deferred input port allowed. Ignoring port ...</td>
<td>In MLDesigner you can set the type of one or more output ports directly to the type of one of the inputs by setting the type to =input_name. In BONeS Designer you only have the deferred property. If more than one input port has this property set to true the converter can not decide which input to take if an output has this property enabled. So the first input found with deferred property set to true is taken for all output ports.</td>
<td></td>
</tr>
<tr>
<td>Parse Error in code file of primitive...</td>
<td>See Unexpected/missing...</td>
<td></td>
</tr>
</tbody>
</table>
# Table 11.1: Common Errors after BONeS Model Conversion

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed parameter in instance ... ignored</td>
<td>Because there is only one global seed for all random number generators in MLDesigner a special parameter would have no use.</td>
<td>Nothing can be done.</td>
</tr>
<tr>
<td>Unexpected end of ...</td>
<td>The model definition file ends before the parser has read all closing brackets.</td>
<td>You must ensure that the models are saved properly by BONeS Designer. Probably an older version of BONeS Designer is the reason.</td>
</tr>
<tr>
<td>Unexpected/missing token/end of expression before/after ...</td>
<td>The BONeS Designer primitive file could not be parsed. The conversion of primitive code files is not finished. This may be caused by several problems (e.g. user defined macros are used in the code,...).</td>
<td>You must convert the code manually.</td>
</tr>
<tr>
<td>Unknown ... type in ...</td>
<td>The parser of the converter expects a special tag (or tags) at some point in the model definition files. If there is an unexpected tag, this error message is created. Sections that are lead by unknown tags are ignored leading to lost information. The tags of the BONeS Designer 4.* are known by the parser and problems of this kind should only appear with older BONeS versions.</td>
<td>You must upgrade your libraries to at least version 4.0 of BONeS Designer.</td>
</tr>
</tbody>
</table>
11.2.7 BONeS Categories

The following section shows a list of BONeS components and their MLDesigner equivalents if one exists. These tables are sorted by category. For a complete list in alphabetical order see sec. 11.2.8.

Address Mapping

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = Name ?</td>
<td>AddressEName</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Address in List (Param) ?</td>
<td>AddressInListMatchingParam</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Address List (No Error)</td>
<td>AddressListNoError</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Address List Matching Param</td>
<td>AddressListMatchingParam</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Check For Duplicate Name</td>
<td>CheckDuplicateName</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Get Address Table Length</td>
<td>LengthOfAddressTable</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Multiple Name to Address</td>
<td>NameToAddressMultiple</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Name to Address</td>
<td>NameToAddress</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Name to Address (File)</td>
<td>NameToAddressFile</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Random Address</td>
<td>RandomAddress</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Write Address Table to Info</td>
<td>DisplayAddressTable</td>
<td>AddressMapping</td>
</tr>
</tbody>
</table>

Table 11.2: Address Mapping

Arithmetic

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>1-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>2-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>3-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>4-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>5-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Expression</td>
<td>Expression</td>
<td>Arithmetic</td>
</tr>
</tbody>
</table>

Table 11.3: Arithmetic

Arithmetic > Integer
### 11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+</td>
<td>PlusConstInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>1-</td>
<td>PlusConstInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>1</td>
<td>SubInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>I*</td>
<td>MultiplyInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>I+</td>
<td>AddInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>I/</td>
<td>DivInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>I/ IMod</td>
<td>ModInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>I/ Protect</td>
<td>DivProtectInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Iabs Val</td>
<td>AbsInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Ichs</td>
<td>ChangeSignInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Igain</td>
<td>GainInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Ilimiter</td>
<td>LimiterInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Imax</td>
<td>MaxInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Imin</td>
<td>MinInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Imod</td>
<td>ModNInt</td>
<td>Arithmetic</td>
</tr>
</tbody>
</table>

Table 11.4: Arithmetic > Integer

### Arithmetic > Real

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/R</td>
<td>InvertFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Bessel (x)</td>
<td>BesselFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Cos</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>exp (x)</td>
<td>ExpFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>ln (x)</td>
<td>LogFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R-</td>
<td>MinusFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R*</td>
<td>MultiplyFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R+</td>
<td>AddFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R/</td>
<td>DivFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R/ Protect</td>
<td>DivProtectFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rabs Val</td>
<td>AbsFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rchs</td>
<td>ChangeSignFloat</td>
<td>Arithmetic</td>
</tr>
</tbody>
</table>
11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rgain</td>
<td>GainFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rlimiter</td>
<td>LimiterFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rmax</td>
<td>MaxFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rmin</td>
<td>MinFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rsqrt</td>
<td>SqrtFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>sin(x)</td>
<td>Sin</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>tan(x)</td>
<td>Tan</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>X^I</td>
<td>PowInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>X ` Iconst</td>
<td>PowConstInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>X^Y</td>
<td>PowFloat</td>
<td>Arithmetic</td>
</tr>
</tbody>
</table>

Table 11.5: Arithmetic > Real

### Bitwise Operations > Integer

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise And</td>
<td>AndBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Left Shift</td>
<td>LeftShiftBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise One’s Complement</td>
<td>OnesComplementBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>OrBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Right Shift</td>
<td>RightShiftBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Xor</td>
<td>XorBit</td>
<td>BitwiseOperations</td>
</tr>
</tbody>
</table>

Table 11.6: Bitwise Operations > Integer

### Comparison

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even</td>
<td>Even</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&lt;</td>
<td>LtInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&lt;=</td>
<td>LeInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&gt;</td>
<td>GtInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&gt;=</td>
<td>GeInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I==</td>
<td>EqInt</td>
<td>Comparison</td>
</tr>
</tbody>
</table>

Table 11.6: Bitwise Operations > Integer
11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd</td>
<td>Odd</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&lt;</td>
<td>LtFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&lt;=</td>
<td>LeFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&gt;</td>
<td>GtFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&gt;=</td>
<td>GeFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R==</td>
<td>EqFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&lt;</td>
<td>LtString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&lt;=</td>
<td>LeString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&gt;</td>
<td>GtString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&gt;=</td>
<td>GeString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str==</td>
<td>EqString</td>
<td>Comparison</td>
</tr>
</tbody>
</table>

Table 11.7: Comparison

Conversions

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int to Real</td>
<td>IntToFloat</td>
<td>Conversion</td>
</tr>
<tr>
<td>Round</td>
<td>FloatRoundToInt</td>
<td>Conversion</td>
</tr>
<tr>
<td>Truncate</td>
<td>FloatFloor</td>
<td>Conversion</td>
</tr>
</tbody>
</table>

Table 11.8: Conversions

Counters

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>FloatAccumulator</td>
<td>Counters</td>
</tr>
<tr>
<td>Circular Counter</td>
<td>CircularAccumulator</td>
<td>Counters</td>
</tr>
<tr>
<td>Counter</td>
<td>Counter</td>
<td>Counters</td>
</tr>
<tr>
<td>Int Accumulator</td>
<td>IntAccumulator</td>
<td>Counters</td>
</tr>
<tr>
<td>Simple Counter</td>
<td>SimpleCounter</td>
<td>Counters</td>
</tr>
<tr>
<td>Up/Down Counter</td>
<td>UDCounter</td>
<td>Counters</td>
</tr>
<tr>
<td>Up/Down Counter -Change Value</td>
<td>VarUDCounter</td>
<td>Counters</td>
</tr>
</tbody>
</table>

Table 11.9: Counters
DS Access/Modify

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add to Field</td>
<td>AddToFieldValueDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Add to Field - Param</td>
<td>AddFieldParamDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Coerce DS</td>
<td>CoerceDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Create DS</td>
<td>CreateDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Create DS - Param</td>
<td>CreateParamDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Declare DS</td>
<td>DeclareDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Insert Field</td>
<td>InsertFieldDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Insert Field - Param</td>
<td>InsertFieldParamDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Insert Field - TNow</td>
<td>InsertFieldTNowDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Select Field</td>
<td>SelectFieldDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Type Switch</td>
<td>TypeSwitchDS</td>
<td>DSHandling</td>
</tr>
</tbody>
</table>

Table 11.10: DS Access/Modify

DS TYPE Operations

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>T==</td>
<td>TypeIsEqual</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Type Compatible</td>
<td>TypeIsCompatible</td>
<td>DSHandling</td>
</tr>
<tr>
<td>TYPE Const</td>
<td>TypeConst</td>
<td>DSHandling</td>
</tr>
</tbody>
</table>

Table 11.11: DS TYPE Operations

Delays

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs Delay</td>
<td>VarAndDelay</td>
<td>Delays</td>
</tr>
<tr>
<td>Fixed Abs Delay</td>
<td>Delay</td>
<td>Delays</td>
</tr>
</tbody>
</table>

Table 11.12: Delays

Execution Control
### 11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute In Order</td>
<td>ExecuteInOrder</td>
<td>Control</td>
</tr>
<tr>
<td>Execute In Order 3</td>
<td>ExecuteInOrder</td>
<td>Control</td>
</tr>
<tr>
<td>Execute In Order 4</td>
<td>ExecuteInOrder</td>
<td>Control</td>
</tr>
<tr>
<td>Gate</td>
<td>GateTrigger</td>
<td>Control</td>
</tr>
<tr>
<td>Gate - On/Off</td>
<td>GateOnOff</td>
<td>Control</td>
</tr>
<tr>
<td>Gate - Clearable</td>
<td>GateWithReset</td>
<td>Control</td>
</tr>
<tr>
<td>Gate - Trigger First</td>
<td>GateTriggerFirst</td>
<td>Control</td>
</tr>
<tr>
<td>Init</td>
<td>Init</td>
<td>Control</td>
</tr>
<tr>
<td>Merge</td>
<td>Merge</td>
<td>Control</td>
</tr>
<tr>
<td>OneWay</td>
<td>OneWay</td>
<td>Control</td>
</tr>
<tr>
<td>One Pulse</td>
<td>OnePulse</td>
<td>Control</td>
</tr>
<tr>
<td>Synchronize</td>
<td>2InOutSynchronize</td>
<td>Control</td>
</tr>
<tr>
<td>Terminate Simulation</td>
<td>TerminateSimulation</td>
<td>Control</td>
</tr>
<tr>
<td>WrapUp</td>
<td>WrapUp</td>
<td>Control</td>
</tr>
</tbody>
</table>

Table 11.13: Execution Control

### File Access

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close File</td>
<td>FileClose</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File (APPEND)</td>
<td>FileOpenAppendConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File (READ)</td>
<td>FileOpenReadConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File (WRITE)</td>
<td>FileOpenWriteConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File w/ String Input (APPEND)</td>
<td>FileOpenAppend</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File w/ String Input (READ)</td>
<td>FileOpenRead</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File w/ String Input (WRITE)</td>
<td>FileOpenWrite</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Read File (INTEGER)</td>
<td>FileReadInt</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Read File (REAL)</td>
<td>FileReadFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Read File (STRING Line)</td>
<td>FileReadString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Error (NUMERIC)</td>
<td>WriteErrorFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Error (STRING)</td>
<td>WriteErrorString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Error (STRING) - Param</td>
<td>WriteErrorStringConst</td>
<td>FileHandling</td>
</tr>
</tbody>
</table>

Table 11.14: File Access
## 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write File (INTEGER)</td>
<td>FileWriteInt</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (REAL)</td>
<td>FileWriteFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (ROOT-OBJECT)</td>
<td>FileWriteDS</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (STRING)</td>
<td>FileWriteString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (STRING) - Param</td>
<td>FileWriteStringConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Info (NUMERIC)</td>
<td>WriteInfoFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Info (STRING)</td>
<td>WriteInfoString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Info (STRING) - Param</td>
<td>WriteInfoStringConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Warning (NUMERIC)</td>
<td>WriteWarningFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Warning (STRING)</td>
<td>WriteWarningString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Warning (STRING) - Param</td>
<td>WriteWarningStringConst</td>
<td>FileHandling</td>
</tr>
</tbody>
</table>

Table 11.14: File Access
### Goto Blocks > Global Scope

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goto Named Location</td>
<td>GotoLocationByName</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Goto Numbered Location</td>
<td>GotoLocationByNumber</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Named Location (multiple)</td>
<td>LocationByNameMultiple</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Named Location (single)</td>
<td>LocationByNameSingle</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (multiple, replacement)</td>
<td>LocationByNumberMultipleReplacement</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (single)</td>
<td>LocationByNumberSingle</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (variable)</td>
<td>LocationByNumberVariable</td>
<td>GotoGroup</td>
</tr>
</tbody>
</table>

Table 11.15: GotoGroup

### Goto Blocks > Linked with Memory

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goto Numbered Location (linked)</td>
<td>LinkedGotoLocationByNumber</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (linked, multiple, replacement)</td>
<td>LinkedLocByNumMultipleReplacement</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (linked, single)</td>
<td>LinkedLocationByNumberSingle</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (linked, variable)</td>
<td>LinkedLocationByNumberVariable</td>
<td>GotoGroup</td>
</tr>
</tbody>
</table>

Table 11.16: GotoGroup

### Logical

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>LogicFalse</td>
<td>Logic</td>
</tr>
<tr>
<td>True</td>
<td>LogicTrue</td>
<td>Logic</td>
</tr>
<tr>
<td>And</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Nand</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Nor</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Not</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Nxor</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Or</td>
<td>Logic</td>
<td>Logic</td>
</tr>
</tbody>
</table>
### 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xor</td>
<td>Logic</td>
<td>Logic</td>
</tr>
</tbody>
</table>

**Table 11.17: Logical**

### Loops

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Do</td>
<td>DoLoopVarInt</td>
<td>Loops</td>
</tr>
<tr>
<td>Int Do (0,N-1)</td>
<td>Do0n_1LoopInt</td>
<td>Loops</td>
</tr>
<tr>
<td>Int Do (1,N)</td>
<td>Do1NLoopInt</td>
<td>Loops</td>
</tr>
<tr>
<td>Int Do - Param</td>
<td>DoLoopInt</td>
<td>Loops</td>
</tr>
<tr>
<td>Real Do</td>
<td>DoLoopVarFloat</td>
<td>Loops</td>
</tr>
<tr>
<td>Real Do - Param</td>
<td>DoLoopFloat</td>
<td>Loops</td>
</tr>
</tbody>
</table>

**Table 11.18: Loops**

### Memory Access > Global Memory

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Mem Reporter?</td>
<td>GlobalMemoryReporter</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Read Global Memory</td>
<td>GlobalMemoryRead</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Write Global Memory</td>
<td>GlobalMemoryWrite</td>
<td>MemoryAccess</td>
</tr>
</tbody>
</table>

**Table 11.19: Memory Access > Global Memory**

### Memory Access > Linked Memory

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Read Memory</td>
<td>MemoryActiveRead</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem + 1</td>
<td>MemoryInc1</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem - 1</td>
<td>MemoryDec1</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem Insert Field</td>
<td>MemoryInsertField</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Memory Initialized?</td>
<td>MemoryIsInitialized</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem Select Field</td>
<td>MemorySelectField</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Read Memory</td>
<td>MemoryRead</td>
<td>MemoryAccess</td>
</tr>
</tbody>
</table>

**Table 11.20: Memory Access > Linked Memory**
11.2 Converting BOnES Models

<table>
<thead>
<tr>
<th>BOnES Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Memory</td>
<td>MemoryWrite</td>
<td>MemoryAccess</td>
</tr>
</tbody>
</table>

Table 11.20: Memory Access > Linked Memory

## Memory Access > Local Memory

<table>
<thead>
<tr>
<th>BOnES Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Local Mem</td>
<td>LocalMemoryInt</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Local Mem</td>
<td>LocalMemory</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Multiple Buffers</td>
<td>MemoryMultipleBuffers</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Real Local Mem</td>
<td>LocalMemory</td>
<td>MemoryAccess</td>
</tr>
</tbody>
</table>

Table 11.21: Memory Access > Local Memory

## Miscellaneous

<table>
<thead>
<tr>
<th>BOnES Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Hash</td>
<td>AddHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>Dijkstra</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Get Hash</td>
<td>GetHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Remove Hash</td>
<td>RemoveHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Search Hash</td>
<td>SearchHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Sink</td>
<td>BlackHole</td>
<td>Sinks</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>TableLookup</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Time Between Triggers</td>
<td>TimeBetweenTriggers</td>
<td>Miscellaneous</td>
</tr>
</tbody>
</table>

Table 11.22: Miscellaneous

## Number Generators

<table>
<thead>
<tr>
<th>BOnES Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iconst</td>
<td>GenIntConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Iteration</td>
<td>Iteration</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Rconst</td>
<td>GenFloatConst</td>
<td>NumberGenerators</td>
</tr>
</tbody>
</table>
## Number Generators

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNow</td>
<td>TNow</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>TStop</td>
<td>StopTimeGen</td>
<td>NumberGenerators</td>
</tr>
</tbody>
</table>

**Table 11.23: Number Generators**

## Number Generators - Random

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli Rangen</td>
<td>BernoulliInt</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Bernoulli Rangen - Param</td>
<td>BernoulliIntConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Binomial Rangen</td>
<td>BinomialInt</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Binomial Rangen - Param</td>
<td>BinomialIntConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Erlang Rangen - Param</td>
<td>ErlangFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Expon Rangen</td>
<td>ExponentialFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Expon Rangen - Param</td>
<td>ExponentialFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Gamma Rangen</td>
<td>GammaFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Gamma Rangen - Param</td>
<td>GammaFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Geometric Rangen</td>
<td>GeometricInt</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Geometric Rangen - Param</td>
<td>GeometricIntConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[0,N-1]</td>
<td>Uniform0ToNInt</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[0,N-1] != C</td>
<td>Uniform0ToNNotCInt</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[0,N-1] - Param</td>
<td>Uniform0ToNNotCIntConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[Min,Max]</td>
<td>IntMinMaxFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[Min,Max] - Param</td>
<td>IntMinMaxFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>N(0,1) Rangen</td>
<td>Normal01Float</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Normal Rangen</td>
<td>NormalFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Normal Rangen - Param</td>
<td>NormalFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Poisson Rangen</td>
<td>PoissonFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Poisson Rangen - Param</td>
<td>PoissonFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Rayleigh Rangen</td>
<td>RayleighFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Rice Rangen</td>
<td>RiceFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Triangle (a,b,c) Rangen</td>
<td>TriangleFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Triangle (a,b,c) Rangen - Param</td>
<td>TriangleFloatConst</td>
<td>NumberGenerators</td>
</tr>
</tbody>
</table>
11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>U[0,1) Rangen</td>
<td>Uniform01Rangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Uniform Rangen</td>
<td>UniformFloat</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Uniform Rangen - Param</td>
<td>UniformFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>User_CDF RanGen</td>
<td>CDFFloat</td>
<td>NumberGenerators</td>
</tr>
</tbody>
</table>

Table 11.24: Number Generators > Random

**Quantity-Shared Resource**

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Allocate</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Allocate (Basic)</td>
<td>AllocateBasic</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Allocate (Param)</td>
<td>AllocateParam</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Allocate (Priority)</td>
<td>AllocatePriority</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Change Capacity</td>
<td>ChangeCap</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Consume Resource Units</td>
<td>ConsumeUnits</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Free</td>
<td>Free</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Free (Basic)</td>
<td>Free Basic</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Free (Param)</td>
<td>Free Param</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Migrate Resource Units</td>
<td>MigrateUnits</td>
<td>QuantityResources</td>
</tr>
</tbody>
</table>

Table 11.25: Quantity-Shared Resource

**Quantity-Shared Resource > *Internals***

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Read Q-Resource</td>
<td>ActiveReadQuantityResource</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>QR State of Dimension</td>
<td>QrStateOfDimension</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>QR State of Modified Dimension</td>
<td>QrStateOfModifiedDimension</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>QSR Last Dimension Modified</td>
<td>QrLastDimensionModified</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>QSRState</td>
<td>QrState</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>Read Q-Shared Resource</td>
<td>ReadQuantityResource</td>
<td>QuantityResources/Internals</td>
</tr>
</tbody>
</table>

Table 11.26: Quantity-Shared Resource > *Internals***
## Queues

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO Priority</td>
<td>FIFOPriority</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO Priority w/Length</td>
<td>FIFOPriorityWithLength</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO Priority w/Peak</td>
<td>FIFOPriorityWithPeak</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO Priority w/Reject</td>
<td>FIFOPriorityWithReject</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Peak</td>
<td>FIFOWithPeak</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Reject</td>
<td>FIFOWithReject</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Reset</td>
<td>FIFOPriorityWithReset</td>
<td>Queues</td>
</tr>
<tr>
<td>LIFO Priority</td>
<td>LIFOPriority</td>
<td>Queues</td>
</tr>
<tr>
<td>Multiple Priority Queues w/Peak</td>
<td>MultiplePriorityQueuesWithPeak</td>
<td>Queues</td>
</tr>
<tr>
<td>Multiple Queues</td>
<td>MultipleQueues</td>
<td>Queues</td>
</tr>
<tr>
<td>Multiple Queues w/Peak</td>
<td>MultipleQueuesWithPeak</td>
<td>Queues</td>
</tr>
<tr>
<td>Simple FIFO</td>
<td>SimpleFIFO</td>
<td>Queues</td>
</tr>
<tr>
<td>Simple LIFO</td>
<td>SimpleLIFO</td>
<td>Queues</td>
</tr>
</tbody>
</table>

Table 11.27: Queues

## Queues > Components

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Queue</td>
<td>ClearQueue</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>FIFO Priority w/Stats</td>
<td>FIFOPriorityWithStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Final Queue Statistics</td>
<td>FinalQueueStatistics</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>General Queue Internals</td>
<td>GeneralQueueInternals</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Get Queue Length</td>
<td>GetQueueLength</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Get Queue Length (Priority)</td>
<td>GetQueueLengthPriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Get Queue Statistics</td>
<td>GetQueueStatistics</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Insert (Position)</td>
<td>InsertPosition</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Insert (Priority)</td>
<td>InsertPriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Peek (Position)</td>
<td>PeekPosition</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Peek (Priority)</td>
<td>PeekPriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Peek (Tag)</td>
<td>PeekTag</td>
<td>Queues/Internals</td>
</tr>
</tbody>
</table>
11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Num Check</td>
<td>QueueNumCheck</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Queue Overflow Reporter</td>
<td>QueueOverflowReporter</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Read Active Queue Statistics</td>
<td>ReadActiveQueueStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Remove (Position)</td>
<td>RemovePosition</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Remove (Priority)</td>
<td>RemovePriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Remove (Tag)</td>
<td>RemoveTag</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Reset Stats</td>
<td>ResetStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Update Stats</td>
<td>UpdateStats</td>
<td>Queues/Internals</td>
</tr>
</tbody>
</table>

Table 11.28: Queues> Components

Queues & Servers

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO w/Servers</td>
<td>FIFOWithServers</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>Multiple Servers</td>
<td>MultipleServers</td>
<td>QueuesAndServer</td>
</tr>
</tbody>
</table>

Table 11.29: Queues & Servers

Queues & Servers> *Internals*

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO w/Servers Stats</td>
<td>FIFOWithServerStats</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>FIFO w/Servers</td>
<td>FIFOWithServer</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>Multiple Servers</td>
<td>MultipleServers</td>
<td>QueuesAndServers</td>
</tr>
<tr>
<td>Parallel Queues</td>
<td>ParallelQueues</td>
<td>QueuesAndServer</td>
</tr>
</tbody>
</table>

Table 11.30: Queues & Servers> *Internals*

SET Operations

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER to SET</td>
<td>IntegerToEnum</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>S==</td>
<td>(EnumIsEqual)</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>S == Switch</td>
<td>EnumIsEqualSwitch</td>
<td>EnumOperations</td>
</tr>
</tbody>
</table>
### 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sconst</td>
<td>EnumConst</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>SET Rangen</td>
<td>EnumRanGen</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>SET to INTEGER</td>
<td>EnumToInteger</td>
<td>EnumOperations</td>
</tr>
</tbody>
</table>

Table 11.31: SET Operations

### Server Resource

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>get SRM</td>
<td>GetSRM</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Modify SRM</td>
<td>ModifySRM</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service</td>
<td>Service</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service (Basic)</td>
<td>ServiceBasic</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service (Dimensioned)</td>
<td>ServiceDimensioned</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service (Priority)</td>
<td>ServicePriority</td>
<td>ServerResources</td>
</tr>
</tbody>
</table>

Table 11.32: Server Resource

### Server Resource > *Internals*

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Read S-Resource</td>
<td>ActiveReadServerResource</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>Read Server Resource</td>
<td>ReadServerResource</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>SR Last Dimension Modified</td>
<td>SrLastDimensionModifies</td>
<td>No MLD Category found !</td>
</tr>
<tr>
<td>SR State</td>
<td>SrState</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>SR State of Dimension</td>
<td>SrStateOfDimension</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>SR State of Modified Dimension</td>
<td>SrStateOfModifiedDimension</td>
<td>ServerResources/Internals</td>
</tr>
</tbody>
</table>

Table 11.33: Server Resource > *Internals*

### Statistical > Batch

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Mean</td>
<td>BatchMean</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Rmax</td>
<td>BatchRMax</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Rmin</td>
<td>BatchRMin</td>
<td>Statistics</td>
</tr>
</tbody>
</table>
### 11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Statistics</td>
<td>BatchStatistics</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Timing</td>
<td>BatchTiming</td>
<td>Statistics</td>
</tr>
</tbody>
</table>

Table 11.34: Statistical > Batch

#### Statistical > General

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Average</td>
<td>Statistics</td>
</tr>
<tr>
<td>Dimensioned Ensemble Average</td>
<td>DimEnsembleAverage</td>
<td>Statistics</td>
</tr>
<tr>
<td>Dimensioned Ensemble Avg w/ Min and Max</td>
<td>DimEnsembleAvgMinMax</td>
<td>Statistics</td>
</tr>
<tr>
<td>Dimensioned Time Average</td>
<td>DimensionedTimeAverage</td>
<td>Statistics</td>
</tr>
<tr>
<td>Dimensioned Time Avg w/ Min and Max</td>
<td>DimensionedTimeAvgMinMax</td>
<td>Statistics</td>
</tr>
<tr>
<td>General N-th Moment</td>
<td>GeneralNthMoment</td>
<td>Statistics</td>
</tr>
<tr>
<td>Mean &amp; Variance</td>
<td>MeanAndVariance</td>
<td>Statistics</td>
</tr>
<tr>
<td>Throughput</td>
<td>Throughput</td>
<td>Statistics</td>
</tr>
<tr>
<td>Time Average</td>
<td>TimeAverage</td>
<td>Statistics</td>
</tr>
<tr>
<td>Weighted General Moments</td>
<td>WeightedGeneralMoments</td>
<td>Statistics</td>
</tr>
<tr>
<td>Weighted Mean and Variance</td>
<td>WeightedMeanAndVariance</td>
<td>Statistics</td>
</tr>
</tbody>
</table>

Table 11.35: Statistical > General

#### Statistical > Histogram

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>Histogram</td>
<td>Statistics</td>
</tr>
</tbody>
</table>

Table 11.36: Statistical > Histogram
### Statistical > Histogram > *Internals*

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find Bin Number</td>
<td>FindBinNumber</td>
<td>Statistics</td>
</tr>
<tr>
<td>Report Histogram</td>
<td>ReportHistogram</td>
<td>Statistics</td>
</tr>
</tbody>
</table>

Table 11.37: Statistical > Histogram > *Internals*

### Statistical > Misc

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct Dimensioned Basic Statistics</td>
<td>ConstructDimensionedBasicStatistics</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct Dimensioned Basic Stats</td>
<td>ConstructDimensionedBasicStatsAllFields</td>
<td>Statistics</td>
</tr>
<tr>
<td>- all fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Construct TA Stats w/Priorities</td>
<td>ConstructTAStatsWithPriority</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct Time-Average Statistics</td>
<td>ConstructTimeAverageStatistics</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct Time-Avg Stats - all fields</td>
<td>ConstructTimeAvgStatsAllFields</td>
<td>Statistics</td>
</tr>
</tbody>
</table>

Table 11.38: Statistical > Misc

### String Operations

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract String</td>
<td>XtractString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>Int to String</td>
<td>ConvertToString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>Real to String</td>
<td>ConvertToString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>Split String</td>
<td>SplitString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Concat</td>
<td>ConcatString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Constant</td>
<td>ConstStringGen</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Length</td>
<td>StringLength</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Search</td>
<td>FindOccurrence</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String to Int</td>
<td>StringToInt</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String to Real</td>
<td>StringToFloat</td>
<td>StringOperations</td>
</tr>
</tbody>
</table>

Table 11.39: String Operations
### Switches

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Input Expression Switch</td>
<td>InputExpressionSwitch</td>
<td></td>
</tr>
<tr>
<td>Expression Switch</td>
<td>ExpressionSwitch</td>
<td></td>
</tr>
<tr>
<td>I &lt;= C ?</td>
<td>IntLeThresh</td>
<td></td>
</tr>
<tr>
<td>I &lt; C ?</td>
<td>IntLtThresh</td>
<td></td>
</tr>
<tr>
<td>I &gt;= C ?</td>
<td>IntGeThresh</td>
<td></td>
</tr>
<tr>
<td>I &gt; C ?</td>
<td>IntGtThresh</td>
<td></td>
</tr>
<tr>
<td>I == C ?</td>
<td>IntEqThresh</td>
<td></td>
</tr>
<tr>
<td>Memory Switch</td>
<td>MemorySwitch</td>
<td></td>
</tr>
<tr>
<td>R &lt;= C ?</td>
<td>FloatLeThresh</td>
<td></td>
</tr>
<tr>
<td>R &lt; C ?</td>
<td>FloatLtThresh</td>
<td></td>
</tr>
<tr>
<td>R &gt;= C ?</td>
<td>FloatGeThresh</td>
<td></td>
</tr>
<tr>
<td>R &gt; C ?</td>
<td>FloatGtThresh</td>
<td></td>
</tr>
<tr>
<td>R == C ?</td>
<td>FloatEqThresh</td>
<td></td>
</tr>
<tr>
<td>Random Switch</td>
<td>VarProbSwitch</td>
<td></td>
</tr>
<tr>
<td>Random Switch - Param</td>
<td>ProbSwitch</td>
<td></td>
</tr>
<tr>
<td>Real Within Boundaries ?</td>
<td>FloatWithinRange</td>
<td></td>
</tr>
<tr>
<td>Str == C ?</td>
<td>StringEqualsConst</td>
<td></td>
</tr>
<tr>
<td>Switch</td>
<td>Switch</td>
<td></td>
</tr>
<tr>
<td>Switch 4-Way</td>
<td>4WaySwitchConst</td>
<td></td>
</tr>
<tr>
<td>T &gt; Startup ?</td>
<td>TNowGtStartup</td>
<td></td>
</tr>
<tr>
<td>TNow &gt;= Param ?</td>
<td>TNowGeStartup</td>
<td></td>
</tr>
<tr>
<td>True N Times</td>
<td>TrueNTimesConst</td>
<td></td>
</tr>
</tbody>
</table>

Table 11.40: Switches

### Timers

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm Active</td>
<td>AlarmActive</td>
<td>Timers</td>
</tr>
<tr>
<td>Cancel Alarm</td>
<td>CancelAlarm</td>
<td>Timers</td>
</tr>
<tr>
<td>Cancel Timer</td>
<td>CancelTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Residual Time</td>
<td>ResidualTime</td>
<td>Timers</td>
</tr>
</tbody>
</table>
11 Import/Conversion of Models

Table 11.41: Timers

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart Alarm</td>
<td>RestartAlarm</td>
<td>Timers</td>
</tr>
<tr>
<td>Restart Timer</td>
<td>RestartTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Service Timer</td>
<td>ServiceTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Set Alarm</td>
<td>SetAlarm</td>
<td>Timers</td>
</tr>
<tr>
<td>Start Timer</td>
<td>StartTimer</td>
<td>Timers</td>
</tr>
</tbody>
</table>

Traffic Generators

Table 11.42: Traffic Generators

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Pulse Train</td>
<td>ArbitraryPulseTrain</td>
<td>Sources</td>
</tr>
<tr>
<td>Bursty Pulse Train (Expon,Geom)</td>
<td>BurstyPulseTrainExponGeom</td>
<td>Sources</td>
</tr>
<tr>
<td>Poisson Pulse Train</td>
<td>PoissonPulseTrain</td>
<td>Sources</td>
</tr>
<tr>
<td>Poisson Pulse Train (variable Inter-Pulse Time)</td>
<td>PoissonPulseTrainVarInterPulseTime</td>
<td>Sources</td>
</tr>
<tr>
<td>Poisson Pulse Train -Hier</td>
<td>PoissonPulseTrainHier</td>
<td>Sources</td>
</tr>
<tr>
<td>Uniform Pulse Train</td>
<td>UniformPulseTrain</td>
<td>Sources</td>
</tr>
<tr>
<td>Uniform Pulse Train (variable Inter-Pulse Time)</td>
<td>UniformPulseTrainVarInterPulseTime</td>
<td>Sources</td>
</tr>
</tbody>
</table>

Vector Operations> General

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Access Element</td>
<td>AccessElementVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Change Length</td>
<td>ChangeLengthVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Create</td>
<td>CreateVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Length</td>
<td>LengthOfVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Access</td>
<td>MemoryAccessVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Init</td>
<td>MemoryInitVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Length</td>
<td>MemoryLengthVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Set</td>
<td>MemorySetVector</td>
<td>VectorOperations/General</td>
</tr>
</tbody>
</table>
### 11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Set Element</td>
<td>SetElementVector</td>
<td>VectorOperations/General</td>
</tr>
</tbody>
</table>

Table 11.43: Vector Operations > General

#### Vector Operations > Int Matrix

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMatrix Access Element</td>
<td>AccessElementIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Create</td>
<td>CreateIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Dimensions</td>
<td>DimensionsIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Mem Access</td>
<td>MemoryAccessIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Mem Set</td>
<td>MemorySetIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Read File</td>
<td>ReadFileIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Set Element</td>
<td>SetElementIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Write File</td>
<td>WriteFileIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>IMatrix Write File (tabular)</td>
<td>WriteFileTabIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>STRING to IMatrix</td>
<td>StringToIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
</tbody>
</table>

Table 11.44: Vector Operations > Int Matrix

#### Vector Operations > Int Vector

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVect Access Element</td>
<td>AccessElementIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Create</td>
<td>CreateIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Index of Value</td>
<td>IndexOfValueIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Length</td>
<td>LengthOfIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem +/-</td>
<td>MemoryAddIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem +/- With Integer Output</td>
<td>MemoryAddIVectorWithIntOut</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem Access</td>
<td>MemoryAccessIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem Change Length</td>
<td>MemoryChangeLengthIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem Init</td>
<td>MemoryInitIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem Largest in Range</td>
<td>MemoryLargestInRangeIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
</tbody>
</table>
11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVect Mem Length</td>
<td>MemoryLengthIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem Set</td>
<td>MemorySetIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Mem Smallest in Range</td>
<td>MemorySmallestInRangeIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Read File</td>
<td>ReadFileIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Set Element</td>
<td>SetElementIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVect Write File</td>
<td>WriteFileIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>STRING to IVect</td>
<td>StringToIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
</tbody>
</table>

Table 11.45: Vector Operations > Int Vector

**Vector Operations > Real Matrix**

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMatrix Access Element</td>
<td>AccessElementFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Create</td>
<td>CreateFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Dimensions</td>
<td>DimensionsFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Mem Access</td>
<td>MemoryAccessFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Mem Set</td>
<td>MemorySetFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Read File</td>
<td>ReadFileFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Set Element</td>
<td>SetElementFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Write File</td>
<td>WriteFileFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Write File (tabular)</td>
<td>WriteFileTabFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>STRING to RMatrix</td>
<td>StringToFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
</tbody>
</table>

Table 11.46: Vector Operations > Real Matrix

**Vector Operations > Real Vector**

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVect Access Element</td>
<td>AccessElementFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Create</td>
<td>CreateFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Length</td>
<td>LengthOfFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem +/-</td>
<td>MemoryAddFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
</tbody>
</table>
11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R Vect Mem +/- With Real Output</td>
<td>MemoryAddFVectorWithFloatOut</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Access</td>
<td>MemoryAccessFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Change Length</td>
<td>MemoryChangeLengthFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Init</td>
<td>MemoryInitFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Largest in Range</td>
<td>MemoryLargestInRangeFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Length</td>
<td>MemoryLengthFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Set</td>
<td>MemorySetFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Mem Smallest in Range</td>
<td>MemorySmallestInRangeFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Read File</td>
<td>ReadFileFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Set Element</td>
<td>SetElementFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>R Vect Write File</td>
<td>WriteFileFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>STRING to R Vect</td>
<td>StringToFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
</tbody>
</table>

Table 11.47: Vector Operations > Real Vector

11.2.8 BONeS Primitives

This is a list of all BONeS primitives with MLDesigner equivalents in alphabetical order.

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+</td>
<td>PlusConstInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>1-</td>
<td>PlusConstInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>1-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>1-Input Expression Switch</td>
<td>InputExpressionSwitch</td>
<td>Switches</td>
</tr>
<tr>
<td>1/R</td>
<td>InvertFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>10-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>2-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>3-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>4-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>5-Input Expression</td>
<td>InputExpression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Abs Delay</td>
<td>VarAndDelay</td>
<td>Delays</td>
</tr>
<tr>
<td>Accumulator</td>
<td>FloatAccumulator</td>
<td>Counters</td>
</tr>
<tr>
<td>Active Read Memory</td>
<td>MemoryActiveRead</td>
<td>MemoryAccess</td>
</tr>
</tbody>
</table>
## 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Read Q-Resource</td>
<td>ActiveReadQuantityResource</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>Active Read S-Resource</td>
<td>ActiveReadServerResource</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>Add Hash</td>
<td>AddHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Add to Field</td>
<td>AddToFieldValueDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Add to Field - Param</td>
<td>AddFieldParamDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Address = Name ?</td>
<td>AddressEName</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Address in List (Param) ?</td>
<td>AddressInListMatchingParam</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Address List (No Error)</td>
<td>AddressListNoError</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Address List Matching Param</td>
<td>AddressListMatchingParam</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Alarm Active</td>
<td>AlarmActive</td>
<td>Timers</td>
</tr>
<tr>
<td>Allocate</td>
<td>Allocate</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Allocate (Basic)</td>
<td>AllocateBasic</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Allocate (Param)</td>
<td>AllocateParam</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Allocate (Priority)</td>
<td>AllocatePriority</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>And</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Arbitrary Pulse Train</td>
<td>ArbitraryPulseTrain</td>
<td>Sources</td>
</tr>
<tr>
<td>Average</td>
<td>Average</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Mean</td>
<td>BatchMean</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Rmax</td>
<td>BatchRMax</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Rmin</td>
<td>BatchRMin</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Statistics</td>
<td>BatchStatistics</td>
<td>Statistics</td>
</tr>
<tr>
<td>Batch Timing</td>
<td>BatchTiming</td>
<td>Statistics</td>
</tr>
<tr>
<td>Bernoulli Rangen</td>
<td>BernoulliRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Bernoulli Rangen - Param</td>
<td>BernoulliRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Bessel (x)</td>
<td>BesselFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Binomial Rangen</td>
<td>BinomialRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Binomial Rangen - Param</td>
<td>BinomialRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Bitwise And</td>
<td>AndBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Left Shift</td>
<td>LeftShiftBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise One’s Complement</td>
<td>OnesComplementBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>OrBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>--------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Bitwise Right Shift</td>
<td>RightShiftBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bitwise Xor</td>
<td>XorBit</td>
<td>BitwiseOperations</td>
</tr>
<tr>
<td>Bursty Pulse Train (Expon, Geom)</td>
<td>BurstyPulseTrainExponGeom</td>
<td>Sources</td>
</tr>
<tr>
<td>Cancel Alarm</td>
<td>CancelAlarm</td>
<td>Timers</td>
</tr>
<tr>
<td>Cancel Timer</td>
<td>CancelTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Change Capacity</td>
<td>ChangeCap</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Check For Duplicate Name</td>
<td>CheckDuplicateName</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Circular Counter</td>
<td>CircularAccumulator</td>
<td>Counters</td>
</tr>
<tr>
<td>Clear Queue</td>
<td>ClearQueue</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Close File</td>
<td>FileClose</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Coerce DS</td>
<td>CoerceDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Construct Dimensioned Basic Statistics</td>
<td>ConstructDimensionedBasicStatistics</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct Dimensioned Basic Stats - all fields</td>
<td>ConstructDimensionedBasicStatsAllFields</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct TA Stats w/Priorities</td>
<td>ConstructTASStatsWithPriority</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct Time-Average Statistics</td>
<td>ConstructTimeAverageStatistics</td>
<td>Statistics</td>
</tr>
<tr>
<td>Construct Time-Avg Stats - all fields</td>
<td>ConstructTimeAvgStatsAllFields</td>
<td>Statistics</td>
</tr>
<tr>
<td>Consume Resource Units</td>
<td>ConsumeUnits</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Cos</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Counter</td>
<td>Counter</td>
<td>Counters</td>
</tr>
<tr>
<td>Create DS</td>
<td>CreateDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Create DS - Param</td>
<td>CreateParamDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Declare DS</td>
<td>DeclareDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>Dijkstra</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Dimensioned Ensemble Average</td>
<td>DimEnsembleAverage</td>
<td>Statistics</td>
</tr>
<tr>
<td>Dimensioned Ensemble Avg w/ Min and Max</td>
<td>DimEnsembleAvgMinMax</td>
<td>Statistics</td>
</tr>
<tr>
<td>Dimensioned Time Average</td>
<td>DimensionedTimeAverage</td>
<td>Statistics</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Dimensioned Time Avg w/ Min and Max</td>
<td>DimensionedTimeAvgMinMax</td>
<td>Statistics</td>
</tr>
<tr>
<td>Erlang Rangen - Param</td>
<td>ErlangRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Even</td>
<td>Even</td>
<td>Comparison</td>
</tr>
<tr>
<td>Execute In Order</td>
<td>ExecuteInOrder</td>
<td>Control</td>
</tr>
<tr>
<td>Execute In Order 3</td>
<td>ExecuteInOrder</td>
<td>Control</td>
</tr>
<tr>
<td>Execute In Order 4</td>
<td>ExecuteInOrder</td>
<td>Control</td>
</tr>
<tr>
<td>exp (x)</td>
<td>ExpFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Expon Rangen</td>
<td>ExpRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Expon Rangen - Param</td>
<td>ExpRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Expression</td>
<td>Expression</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Expression Switch</td>
<td>ExpressionSwitch</td>
<td>Switches</td>
</tr>
<tr>
<td>Extract String</td>
<td>XtractString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>False</td>
<td>LogicFalse</td>
<td>Logic</td>
</tr>
<tr>
<td>FIFO Priority w/Stats</td>
<td>FIFOPriorityWithStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>FIFO Priority</td>
<td>FIFOPriority</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO Priority w/Length</td>
<td>FIFOPriorityWithLength</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO Priority w/Peak</td>
<td>FIFOPriorityWithPeak</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO Priority w/Reject</td>
<td>FIFOPriorityWithReject</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Peek</td>
<td>FIFOWithPeek</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Reject</td>
<td>FIFOWithReject</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Reset</td>
<td>FIFOPriorityWithReset</td>
<td>Queues</td>
</tr>
<tr>
<td>FIFO w/Servers</td>
<td>FIFOWithServers</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>FIFO w/Servers Stats</td>
<td>FIFOWithServerStats</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>Final Queue Statistics</td>
<td>FinalQueueStatistics</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Find Bin Number</td>
<td>FindBinNumber</td>
<td>Statistics</td>
</tr>
<tr>
<td>Fixed Abs Delay</td>
<td>Delay</td>
<td>Delays</td>
</tr>
<tr>
<td>Free</td>
<td>Free</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Free (Basic)</td>
<td>FreeBasic</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Free (Param)</td>
<td>FreeParam</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Gamma Rangen</td>
<td>GammaRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Gamma Rangen - Param</td>
<td>GammaRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Gate</td>
<td>GateTrigger</td>
<td>Control</td>
</tr>
<tr>
<td>Gate - On/Off</td>
<td>GateOnOff</td>
<td>Control</td>
</tr>
<tr>
<td>Gate - Clearable</td>
<td>GateWithReset</td>
<td>Control</td>
</tr>
<tr>
<td>Gate - Trigger First</td>
<td>GateTriggerFirst</td>
<td>Control</td>
</tr>
<tr>
<td>Get Address Table Length</td>
<td>LengthOfAddressTable</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>General N-th Moment</td>
<td>GeneralNthMoment</td>
<td>Statistics</td>
</tr>
<tr>
<td>General Queue</td>
<td>GeneralQueue</td>
<td>Queues</td>
</tr>
<tr>
<td>General Queue Internals</td>
<td>GeneralQueueInternals</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Geometric Rangen</td>
<td>GeometricRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Geometric Rangen - Param</td>
<td>GeometricRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Get Hash</td>
<td>GetHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Get Queue Length</td>
<td>GetQueueLength</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Get Queue Length (Priority)</td>
<td>GetQueueLengthPriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Get Queue Statistics</td>
<td>GetQueueStatistics</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>get SRM</td>
<td>GetSRM</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Global Mem Reporter?</td>
<td>GlobalMemoryReporter</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Goto Named Location</td>
<td>GotoLocationByName</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Goto Numbered Location</td>
<td>GotoLocationByNumber</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Goto Numbered Location (linked)</td>
<td>LinkedGotoLocationByNumber</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Histogram</td>
<td>Histogram</td>
<td>Statistics</td>
</tr>
<tr>
<td>I &lt; C ?</td>
<td>IntLtThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>I &lt;= C ?</td>
<td>IntLeThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>I &gt; C ?</td>
<td>IntGtThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>I &gt;= C ?</td>
<td>IntGeThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>I == C ?</td>
<td>IntEqThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>I&lt;</td>
<td>LtInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&lt;=</td>
<td>LeInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&gt;</td>
<td>GtInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I&gt;=</td>
<td>GeInt</td>
<td>Comparison</td>
</tr>
<tr>
<td>I*</td>
<td>MultiplyInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>I+</td>
<td>AddInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td></td>
<td>SubInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>DivInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>ModInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>DivProtectInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>EqInt</td>
<td>Comparison</td>
</tr>
<tr>
<td></td>
<td>AbsInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>ChangeSignInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>GenIntConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td></td>
<td>GainInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>LimiterInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>AccessElementIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>CreateIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>DimensionsIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>MemoryAccessIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>MemorySetIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>ReadFileIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>SetElementIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>WriteFileIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>WriteFileTabIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td></td>
<td>MaxInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>MinInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>ModNInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>Init</td>
<td>Control</td>
</tr>
<tr>
<td></td>
<td>InsertPosition</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td></td>
<td>InsertPriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td></td>
<td>InsertFieldDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td></td>
<td>InsertFieldParamDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td></td>
<td>InsertFieldTNowDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td></td>
<td>IntAccumulator</td>
<td>Counters</td>
</tr>
<tr>
<td></td>
<td>DoLoopVarInt</td>
<td>Loops</td>
</tr>
<tr>
<td></td>
<td>Do0n_ILoopInt</td>
<td>Loops</td>
</tr>
<tr>
<td>BONEs Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Int Do (1,N)</td>
<td>Do1NLoopInt</td>
<td>Loops</td>
</tr>
<tr>
<td>Int Do - Param</td>
<td>DoLoopInt</td>
<td>Loops</td>
</tr>
<tr>
<td>Int Local Mem</td>
<td>LocalMemoryInt</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Int to Real</td>
<td>IntToFloat</td>
<td>Conversion</td>
</tr>
<tr>
<td>Int to String</td>
<td>ConvertToString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>INTEGER to SET</td>
<td>Iteration</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>Iteration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IU[0,N-1]</td>
<td>Int0NRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[0,N-1] != C</td>
<td>Int0NNotCRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[0,N-1] - Param</td>
<td>Int0NRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[Min,Max]</td>
<td>IntMinMaxRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IU[Min,Max] - Param</td>
<td>IntMinMaxRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>IVec Access Element</td>
<td>AccessElementIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Create</td>
<td>CreateIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Index of Value</td>
<td>IndexOfValueIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Length</td>
<td>LengthOfIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem +/-</td>
<td>MemoryAddIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem +/- With Integer Output</td>
<td>MemoryAddIVectorWithIntOut</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Access</td>
<td>MemoryAccessIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Change Length</td>
<td>MemoryChangeLengthIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Init</td>
<td>MemoryInitIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Largest in Range</td>
<td>MemoryLargestInRangeIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Length</td>
<td>MemoryLengthIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Set</td>
<td>MemorySetIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Mem Smallest in Range</td>
<td>MemorySmallestInRangeIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Read File</td>
<td>ReadFileIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Set Element</td>
<td>SetElementIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>IVec Write File</td>
<td>WriteFileIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>LIFO Priority</td>
<td>LIFOPriority</td>
<td>Queues</td>
</tr>
<tr>
<td>ln (x)</td>
<td>LogFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>--------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Local Mem</td>
<td>LocalMemory</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mean &amp; Variance</td>
<td>MeanAndVariance</td>
<td>Statistics</td>
</tr>
<tr>
<td>Mem + 1</td>
<td>MemoryInc1</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem - 1</td>
<td>MemoryDec1</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem Insert Field</td>
<td>MemoryInsertField</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Mem Select Field</td>
<td>MemorySelectField</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Memory Initialized?</td>
<td>MemoryIsInitialized</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Memory Switch</td>
<td>MemorySwitch</td>
<td>Switches</td>
</tr>
<tr>
<td>Merge</td>
<td>Merge</td>
<td>Control</td>
</tr>
<tr>
<td>Migrate Resource Units</td>
<td>MigrateUnits</td>
<td>QuantityResources</td>
</tr>
<tr>
<td>Modify SRM</td>
<td>ModifySRM</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Multiple Buffers</td>
<td>MemoryMultipleBuffers</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Multiple Name to Address</td>
<td>NameToAddressMultiple</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Multiple Priority Queues w/Peek</td>
<td>MultiplePriorityQueuesWithPeek</td>
<td>Queues</td>
</tr>
<tr>
<td>Multiple Queues</td>
<td>MultipleQueues</td>
<td>Queues</td>
</tr>
<tr>
<td>Multiple Queues w/Peek</td>
<td>MultipleQueuesWithPeek</td>
<td>Queues</td>
</tr>
<tr>
<td>Multiple Servers</td>
<td>MultipleServers</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>N(0,1) Rangen</td>
<td>NormalMean0Var1Rangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Name to Address</td>
<td>NameToAddress</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Name to Address (File)</td>
<td>NameToAddressFile</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Nand</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Nor</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Normal Rangen</td>
<td>NormalRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Normal Rangen - Param</td>
<td>NormalRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Not</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Named Location (multiple)</td>
<td>LocationByNameMultiple</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Named Location (single)</td>
<td>LocationByNameSingle</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (linked, multiple, replacement)</td>
<td>LinkedLocByNumMultipleReplacement</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Numbered Location (linked, single)</td>
<td>LinkedLocationByNumberSingle</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (linked, variable)</td>
<td>LinkedLocationByNumberVariable</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (multiple, replacement)</td>
<td>LocationByNumberMultipleReplacement</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (single)</td>
<td>LocationByNumberSingle</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Numbered Location (variable)</td>
<td>LocationByNumberVariable</td>
<td>GotoGroup</td>
</tr>
<tr>
<td>Nxnor</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Odd</td>
<td>Odd</td>
<td>Comparison</td>
</tr>
<tr>
<td>One Pulse</td>
<td>OnePulse</td>
<td>Control</td>
</tr>
<tr>
<td>One_Way</td>
<td>OneWay</td>
<td>Control</td>
</tr>
<tr>
<td>Open File (APPEND)</td>
<td>FileOpenAppendConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File (READ)</td>
<td>FileOpenReadConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File (WRITE)</td>
<td>FileOpenWriteConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File w/ String Input (APPEND)</td>
<td>FileOpenAppend</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File w/ String Input (READ)</td>
<td>FileOpenRead</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Open File w/ String Input (WRITE)</td>
<td>FileOpenWrite</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Or</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>Parallel Queues</td>
<td>ParallelQueues</td>
<td>QueuesAndServer</td>
</tr>
<tr>
<td>Peek (Position)</td>
<td>PeekPosition</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Peek (Priority)</td>
<td>PeekPriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Peek (Tag)</td>
<td>PeekTag</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Poisson Pulse Train</td>
<td>PoissonPulseTrain</td>
<td>Sources</td>
</tr>
<tr>
<td>Poisson Pulse Train (variable Inter-Pulse Time)</td>
<td>PoissonPulseTrainVarInterPulseTime</td>
<td>Sources</td>
</tr>
<tr>
<td>Poisson Pulse Train -Hier</td>
<td>PoissonPulseTrainHier</td>
<td>Sources</td>
</tr>
<tr>
<td>Poisson Rangen</td>
<td>PoissonRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Poisson Rangen - Param</td>
<td>PoissonRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>QR State of Dimension</td>
<td>QrStateOfDimension</td>
<td>QuantityResources/Internals</td>
</tr>
</tbody>
</table>
### 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>QR State of Modified Dimension</td>
<td>QrStateOfModifiedDimension</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>QSR Last Dimension Modified</td>
<td>QrLastDimensionModified</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>QSR State</td>
<td>QrState</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>Queue Num Check</td>
<td>QueueNumCheck</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Queue Overflow Reporter</td>
<td>QueueOverflowReporter</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>R &lt; C ?</td>
<td>FloatLtThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>R &lt;= C ?</td>
<td>FloatLeThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>R &gt; C ?</td>
<td>FloatGrThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>R &gt;= C ?</td>
<td>FloatGeThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>R == C ?</td>
<td>FloatEqThresh</td>
<td>Switches</td>
</tr>
<tr>
<td>R&lt;</td>
<td>LtFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&lt;=</td>
<td>LeFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&gt;</td>
<td>GtFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R&gt;=</td>
<td>GeFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>R*</td>
<td>MultiplyFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R+</td>
<td>AddFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R-</td>
<td>MinusFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>DivFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R/ Protect</td>
<td>DivProtectFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>R==</td>
<td>EqFloat</td>
<td>Comparison</td>
</tr>
<tr>
<td>Rabs Val</td>
<td>AbsFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Random Address</td>
<td>RandomAddress</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Random Switch</td>
<td>VarProbSwitch</td>
<td>Switches</td>
</tr>
<tr>
<td>Random Switch - Param</td>
<td>ProbSwitch</td>
<td>Switches</td>
</tr>
<tr>
<td>Rayleigh Rangen</td>
<td>RayleighRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Rchs</td>
<td>ChangeSignFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rconst</td>
<td>GenFloatConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Read Active Queue Statistics</td>
<td>ReadActiveQueueStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Read File (INTEGER)</td>
<td>FileReadInt</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Read File (REAL)</td>
<td>FileReadFloat</td>
<td>FileHandling</td>
</tr>
</tbody>
</table>

**Note:**
- **MLDesigner Version 2.8**
### 11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read File (STRING Line)</td>
<td>FileReadString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Read Global Memory</td>
<td>GlobalMemoryRead</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Read Memory</td>
<td>MemoryRead</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Read Q-Shared Resource</td>
<td>ReadQuantityResource</td>
<td>QuantityResources/Internals</td>
</tr>
<tr>
<td>Read Server Resource</td>
<td>ReadServerResource</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>Real Do</td>
<td>DoLoopVarFloat</td>
<td>Loops</td>
</tr>
<tr>
<td>Real Do - Param</td>
<td>DoLoopFloat</td>
<td>Loops</td>
</tr>
<tr>
<td>Real Local Mem</td>
<td>LocalMemory</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Real to String</td>
<td>ConvertToString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>Real Within Boundaries ?</td>
<td>FloatWithinRange</td>
<td>Switches</td>
</tr>
<tr>
<td>Remove (Position)</td>
<td>RemovePosition</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Remove (Priority)</td>
<td>RemovePriority</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Remove (Tag)</td>
<td>RemoveTag</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Remove Hash</td>
<td>RemoveHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Report Histogram</td>
<td>ReportHistogram</td>
<td>Statistics</td>
</tr>
<tr>
<td>Reset Stats</td>
<td>ResetStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>Residual Time</td>
<td>ResidualTime</td>
<td>Timers</td>
</tr>
<tr>
<td>Restart Alarm</td>
<td>RestartAlarm</td>
<td>Timers</td>
</tr>
<tr>
<td>Restart Timer</td>
<td>RestartTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Rgain</td>
<td>GainFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rice Rangen</td>
<td>RiceRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Rlimiter</td>
<td>LimiterFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>RMatrix Access Element</td>
<td>AccessElementFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Create</td>
<td>CreateFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Dimensions</td>
<td>DimensionsFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Mem Access</td>
<td>MemoryAccessFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Mem Set</td>
<td>MemorySetFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Read File</td>
<td>ReadFileFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Set Element</td>
<td>SetElementFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Write File</td>
<td>WriteFileFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>RMatrix Write File (tabular)</td>
<td>WriteFileTabFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
</tbody>
</table>
## 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rmax</td>
<td>MaxFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Rmin</td>
<td>MinFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Round</td>
<td>FloatRoundToInt</td>
<td>Conversion</td>
</tr>
<tr>
<td>Rsqrt</td>
<td>SqrtFloat</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>RVect Access Element</td>
<td>AccessElementFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Create</td>
<td>CreateFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Length</td>
<td>LengthOfFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem +/-</td>
<td>MemoryAddFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem +/- With Real Output</td>
<td>MemoryAddFVectorWithFloatOut</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Access</td>
<td>MemoryAccessFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Change Length</td>
<td>MemoryChangeLengthFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Init</td>
<td>MemoryInitFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Largest in Range</td>
<td>MemoryLargestInRangeFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Length</td>
<td>MemoryLengthFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Set</td>
<td>MemorySetFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Mem Smallest in Range</td>
<td>MemorySmallestInRangeFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Read File</td>
<td>ReadFileFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Set Element</td>
<td>SetElementFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>RVect Write File</td>
<td>WriteFileFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>S == Switch</td>
<td>EnumIsEqualSwitch</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>S==</td>
<td>EnumIsEqual</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>Sconst</td>
<td>EnumConst</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>Search Hash</td>
<td>SearchHash</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Select Field</td>
<td>SelectFieldDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Service</td>
<td>Service</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service (Basic)</td>
<td>ServiceBasic</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service (Dimensioned)</td>
<td>ServiceDimensioned</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service (Priority)</td>
<td>ServicePriority</td>
<td>ServerResources</td>
</tr>
<tr>
<td>Service Timer</td>
<td>ServiceTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Set Alarm</td>
<td>SetAlarm</td>
<td>Timers</td>
</tr>
</tbody>
</table>
## 11.2 Converting BONeS Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET Rangen</td>
<td>EnumRanGen</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>SET to INTEGER</td>
<td>EnumToInteger</td>
<td>EnumOperations</td>
</tr>
<tr>
<td>Simple Counter</td>
<td>SimpleCounter</td>
<td>Counters</td>
</tr>
<tr>
<td>Simple FIFO</td>
<td>SimpleFIFO</td>
<td>Queues</td>
</tr>
<tr>
<td>Simple LIFO</td>
<td>SimpleLIFO</td>
<td>Queues</td>
</tr>
<tr>
<td>sin(x)</td>
<td>Sin</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Sink</td>
<td>BlackHole</td>
<td>Sinks</td>
</tr>
<tr>
<td>Split String</td>
<td>SplitString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>SR Last Dimension Modified</td>
<td>SrLastDimensionModifies</td>
<td>No MLD Category found !</td>
</tr>
<tr>
<td>SR State</td>
<td>SrState</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>SR State of Dimension</td>
<td>SrStateOfDimension</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>SR State of Modified Dimension</td>
<td>SrStateOfModifiedDimension</td>
<td>ServerResources/Internals</td>
</tr>
<tr>
<td>Start Timer</td>
<td>StartTimer</td>
<td>Timers</td>
</tr>
<tr>
<td>Str == C ?</td>
<td>StringEqualsConst</td>
<td>Switches</td>
</tr>
<tr>
<td>Str&lt;</td>
<td>LtString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&lt;=</td>
<td>LeString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&gt;</td>
<td>GtString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str&gt;=</td>
<td>GeString</td>
<td>Comparison</td>
</tr>
<tr>
<td>Str==</td>
<td>EqString</td>
<td>Comparison</td>
</tr>
<tr>
<td>String Concat</td>
<td>ConcatString</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Constant</td>
<td>ConstStringGen</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Length</td>
<td>StringLength</td>
<td>StringOperations</td>
</tr>
<tr>
<td>String Search</td>
<td>FindOccurrence</td>
<td>StringOperations</td>
</tr>
<tr>
<td>STRING to IMatrix</td>
<td>StringToIMatrix</td>
<td>MatrixOperations/IntMatrix</td>
</tr>
<tr>
<td>String to Int</td>
<td>StringToInt</td>
<td>StringOperations</td>
</tr>
<tr>
<td>STRING to IVect</td>
<td>StringToIVector</td>
<td>VectorOperations/IntVector</td>
</tr>
<tr>
<td>String to Real</td>
<td>StringToFloat</td>
<td>StringOperations</td>
</tr>
<tr>
<td>STRING to RMatrix</td>
<td>StringToFMatrix</td>
<td>MatrixOperations/FloatMatrix</td>
</tr>
<tr>
<td>STRING to RVect</td>
<td>StringToFVector</td>
<td>VectorOperations/FloatVector</td>
</tr>
<tr>
<td>Switch</td>
<td>Switch</td>
<td>Switches</td>
</tr>
<tr>
<td>Switch 4-Way</td>
<td>4WaySwitchConst</td>
<td>Switches</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Synchronize</td>
<td>2InOutSynchronize</td>
<td>Control</td>
</tr>
<tr>
<td>T &gt; Startup ?</td>
<td>TNowGtStartup</td>
<td>Switches</td>
</tr>
<tr>
<td>T==</td>
<td>TypeIsEqual</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>TableLookup</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>tan(x)</td>
<td>Tan</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Terminate Simulation</td>
<td>TerminateSimulation</td>
<td>Control</td>
</tr>
<tr>
<td>Throughput</td>
<td>Throughput</td>
<td>Statistics</td>
</tr>
<tr>
<td>Time Average</td>
<td>TimeAverage</td>
<td>Statistics</td>
</tr>
<tr>
<td>Time Between Triggers</td>
<td>TimeBetweenTriggers</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>TNow</td>
<td>TNow</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>TNow &gt;= Param ?</td>
<td>TNowGeStartup</td>
<td>Switches</td>
</tr>
<tr>
<td>Triangle (a,b,c) Rangen</td>
<td>TriangleRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Triangle (a,b,c) Rangen - Param</td>
<td>TriangleRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>TRUE</td>
<td>LogicTrue</td>
<td>Logic</td>
</tr>
<tr>
<td>True N Times</td>
<td>TrueNTimesConst</td>
<td>Switches</td>
</tr>
<tr>
<td>Truncate</td>
<td>FloatFloor</td>
<td>Conversion</td>
</tr>
<tr>
<td>TStop</td>
<td>StopTimeGen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Type Compatible</td>
<td>TypeIsCompatible</td>
<td>DSHandling</td>
</tr>
<tr>
<td>TYPE Const</td>
<td>TypeConst</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Type Switch</td>
<td>TypeSwitchDS</td>
<td>DSHandling</td>
</tr>
<tr>
<td>Uniform Pulse Train</td>
<td>UniformPulseTrain</td>
<td>Sources</td>
</tr>
<tr>
<td>Uniform Pulse Train (variable Inter-Pulse Time)</td>
<td>UniformPulseTrainVarInterPulseTime</td>
<td>Sources</td>
</tr>
<tr>
<td>Uniform Rangen</td>
<td>UniformRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Uniform Rangen - Param</td>
<td>UniformRangenConst</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>Up/Down Counter</td>
<td>UDCounter</td>
<td>Counters</td>
</tr>
<tr>
<td>Up/Down Counter - Change Value</td>
<td>VarUDCounter</td>
<td>Counters</td>
</tr>
<tr>
<td>Update Stats</td>
<td>UpdateStats</td>
<td>Queues/Internals</td>
</tr>
<tr>
<td>User_CDF Rangen</td>
<td>UserCDFRangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>U[0,1) Rangen</td>
<td>UniformGe0L1Rangen</td>
<td>NumberGenerators</td>
</tr>
<tr>
<td>BONeS Primitive</td>
<td>MLD Primitive</td>
<td>MLD Category</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Vector Access Element</td>
<td>AccessElementVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Change Length</td>
<td>ChangeLengthVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Create</td>
<td>CreateVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Length</td>
<td>LengthOfVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Access</td>
<td>MemoryAccessVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Init</td>
<td>MemoryInitVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Length</td>
<td>MemoryLengthVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Mem Set</td>
<td>MemorySetVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Vector Set Element</td>
<td>SetElementVector</td>
<td>VectorOperations/General</td>
</tr>
<tr>
<td>Weighted General Moments</td>
<td>WeightedGeneralMoments</td>
<td>Statistics</td>
</tr>
<tr>
<td>Weighted Mean and Variance</td>
<td>WeightedMeanAndVariance</td>
<td>Statistics</td>
</tr>
<tr>
<td>WrapUp</td>
<td>WrapUp</td>
<td>Control</td>
</tr>
<tr>
<td>Write Address Table to Info</td>
<td>DisplayAddressTable</td>
<td>AddressMapping</td>
</tr>
<tr>
<td>Write Error (NUMERIC)</td>
<td>WriteErrorFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Error (STRING)</td>
<td>WriteErrorString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Error (STRING) - Param</td>
<td>WriteErrorStringConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (INTEGER)</td>
<td>FileWriteInt</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (REAL)</td>
<td>FileWriteFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (ROOT-OBJECT)</td>
<td>FileWriteDS</td>
<td>Filehandling</td>
</tr>
<tr>
<td>Write File (STRING)</td>
<td>FileWriteString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write File (STRING) - Param</td>
<td>FileWriteStringConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Global Memory</td>
<td>GlobalMemoryWrite</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Write Info (NUMERIC)</td>
<td>WriteInfoFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Info (STRING)</td>
<td>WriteInfoString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Info (STRING) - Param</td>
<td>WriteInfoStringConst</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Memory</td>
<td>MemoryWrite</td>
<td>MemoryAccess</td>
</tr>
<tr>
<td>Write Warning (NUMERIC)</td>
<td>WriteWarningFloat</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Warning (STRING)</td>
<td>WriteWarningString</td>
<td>FileHandling</td>
</tr>
<tr>
<td>Write Warning (STRING) - Param</td>
<td>WriteWarningStringConst</td>
<td>FileHandling</td>
</tr>
</tbody>
</table>
## 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>BONeS Primitive</th>
<th>MLD Primitive</th>
<th>MLD Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ^ Iconst</td>
<td>PowConstInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Xor</td>
<td>Logic</td>
<td>Logic</td>
</tr>
<tr>
<td>X^I</td>
<td>PowInt</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>X^Y</td>
<td>PowFloat</td>
<td>Arithmetic</td>
</tr>
</tbody>
</table>

Table 11.48: Table of BONeS primitives in alphabetical order
11.3 COSSAP Conversion Tool

MLDesign Technologies, Inc. realized that many users want interoperability with other tools and need to convert design Intellectual Property (IP) such as primitives, modules, models and data files they have developed using other tools, to a format compatible with MLDesigner. Because MLDesigner does not have the ability to execute these models in their legacy format, or perform conversions during simulation runs, we developed a tool to convert existing project libraries, developed for use with other tools, to MLDesigner types enabling the incorporation of these elements into MLDesigner systems and simulations.

Using the Convert COSSAP tool, found in the main File menu, you can convert block diagrams, generic C++, generic C, C primitive files developed using COSSAP to the Synchronous Data Flow (SDF) domain in MLDesigner. With version 2.2 Fortran to C (FtoC) conversion is also possible. The code is first converted to C and then imported. This process can be done on a central computer and the converted libraries can then be copied to all work stations with a licensed version of MLDesigner. This is possible without an installed version of COSSAP on the computer where the conversion is being performed. All models in the project library are converted to MLDesigner format i.e., XML and .pl and all naming conventions are standardized to MLDesigner types (if one exists). In cases where an equivalent type does not exist in MLDesigner, you will be prompted to provide an equivalent primitive.

**NOTE:** MLDesign Technologies, Inc. makes no claims about the accuracy of the conversion utilities. We recommend that users validate converted files before using them in critical simulations. We also recommend the individual responsible for the conversion process be skilled and capable of performing the validation.

11.3.1 Prerequisites and Limitations

Our file conversion utilities do not produce 100% results all the time and are provided with the following disclaimers:

- It is possible that some user input or intervention may be required. We recommend that individuals performing file conversion be skilled in system modeling and that they have access to complete documentation of the files to be converted.
- MLDesign Technologies, Inc. does not guarantee the accuracy of the file conversion utilities. Users must validate all converted files before using them in sensitive systems.
- Not all blocks available in the legacy tool are available in MLDesigner. Not all the blocks in a legacy model may be available in MLDesigner libraries.

11.3.2 How to Convert COSSAP Project Libraries

To start the conversion process you need a COSSAP project archive and MLDesigner installed on your system. Expand the File menu and select Convert COSSAP to open the Convert COSSAP Model dialog (see fig. 11.2).
11.3.3 The User Mapper File

The first time a project is converted an entry is made in the user mapper file if there is no equivalent entry in the `import.mapper` file found in `$MLD/import/COSSAP`. The user mapper file is located in `~/.mld` and lists the user models as well as their location.

If a project to be converted contains models which are listed in the user mapper you have the choice of whether to use the existing model or to re-convert the model. Check the **Prefer user mapper entries** check box to use previously converted items. The decision to use the user mapped entries must be made before the library is read. If you change your decision after the project has been read, be sure to reload the project by clicking the double arrow icon.

11.3.4 Prefer User Mapper Entries and Overwrite Existing Files

Using both the **Prefer user mapper entries** and **Overwrite existing file** options simultaneously only makes sense where previously converted models have changed outside the MLDesigner environment.

![Convert COSSAP Dialog](image)

**Figure 11.2: Convert COSSAP Dialog**

The button labeled with the single arrow is used to revert the file `models.ltbl` found in the directory `d` to its original state. This file is modified during conversion to include references to missing libraries if there are any. This is explained in more detail later.

**NOTE:** The file is not removed if it did not exist prior to conversion; only the contents of the
file are deleted. This should not cause any problems as an empty file does not cause any overhead when the conversion tool is reading a project directory.

11.3.5 Reading Process

While a project is being read a log entry is created in the conversion dialog. This entry reads

Reading <Project name ...>

If a [+] is in front of this entry with an exclamation mark, you must expand the entry to read the errors or warnings generated during the reading process (see fig. 11.3). In front of each entry is either a yellow or a red exclamation mark. The yellow exclamation mark represents a warning that is unlikely to cause problems, and can normally be ignored. Log entries with a red exclamation mark, however, indicate that the error must be addressed if the system or module is to be functional after conversion.

NOTE: Dependency checking is only possible with project libraries and not with standard COS-SAP libraries. This is due to the structure of the library. A project library has a .mtbl file in the d directory which describes where all modules are saved.

Often a red exclamation mark refers to missing dependencies. A module or primitive needed for a system or module to function is stored in another location and not in the directory you are presently converting. These log entries read

There are some models referring to other Libraries ......

Double click this entry to open the dialog where all missing dependencies are displayed. Next to each entry is a check box showing the library name with the missing model in parenthesis.

![Figure 11.3: Log Entries after Reading a Project](image)

To add a library to the list to be converted, check the corresponding check box (see fig. 11.4) and click the folder icon to the right of the input box. A dialog displays with the title bar set to the name of the file that needs to be included. The file has the extension .mtbl. Once all libraries you wish to include are selected click the Accept button. The file models.ltbl found in directory d is modified to include references to the missing libraries.

NOTE: You must have write permission for the directory otherwise the models.ltbl file cannot be modified. If the file does not exist then it is created.

Pressing the Accept button reloads the project and missing dependencies or warnings - if there are any errors - are once again generated in the log window. It is possible to repeat the procedure described here until you are satisfied that all modules or primitives needed are included.
11 Import/Conversion of Models

Figure 11.4: Missing Libraries Dialog

All items in the project folder as well as dependent libraries are now displayed in the Library window of the dialog. The System Schematics are listed in the left window of the dialog. A single click on the Select All button ensures that all elements and dependent libraries, needed for the System Schematics to run, are converted. To convert all modules in the project directory you must click in the box next to the All Libraries entry in the Libraries window. You will notice the amount of items read differs to the amount of items selected.

NOTE: There are some limitations regarding the deselection of items in the System Schematics window. Deselection is performed from the top down and items selected in the Libraries window are not deselected when the System Schematic is deselected.

Click the Convert button to begin the conversion process. A progress bar shows the status of the conversion process.

11.3.5.1 Recommendation

We recommend you convert projects that contain all modules and primitives needed for the system schematic to run. This ensures the libraries have their own root entry in $MLD_USER. A partial conversion on the other hand leads to permanent model searching once the conversion process is over.

11.3.5.2 Log Entries

For every warning a yellow “!” and for every error a red “!” is displayed in the conversion log window. The yellow exclamation marks can usually be ignored; the red ones will mostly lead to incomplete models that cannot be executed after conversion.

After closing the conversion dialog, all log entries are also visible in the MLD designer Log window.
11.3 COSSAP Conversion Tool

NOTE: The amount of errors and warnings displayed depends on the **Settings/Console/View** configuration. If **Show errors** and **Show warnings** are checked, all log entries will be visible.

A click on a log entry in the MLDesigner Log window after conversion will open the relevant item in the Model Editor Window. If the entry refers to a primitive or a module the relevant item is highlighted in the Model Editor Window. If the log entry refers to a system the system will be opened in the Model Editor Window. The same behavior applies to the **Convert COSSAP model** dialog Log window.

You can convert the entire library by clicking the **All libraries** check box. The following components are converted or generated by MLDesigner.

- System schematics
- User-defined libraries
- Hierarchical models
- Primitives.

### 11.3.6 Conversion Process

There are two mapper files that contain a list of all COSSAP primitives and models that have MLDesigner equivalents. The `import.mapper` file located in `$MLD/import/COSSAP` is write protected and must not be altered. Located in your directory, `.mld` is a file called `cos-sap.mapper`. When importing a COSSAP module, all primitives called or referenced in the module are parsed and compared with those listed in the mapper file. If there are primitives that are not listed in the mapper file, an entry is created and the primitive is imported.

If there are inconsistencies regarding port names a warning message is displayed. If a referenced primitive is not listed in the mapper file, or it is not located in the project directory, an error message will be displayed.

### 11.3.7 Conversion of Schematics.

There are two types of schematic in COSSAP. They are converted to MLDesigner types as follows:

- System schematics, converted to MLDesigner systems;
- Hierarchical models, converted to MLDesigner modules.

The conversion process is achieved using a mapper file containing a list of COSSAP models and available MLDesigner equivalents. This mapper file contains entries of most of the models in **DSP**, **MOD63**, **COD_MOD** and **MATRIX COSSAP** libraries. Every instance of a model which is included in the mapper file is converted to the corresponding MLDesigner model type.

The following sub-headings describe elements that are converted to MLDesigner types on import.
11 Import/Conversion of Models

11.3.7.1 Parameters

Parameters in schematic files have a type and a default value. Types are:

- Integer,
- Real, or
- String.

Parameters in the .sch file are parsed prior to conversion and compared with those in the .mdef file. If their types differ, a warning message is displayed.

11.3.7.2 Expressions

Mathematical expressions and functions used as default values.

+ - * / 

11.3.7.3 Functions

- ** power function;
- sqrt(x) square root of x;
- real(x) typecasting from integer to real, like (float)x;
- sqr (x) x squared;
- log (x) logarithm with base 10;
- int (x) typecasting to integer, like (int)x;
- min (x,y) minimum value of x and y;
- max (x,y) maximum value of x and y;
- abs (x) absolute value of x;
- sign(x) sign of x;
- sin (x) sine function of x;
- cos (x) cosine function of x;

11.3.7.4 Variables

Specified as follows:

$Variable1
$Variable2

11.3.7.5 Ports

The following items are specified for ports

- Function of the port- input/output, multiple or single.
- Name of the port.

The type of port is also defined as:
11.3 COSSAP Conversion Tool

11.3.7.6 Model Instances

The imported schematic file includes the entire model or system and describes the position of the ports, parameters and their relations. This information is represented as an XML file with the extension .mml for all graphical representations of primitives, modules and systems.

11.3.7.7 Nodes

COSSAP schematic files contain descriptions of Nodes which are equivalent to MLDesigner relations.

11.3.7.8 Assignment Files

Assignment Files which are used for specifying values for the parameters of system schematics (system parameters) are also supported. An assignment file is divided into one or more separate sections. Each section contains information for different sets of values for system parameters. Assignment files are converted to MLDesigner system parameters including parameters specified as dataset identifiers.

11.3.8 Model Definition File

This file is an inseparable part of every COSSAP model. It is a text file containing specific information describing the appearance of the COSSAP model. The MLDesigner import module parses the entire .mdef file collecting all available information.

The sections which can be converted into equivalent MLDesigner items are described here.

11.3.8.1 NAME

Defines the name of the model.

11.3.8.2 DESCRIPTION and SHORT_DESCRIPTION

Provide a comprehensive description and one-line description of the model. These are also used to generate online documentation in HTML format as well as for the generation of tool tip text.
11 Import/Conversion of Models

11.3.8.3 IMPLEMENTATIONS

Defines the names of available implementations for the current model. Each implementation is written in a separate file. There are two main groups:

- BLOCK DIAGRAM - specifies a hierarchical model, schematic;
- Other types - specifying primitive models.

Current version converts the following implementations:

- block diagram (.sch)
- generic C (.gc) and generic C++ (.gcc)
- primitives defined with C implementation (.c)

Import module allows only one implementation of a given model to be selected. You can either choose the C code or the C++ instance. This is done by expanding the model directory in the Libraries panel in the Physical View. You can choose between C and generic C. We recommend you stay with the default setting which always select the generic C code in favor of normal C code implementations.

11.3.8.4 PARAMETERS

Defines the parameters of the model. Each parameter has a name and type. It can also have a default value.

Supported Types are in C/C++:

( I ) INTEGER int
( R ) REAL float
( S ) STRING const char *

Every parameter can have a description of multiple lines.

11.3.8.5 INPUT_PORTS and OUTPUT_PORTS

These sections define the input/output ports of the model.

Supported port types are:

INTEGER
DOUBLE INTEGER
REAL
COMPLEX
DOUBLE

For input ports there is also an optional value that specifies the input data rate for a DSP or VHDL model. This applies only to SDF models and can be an integer value or expression of parameters. A model can have either fixed or variable number of I/O ports. Each I/O port can have a description of multiple lines.
11.3.9 History

The .mdef file contains the following information. This information is available in the converted primitive.

- Release and version number
- Date
- Author

11.3.10 Declarations

Support for keywords:

```
INPUT_PORT
OUTPUT_PORT
RATE
PARAMETER
INPUT_DATASET
LENGTH
STATE
CONST
BLOCKFACTOR
INCLUDE
```

11.3.11 Functional code

This section is parsed and converted into an MLDesigner primitive file with a .pl extension. The user defined error codes described in the .mdef file are supported also.

11.3.12 Dataset Handling Library

This is a special dynamic library included in the kernel of SDF domain. It provides the interface between MLDesigner primitive modules and dataset files. It can also be used for developing new primitives using standard COSSAP dataset file formats. Input and output datasets of the following type are supported:

```
REAL - float
INTEGER - long int; int
STRING - char **
BIT_VECTOR - long int; int (B(n downto m)) (Motorola Convention)
```

11.3.13 Dataset Parameters

Generic C files use only input datasets. You can use parameters for every dataset to specify the filename and file format. All ASCII formats are supported. The dataset files are copied during conversion to the corresponding MLDesigner library. The dataset file type is passed directly to the corresponding primitive. For every model containing datasets the following parameters are
11 Import/Conversion of Models

included: Identifier, ChartFormat, ReadMode (Implicit/Explicit), FileType, and FileName.

The following dataset parameters are imported:

11.3.13.1 DS_NAME

It is the dataset’s name.

11.3.13.2 Input/Output Datasets

These parameters start with ids (input dataset) or ods (output dataset) followed by

• FileName - this is the full path to the input/output dataset’s filename, from which you want to read (input dataset) or to which you want to write (output dataset).
• Identifier - this is set automatically during import. It is used for internal purposes and is normally set to zero.
• ChartFormat - this displays the format of the plot you want to use for your dataset. This is generally used by COSSAP and currently not supported by MLDesigner, because most imported models have as plotting primitive our re-implemented Dump primitive (DmpInt, DmpFloat) that has its own FORMAT parameter. Usually the value of ChartFormat is 0, otherwise it is set during import.
• ReadMode/WriteMode - 1 for implicit and 0 for explicit (see sec. 11.3.14).
• FileType - the following data formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>ascii format</td>
</tr>
<tr>
<td>ASCII_MATRIX</td>
<td>ascii format for matrices</td>
</tr>
<tr>
<td>ASCII_LF_EOL</td>
<td>ascii line feed end of record</td>
</tr>
<tr>
<td>ASCII_BL_EOL</td>
<td>ascii blank line end of record</td>
</tr>
<tr>
<td>RAW_8BIT</td>
<td>raw binary 8 bit</td>
</tr>
<tr>
<td>RAW_16BIT</td>
<td>raw binary 16 bit</td>
</tr>
<tr>
<td>RAW_32BIT</td>
<td>raw binary 32 bit</td>
</tr>
<tr>
<td>RAW_U8BIT</td>
<td>unsigned raw 8 bit</td>
</tr>
<tr>
<td>RAW_U16BIT</td>
<td>unsigned raw 16 bit</td>
</tr>
<tr>
<td>RAW_U32BIT</td>
<td>unsigned raw 32 bit</td>
</tr>
</tbody>
</table>

11.3.14 EXIT_FLAG

We now support the EXIT_FLAG specified in the .mdef file of primitives. An additional parameter called EndCondition for the appropriate primitive is generated. The parameter values in the Model Properties Editor show this parameter as a boolean of type int with the value set to Yes/No. A new line in the .pl file’s setup method will be generated

```c
if ((bool)EndCondition) willRequestEnd();
```
as interface to the kernel.
Readmode and Writemode

When you click on a dump primitive in a system, you have the option to set the output mode to explicit (0) or implicit (1). Setting the ReadMode to 0 results in a file containing the results of the last simulation being saved in the location specified in the input field of the parameter [P] ods_OUT_DATA_FileName.

To do this click the icon to the right of the [P] ods_OUT_DATA_ReadMode input field in the Instance Properties Window and set the value to 0. Make sure you have write permission for the selected directory. MLDesigner will only be able to generate the file if the directories in the path you entered exist.

11.3.15 Unsupported Features

The following are object code file tables parsed by the converter, but not supported by MLDesigner.

11.3.15.1 Graphical elements

Arc - defined with coordinates beginning and center;
Line - defined with coordinates beginning and end;
Circle - defined with coordinates center and radius;
Rectangle - defined with coordinates top left and bottom right.

11.3.15.2 Symbol files

Files .sym are not parsed at the moment. Each symbol file provide the appearance of the model. The same graphical elements used in schematic files (lines, arcs a.s.o.) are used there. The symbol file also contains the parameters of the model, datasets, ports a.s.o.

11.3.16 Parsing Model Definition File

11.3.16.1 CLASSES

This is a list of classes to which a model belongs. COSSAP models are organized in a structure of classes. Each model can be a member of one or more classes. If the class name is not specified it is assumed as the model library name.

11.3.16.2 INPUT_PORTS and OUTPUT_PORTS

Port Type BIT_VECTOR - can be

- \((B (n \text{ downto } m))\) - Motorola convention
- \((B (m \text{ to } n))\) - Intel convention.

Variable number of input ports - an arbitrary number of input ports (at least 1), with an identical number of output ports. Variable number of output ports - an arbitrary number of output ports (at least 1), with an identical number of input ports.
11.3.16.3 HDL_INPUT_PORTS

This section defines the CLOCK and RESET ports of an HDL model. This information is required only by the HDL Code Generator (VCG). These ports are of type STD_LOGIC. A model can have only one RESET and one CLOCK input port. This section is part of STD_LOGIC which is not supported currently by MLDesigner.

11.3.16.4 Dataset

Output Dataset Chart Type:

- LINE2 Line chart;
- BAR_15, BAR_1 Bar chart, positive values only;
- BAR_28, BAR_2 Bar chart, positive and negative values;
- HISTOGRAM_16, HISTOGRAM_1 Histogram, positive values only;
- HISTOGRAM_29, HISTOGRAM_2 Histogram, positive and negative values;
- STATISTICS Text;
- TOWER10 Tower chart;
- SCATTER11 Scatter diagram;
- EYE_PATTERN12 Eye pattern diagram;
- STAIRCASE13 Staircase diagram;
- DIGITAL_32_BIT14 Up to 32-bit digital diagram;

For each chart format, text strings can be defined to label the dataset in the graphics output. The number of text strings depends on the chart type and the model.

11.3.17 Conversion of Primitive Models

The following languages are not supported (with corresponding filename extensions):

- C ++ implementation (.cc)
- VHDL IEEE 1076 (.vhdl)
- Verilog (.v)

These implementations are not selectable in the Import COSSAP Model dialog.

Dataset Handling Library

The following binary format is not supported.

BINARY - binary format;

11.3.18 Limitations with COSSAP Project Conversion

If a primitive or model is not included in both the mapper file and the COSSAP project folder, its instance is not created. A warning is displayed to keep you informed. This error normally occurs when a primitive referenced or called by a system or module is located in an external directory and the directory path is no longer valid.
11.3 COSSAP Conversion Tool

11.3.18.1 Error Handling

Error Handling for imported systems are problematic because the errors often refer to Kernel internal functions of the COSSAP kernel. This normally arises where the switch `init` is as follows:

```c
switch(init) if(init==1)
```

System error codes are specified with negative values, e.g.

```c
ExitOnError (-1)
```

There are some COSSAP internal errors handled by MLDesigner. All known COSSAP errors have negative values. Unknown errors display the default error message "Unknown error in 'ExitOnError' ".

11.3.18.2 C Code Limitations

It is not possible to import COSSAP internal header files unless you have physically copied the file to your Project library or you have COSSAP installed on your system. This topic is untested and could lead to segmentation Faults.

11.3.19 Input Dataset File Formats

This section explains the ASCII* dataset formats in case it is necessary to write the file by hand.

11.3.19.1 ASCII

A record header indicates the start of a record and also the end of the previous record. The header specifies the number and the type of the record elements. The last record in the dataset ends with a generic header in which the number of elements is set to -1.

The following example shows an input dataset that contains floating point values grouped in two records:

```
# 10 (R)
0.00000000000000e+00
1.00000000000000e+00
2.00000000000000e+00
3.00000000000000e+00
4.00000000000000e+00
5.00000000000000e+00
# 10 (R)
0.00000000000000e+00
-1.00000000000000e+00
-2.00000000000000e+00
-3.00000000000000e+00
-4.00000000000000e+00
-5.00000000000000e+00
```
The matrix size is given by the number of lines and columns in the record, while the records are delimited by a blank line.

In this example the dataset contains two floating point matrices of sizes 4x2 and 3x3 respectively:

\[
\begin{bmatrix}
0.00000000000000e+00 & 1.00000000000000e+00 \\
2.00000000000000e+00 & 3.00000000000000e+00 \\
4.00000000000000e+00 & 5.00000000000000e+00 \\
6.00000000000000e+00 & 7.00000000000000e+00 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.00000000000000e+00 & -1.00000000000000e+00 & -2.00000000000000e+00 \\
-3.00000000000000e+00 & -4.00000000000000e+00 & -5.00000000000000e+00 \\
-6.00000000000000e+00 & -7.00000000000000e+00 & -8.00000000000000e+00 \\
\end{bmatrix}
\]

In this dataset format, each line represents a record because the record separator is an LF character. The following example lists an input dataset with floating point values placed in three records of three, one and two elements, respectively:

\[
\begin{bmatrix}
1.00000000000000e+00 & -2.00000000000000e+00 & 3.00000000000000e+00 \\
4.00000000000000e+00 & 5.00000000000000e+00 & 6.00000000000000e+00 \\
\end{bmatrix}
\]

The records are separated by a blank line in this dataset format and each line contains only one value.

Here is listed a dataset with three records containing floating point values:

\[
\begin{bmatrix}
1.00000000000000e+00 \\
2.00000000000000e+00 \\
3.00000000000000e+00 \\
4.00000000000000e+00 \\
5.00000000000000e+00 \\
6.00000000000000e+00 \\
7.00000000000000e+00 \\
8.00000000000000e+00 \\
9.00000000000000e+00 \\
\end{bmatrix}
\]
## 11.3.20 Troubleshooting Guide for Cossap Model Converter

This section contains a list of possible error and warning messages that may display during or after conversion and how to avoid or deal with them.

### 11.3.20.1 Errors During Conversion.

This table contains a list of possible errors and warnings generated while converting COSSAP models.

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory &quot;DIR&quot; is missing</td>
<td>The directory doesn’t contain the following structure</td>
<td>Warning can be ignored as long as directories contain .mdef files. If no .mdef file is included they are unreadable.</td>
</tr>
<tr>
<td></td>
<td><code>projectdir -c</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>- symbol</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>- schematic</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>- d</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>- symbol</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>- schematic</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>- dspgeneric</code></td>
<td></td>
</tr>
<tr>
<td>No valid implementations</td>
<td>Appears when f2c conversion fails if a function name was specified in the .mdef file (implementation) and isn’t used in the corresponding source file e.g.: IMPLEMENTATIONS 1 SOURCE_CODE GC myFileName (myFuncName) In the .gc file the names of the functions are now init_func1() func1() post_func1()</td>
<td>func1 and myFuncName differ→error message if NO function name was specified, the function used in implementation file (c, gc) has to be the same as the filename. This also happens if:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• gc code is used in a c file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• schematic files have an unsupported format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• implementation file was not found (gc, c, F, sch)</td>
</tr>
<tr>
<td>Cannot open file. . .</td>
<td>File not found or no read permissions</td>
<td></td>
</tr>
</tbody>
</table>
## 11 Import/Conversion of Models

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WARNING!</strong> Port name/Parameter name differs ...</td>
<td>These names are identified by their index numbers so if the same index number has different names in source and .mdef file, this message appears.</td>
<td>Names defined in the .mdef file have higher priority and replace names in converted source code. Name replacement may lead to inconsistencies if names were modified only in source file and additional states/members are implemented with the names specified in .mdef file. <strong>USUALLY</strong> this warning can be ignored!</td>
</tr>
<tr>
<td>Unable to find include file</td>
<td>included files will be copied to imported library directory cossap includes will be ignored</td>
<td>files don’t exist include files need to be found in mdef directory or source file directory</td>
</tr>
<tr>
<td><strong>WARNING!</strong> Mismatched library of schematic</td>
<td>Usually in line 2 of .sch file the library name is not specified or differs from the library directory name where the schematic file is situated. If names differ, they will be ignored, but this may lead to an incorrect mapper entry in / .mld/cossap.mapper for this model</td>
<td><strong>USUALLY</strong> this can be ignored, but it’s better for consistency to correct this name or copy the model to the correct library (the one specified in the .sch file)</td>
</tr>
<tr>
<td>Schematic file has unsupported format → undefined node</td>
<td>There seems to be an unconnected port/node in this model or a missing or wrong netobject for a specific port example: p portname &quot;&quot; &quot;&quot; &quot;&quot; 0 correct: p portname &quot;&quot; SIG_X &quot;&quot; 0 (SIG_X is the netobject the port belongs to)</td>
<td>If there are more such errors in the same file, only the 1st one will be reported</td>
</tr>
<tr>
<td>Library table “d models.ltbl” file has unsupported format in line:</td>
<td>There is probably an empty line or linefeed at the end of the specified file</td>
<td>This is only a warning and can be ignored</td>
</tr>
<tr>
<td>Unknown parameter type in schematic</td>
<td>The type of the parameter is undefined in the schematic. This happens when a placeholder is used ? instead of I, R or S example:parm myParamName ? &quot;&quot; correct:parm myParamName I &quot;&quot;</td>
<td>The ? as a placeholder for anytype is not supported</td>
</tr>
</tbody>
</table>
11.3 COSSAP Conversion Tool

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model definition file has unsupported format or is corrupted in line...</td>
<td>After a tag the listed items (parameters, ports, implementations ...) need to be numbered. <strong>Example:</strong> .CLASSES myclass1 myclass2 <strong>Correct format:</strong> .CLASSES 1 myclass1 2 myclass2</td>
<td></td>
</tr>
</tbody>
</table>

Table 11.49: Possible Errors During COSSAP Model Conversion
11.3.20.2 Messages After Conversion

Once the conversion process is completed, the following errors or warnings may be encountered.

<table>
<thead>
<tr>
<th>Error Dialog</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corresponding port PORTNAME to instance MXXX in module MODULE of library RFMODEL cannot be found. Before making ....</td>
<td>In the schematic file MODULE, the instance MXXX has a port name called PORTNAME, but the port names of the corresponding model (usually a primitive) differ from the used one. Probably there are some additional port name specifications in a file called MODULENAME.v_arc or MODULENAME.v_ent, that COSSAP can evaluate and therefore doesn’t display any warning.</td>
<td>This is definitely an inconsistency and needs to be fixed. Either change the name in the MODULENAME.sch or in the MXXX.mdef file. In the latter case, the MXXX model needs to be converted again before converting the MODULE</td>
</tr>
<tr>
<td>Warning! There is a port naming inconsistency ...</td>
<td>The port names of a converted primitive/module and it’s instance in a system or module differ. <strong>Example:</strong> primitive Add has port name output (as specified in .mdef file). The module’s schematic or .v_arc file has probably been modified and the port was renamed to output. This message appears after conversion, so take a look at the converted model and check all connections. If all ports are connected, the converter was able to determine the correct ports, so there are no problems anymore. Prior to conversion, a message like &quot;model is not up to date&quot;, is displayed during simulation of system using COSSAP.</td>
<td>Re-generate the model’s schematic and .v_arc file so that the names of the instance are equal to the port names of the primitive/module (nearly 100% chance to reconvert without any problems depending on port name inconsistencies)</td>
</tr>
</tbody>
</table>

Table 11.50: Common Errors after COSSAP Model Conversion
Chapter 12

Data Structure Management

12.1 Managing Data Structures

MLDesigner supports the definition of complex data types. The difference between MLDesigner data types and the original base types is that you can define arbitrary structured data types as well as special user defined data types, such as enumerations. For more detailed information about the structure and programming of MLDesigner data structures refer to the Programming Guide/Using Data Types.

NOTE: To differentiate between base data types and MLDesigner data types, the latter are principally called Data Structures (DS).

Data structure types are managed in the Data Editor found in the top right corner of the MLDesigner user interface. The data type editor can be blended in or out using the Data Type View On/Off tool button in the standard toolbar.

The structure of the data type editor is shown in fig. 12.1. A tree in the upper region shows the hierarchy of data structures. The window below displays members or enumeration elements of the currently selected data structure. The type of data structure is indicated by different icons for structured types and base types. The icon color indicates whether they are editable or not.

A green icon means that a data structure is editable, i.e., it is defined in a writable library. Red icons indicate read-only data structures.

NOTE: User-defined Data Structures are defined as part of a library and are only available when the library in which they were created is open in the Model Editor Window.

You will notice three logical items in the data structure tree. These are expandable items with only a + and no red or green icon. These items serve to group structures of the same type as indicated by the name of the folder or sub-folder. These types are:

- Base Types - groups base type data structures such as Float and Integer
- Vector groups Vector data structures
12.1.1 Creating Data Structures

To create a new DS Activate the context menu in the Data Structure tree view or click on the Create New Data Structure icon on the toolbar. If at least one editable library is open in the design area the context menu item New Data Structure is enabled. Select it to open the Data Structure Editor dialog (see fig. 12.7). This dialog is used to create and edit data structures of type:

- composite structures
- enumerations
- vectors.

The data structure type can be selected from the drop-down menu Kind. On selecting a type from the drop-down list, certain input fields change dynamically. Refer to sec. 12.3.1.2 through sec. 12.3.3 for details on the differing elements. The Name, Library and Description fields are common for all data structures with description being the only optional field. The Name must be a valid identifier. The library combo box lists all open editable libraries. Choose a location to create
the new data structure in. Once the data structure properties are defined, click the Save button.

### 12.1.2 Adding Composite Members

To add a new member to a composite data structure, open the context menu by right-clicking anywhere in the lower section while the data structure is selected in the upper sections tree and choose Add Member.

**NOTE:** Note that this menu item, just like the delete item, will only appear if the data structure is editable.

[Figure 12.2: Add Member dialog]

The blank member editor dialog will open (see fig. 12.2). Fill in a name (must be a valid identifier) and give a description for the new member (optional). By default, Root is set as the type of a new member. This can be changed by choosing from a tree of data structures using the tool button next to the type name field. If a base type is chosen as member type, the Default Value field becomes editable, and only if a numeric base type is used as member type, the Subrange field is activated. The subrange field is used to define an interval of valid values for numeric members, given in the usual mathematical notation with the keyword Inf (case insensitive) meaning infinity.

**Examples:**

* (-Inf,2) an interval with no bottom limit and the upper limit being 2, excluding 2 itself
* (-Inf,4) an interval with no bottom limit and the upper limit being 4, including 4
* [3.45, 9.3) 3.45 is included, 9.3 is excluded

From the drop-down menu in the Subrange field, you can select an entry from a list of commonly used subranges. After all fields are filled, click the Save button to save the new member in the library defining the parent data structure.
12 Data Structure Management

12.1.3 Editing Composite Members

To edit a composite data structure member click with the right mouse button on the member to open the context menu fig. 12.3). Choose Edit to open the member editor shown in fig. 12.4 for this member.

Figure 12.3: Data structure member context menu

NOTE: If the data structure is not editable, only a View item will be available which opens the same editor without editing capabilities.

NOTE: Since elements can only be edited in the composite data structure of their origin, inherited elements are not editable and have a grey background in the Add Member dialog.

Figure 12.4: Data Structure Member editor

12.1.4 Deleting Composite Members

If the permissions allow for modifying the parent data structure, you can delete composite data structure member using the item Delete of the context menu for members as shown in fig. 12.3. After confirming a warning message, the member is removed.

NOTE: Note that the library is saved immediately and there is no way to undo deleting a member
12.2 Managing Enumeration Elements

To view or edit elements of an EnumType, use the lower section of the data structure view. Here elements will be listed when an EnumType is selected in the data structure tree. Since elements can only be edited in the EnumType originally defining them, inherited elements are not editable and shown with a grey background. Managing enumeration elements works very similar to managing members of composite data structure members in that it is done through the context menu in the lower section. Right-click in the list to open it.

12.2.1 Adding Enumeration Elements

To add a new element to an editable EnumType, open the context menu anywhere in the lower section while the EnumType is selected in the tree of data structures. Choose Add Element to open the element editor, see fig. 12.4. The Index field defaults to the next available index, but it can be set to any positive whole number including 0. The Value field must be set to a valid identifier. Click the save button to add the new element. If an element with this index or value already exists in the EnumType, you will now be asked to confirm overwriting of the existing element(s).

12.2.2 Editing Enumeration Elements

Open the context menu by right-clicking on the element in the list and choose Edit. The element editor will be shown. For details on this editor, see sec. 12.2.1.

NOTE: The editor will open but will be inactive if the Library where the element is saved is write protected.

12.2.3 Deleting Enumeration Elements

To delete an element, right-click on it in the list and choose Delete from the context menu. This item will only be available for editable enumeration types. When the warning dialog is confirmed, the element will be removed.

NOTE: The library is saved immediately and there is no way to undo deleting an element.

12.3 Data Structure Handling Mechanism

A number of changes have been made to the way MLDesigner handles data structures significantly improving the performance of systems that use them. From Version 2.3 data structures consume much less memory and execution is much faster. Here is a brief look at some of the changes.

- Data structure member values can now be defined by expressions.
- Data structures cannot be renamed if they have derivatives.
- Data structure classes have been split into two classes
  - Type - value of the data structure
  - TypeClass - descriptor for a data structure.
12 Data Structure Management

- Data structure primitive methods have been modified.
- The operator [ ] in Vector returns a clone. This could lead to possible memory leak in old code.
- Cast and assignment operators have been added in Memories. It is now possible to assign a memory directly to a data structure TypeRef, or a data structure to a memory.
- List data structures are now supported.
- A new library called ListOperations has been added in the DE Domain.

The sections of this chapter which have CHANGED or are NEW with Version 2.3 are marked with an icon as seen here.

### 12.3.1 Overview of Data Structures

Data structures are entities for manipulating data in and between modules. If you want to define and use complex data structures, scalar types might not provide sufficient functionality. The Data structure window is a complex of classes which tries to simulate as best as possible the variety of data types. The need for them arises when you want to send complex data structures over input and output ports. These mechanisms are also used by Memories and Events to store data and to read new values from and write new values to them.

There are libraries of useful primitives that can handle data structures. These libraries are described briefly in sec. 12.4.

You can declare both input and output ports as well as parameters as type datastruct.

![Image]

Figure 12.5: Data Type Selection for Input/Output Ports

The Data Structure property is visible in the Port Properties editor for ports defined as type DataStruct. A click on the green icon in the input field opens the Select Data Structure dialog. Here the parent type for the data structure must be selected. It is possible to add new members to
12.3 Data Structure Handling Mechanism

a data structure if there is no suitable type for your purposes. This is done in the Add Members window (bottom right) via the appropriate context menu option.

![Select Data Structure Window]

Figure 12.6: Data Structure Selection for Input/Output Ports

12.3.1.1 Type Checking

When you connect the ports of primitives and modules you should be careful because MLDesigner provides only a basic checking mechanism concerning compatibility of data structure types. No error will appear when you connect incompatible data structure ports. You must take care to connect ports that output and receive the same data structure. To keep compatibility with the scalar type, connecting a data structure port with an integer port or a float float port is allowed. It is, however, necessary to check if the data structure port type outputs or receives a data structure of type Integer or Float.

12.3.1.2 Creating a Composite Data Structure

When selecting Composite as the kind to be created, the parent type must be specified. The default is Root. If this is unchanged a top-level data structure will be created. To create a subdirectory of an existing data structure, click the button to the right of the Parent Type input field and select a data structure from which the new data structure should inherit certain properties.

12.3.2 Creating an Enumeration

Enumeration types are similar to composite types in that they have a parent. By default, this parent is Root.ENUM. To change this, open the list of existing enumerations by clicking the button next
to the Parent Enum field and select an enumeration from which the newEnumeration should inherit.

12.3.3 Creating a Vector

Vectors have no parent, instead it is required that you set their element types. Click the tool button next to the right of the Vector Type field and choose a data structure from the hierarchy.

12.3.4 Editing Existing Data Structures

To make changes to an existing data structure, open the context menu over that data structure in the tree. The uppermost menu item will be "View or Edit, depending on whether or not that data structure, or more precisely, the library the data structure is defined in, is editable. Both items open the data structure editor which allows the data structures properties to be viewed or edited (see sec. 12.1.1).

12.3.4.1 Deleting Data Structures

To delete a data structure, again open the context menu over it and choose Delete. Note that this item is enabled only if file permissions allow deleting. A warning message will appear asking to confirm the action, and a click on Delete will now remove the data structure from the library. Note that the library is saved immediately and there is no way to undo deleting a data structure!

12.3.4.2 Managing Composite Members

To create, edit and delete members of a composite data structure, use the lower region of the data structure view. When a composite data structure is selected in the upper tree region, the lower region displays a list of members defined for that data structure. Members can only be edited in the data structure they are originally defined in. Thus inherited members are shown in the upper
region, with a grey background, indicating they are not editable. Similar to management of the
data structures themselves, members are edited using the context menu. Right-click the mouse in
the lower region to open the context menu.

### 12.3.5 Import Libraries

Every data structure belongs to an XML Library. There are non editable system libraries which
are loaded every time you start MLDesigner and it is possible to define new libraries and add your
own defined data structures to a library.

The **Import Libraries** parameter field is normally filled in automatically and is a reference
to the location where data structures or shared model instances such as memories or events used
in a particular system are stored.

**NOTE:** These user defined data structures are only visible and available while the library within
which they were defined is open in the Model Editor Window.

A major change in the way MLDesigner handles data structures was introduced with version 2.3.
The data structure mechanism splits each of the old data structure classes in two different classes.
The first class describes its type information such as name and inheritance. The other class is
specific to the data structure instance and contains the value information.

The name of the type information class always ends with the **class** suffix (e.g.: DataStructureClass,
VectorClass...). Objects of this class can be changed during the design phase and these changes
modify the structure of data structures. During the simulation phase no changes over instances
of this kind are allowed. The other type of classes represents the value part of a data structure.
Instances of this classes are used in simulation process.

Every data structure value object holds a reference to its **class** object which should be unique for
each defined data structure. This brings improvements in two directions: it uses less memory,
putting all the common information in a single instance and it brings also speed improvements:
duplicating a value object takes less time due to its smaller size and less member variables that
needs to be copied.

A third type of data structure class that represents a reference to a value object exists. This is the
TypeRef class, the base class for all data structure references, and it defines almost all the methods
different data structures can implement. More specific reference classes have the letter \( R \) as the
last letter in their name. Their purpose is to handle the creation, copying and releasing of data
structure values that are to be used in simulation. These classes save you from all the overheads
of cloning and dieing data structure, making primitives programming much easier.

Data structures are organized in a hierarchy with Root as the parent of all data structures. Every
data structure must have a parent, and this can be Root or another data structure of the same type.
You can extend a composite data structure from Root or from another composite data structure
and this means that all its members will be inherited by the newly created data structure (see dia-
gram 12.8). The same is also true for enumerations, in which case the enumeration will inherit all
elements from its parent.

The data types Integer and Float and the classes Matrix and Complex are included under MLDesigner Data Structures here for compatibility reasons.

![MLDesigner Data Structures Diagram]

Figure 12.8: Data Structure Hierarchy

For information on creating new data structures see sec. 12.1.1.

12.3.6 Data Structure string representation

Every data structure is characterized by a Name, a Full Name and a Unique Name. Name is simply the name of the data structure (ex.: TCPProtocol). Full Name consists of the parents’ names and the data structure name separated by periods. (ex.: Root.NetworkProtocol.TCPProtocol). The Unique Name is composed of Library Name and Full Name separated by colon (ex.: SystemDS:Root.NetworkProtocol.TCPProtocol).

It is possible to save and to set values of a data structure from a string. This is called the string representation of a data structure. The syntax of this is composed of the UniqueName and a string that represents its value between curly braces. This value string differs depending on the type of the data structure, and will be described separately for each data structure. Based on this string representation it is possible to set default values for parameters of Data Structure types, Memories and Events.
12.3 Data Structure Handling Mechanism

12.3.7 Data Structure Types

Data Structure types in MLDesigner are grouped by their functionality in three branches as follows:

- Base data structures
- Vector data structures
- Composite data structures

12.3.7.1 Base Data Types

These data structures are used for operations with integers, floats or strings. If you want to use these types you must consider that they are classes which simulate scalar type, and systems which use them run slower compared to those that use only scalar types. Using them makes sense only if they are part of a system which contains complex data structures.

The following data structures are provided as basic types. The class name is in parenthesis:

- `Integer` (IntType) - used to manipulate integer scalar type
- `Float` (FloatType) - used to manipulate float scalar type
- `String` (StringType) - used to manipulate strings
- `ENUM` (EnumType) - for operations that use enumerations.
- `List` (ListType) - represents a list of data structures
- `BitVector` (BitVector) - this data structure is a vector of bits. The elements can take values only 0 and 1.
- `MVL4BitVector` (MVL4BitVector) - data structure to represent the multi valued logic standard. Elements can have values 0, 1, X, Z.

Numerical data structures (integer and float) support base mathematical operations such as addition, subtraction, multiplication and division. In Enumerations values are saved as strings, and as elements they have a special class that contains index and value for each element. That’s why EnumType provides various methods to get it’s elements as strings or as element objects.

Value as string for a base data structure in the string representation is composed of its value surrounded by curly braces (ex.: `Root.Integer{123}`).

12.3.7.2 Vector Data Structures

For Vectors two kinds of data structures exist: Numeric and Generic Vectors. For every vector you can set or change the length. The generic Vector can hold any kind of MLDesigner data structures but if you need to save only scalar values it is much faster to use numeric vectors, as they are optimized for this usage (see fig. 12.9).

12.3.7.3 Numeric Vectors

- `IntVector` - a vector of integer values
- `FloatVector` - a vector of float values
For the vectors that hold numerical values it is possible to set a default value that all elements will hold on initialization.

For Numeric Vector types String value must contain the length followed by a colon and the vector’s elements. If an element is repeated consecutively times you can specify between [ ] how many times is repeated. The last value in the string is multiplied as a default value for all the elements that remained unset.

Example:

```
{Root.IntVector{10:123}}
```

vector with 10 elements all elements set to 123.

```
```

will create the vector

8 8 8 8 4 23 23 23 23 23

**12.3.7.4 Generic VECTOR**

This is a vector of data structures. This generic Vector is implemented as a vector of pointers to data structure objects, so you can store any type of data structures. On initialization, this vector has all elements set to NULL, and you have to set every element one by one.

It is possible to derive the generic vector which means you define a new vector that at initialization has all elements set to the defaults of a data structure previously specified. As a string representation it is possible for a generic vector to specify a vector’s length together with the data structure
name to initialize all elements.

Example:

```
   {Name,{192,168,0,1},{192,168,0,2},110,110}}
```

means a vector of ten TCPProtocol elements.

### 12.3.7.5 Composite data structure

This type of data structure can be used to model complex data and it can have any kind of data structure types as members. Working with this might not be so intuitive. If you want to access a member you first have to search for it by its name, and then get the data inside.

In case of composite data structures, values for members are listed in the order they exist in the data structure, separated by commas. This applies recursively to composite data members.

Example:

```
SystemDS:Root.NetworkProtocol.TCPProtocol{Name,
   {192,168,0,1},{192,168,0,2},110,110}
```

It is important to preserve the members’ order in this string representation, because the values are assigned based on this order. For more information on how to use composite data structures see sec. 15.2.

### 12.4 Data Structure Libraries

MLDesigner provides three libraries containing primitives for handling or manipulating data structures. These are:

- DSHandling (DE and SDF)
- EnumOperations (DE only) and
- VectorOperations (DE only)

Operations performed on data structure might include

- Access or Modify modules
- Create pre-defined data structures
- Extract and insert values of fields, and
- Coerce data structure to another type.
12.4.1 DSHandling Library

The following primitives are found in the DE and SDF domain library. A brief description of their function is here but you can find out more by looking at the online documentation or the source code of each primitive.

- **AddFieldParamDS/AddFieldValueDS**
  Increment/decrement a particular field of the Input data structure. The field must be of a numerical type.

- **CoerceDS**
  Coerce a data structure coming on the input to another specified type.

- **CreateDS**
  Primitive used to instantiate a data structure.

- **CreateParamDS**
  Instantiate a data structure with the values specified in the parameter.

- **DSToFloat**
  Converts a Float base data structure to a scalar type float.

- **DSToInt**
  Converts an Integer base data structure to an integer scalar type.

- **DSToString**
  Converts a base data structure to a string.

- **DeclareDS**
  Tests if an input data structure is of a specified type.

- **InsertFieldDS**
  A data structure arriving on an input port gets inserted as a field into a composite data structure.

- **InsertFieldParamDS**
  Inserts a data structure specified in a datastruct parameter as a field into a composite data structure.

- **PrintDS**
  Primitive used to print on the standard output the values of a data structure.

- **SelectFieldDS**
  Selects a field from a composite data structure and outputs it on a port.

- **SetFieldValueDS**
  Primitive to set the value of a field of a data structure. The field must be of a base type, and the field name and the value are provided by the dsFieldName, and dsFieldValue parameters.

The following primitive is found in the library DE domain/DSHandling only. As the name of the primitive implies, this primitive outputs a time. The SDF domain has no notion of time so this primitive has no place in the SDF domain.

- **InsertFieldTNowDS**
  Sets a selected field of the Input data structure with the current execution time and then places the modified data structure on the Output port.

The *Data Structure Type Operations* primitives check the types of data structures. You can use these to differentiate between data structures of different types, so that MLDesigner can operate on them separately. Type compatibility checks can also be performed.
12.4 Data Structure Libraries

- **TypeConst**
  Puts on the output port a specified data structure.
- **TypeIsCompatible**
  Tests if two incoming data structures are compatible.
- **TypeisEqual**
  Tests if two incoming data structures are equal.
- **TypeSwitchDS**
  The incoming data structure is placed on one of the two outputs depending on its compatibility with the specified type.

12.4.2 EnumOperations Library

Primitives in the *EnumOperations* directory perform operations on data structures of type enumeration.

- **EnumConst**
  Generates an enumeration data structure with a user provided value.
- **EnumIsEqual**
  Tests if two incoming data structures of enumeration type are equal or not.
- **EnumIsEqualSwitch**
  Switch an incoming data structure between two output ports base on the values of an enumeration.
- **EnumRanGen**
  Generates an enumeration with a random set value.
- **EnumToInteger**
  Takes an enumeration coming on the input port and outputs the index of its value.
- **EnumToString**
  Takes an enumeration coming on the input port and outputs the value as a string.
- **IntegerToEnum**
  Generates an enumeration data structure. The name is specified as parameter and the value is the index coming on the input port.

12.4.3 VectorOperations Library

The *Vector Operations* primitives provide access to vectors and matrices. Three kinds of vector primitives are available for handling:

- integer vectors
- float vectors and
- generic vectors.

*Integer* and *Float* vectors can only contain integers and floats respectively. For *generic* vectors you can insert as elements, any data structure type including vectors, to simulate matrix operations. Provided are also primitives to write to and access vectors and elements of vectors that are stored in memories.

Operations you can execute using vectors and primitives are:
12 Data Structure Management

- **AccessElementVector**
  outputs the element at the specified position in the vector coming on the input port.
- **CreateVector**
  creates a vector data structure with the specified length.
- **IndexOfValueVector**
  search for a specified value in a range of elements in vector and returns the index of the first occurrence.
- **LengthOfVector**
  outputs the length of the vector.
- **MemoryAccessVector**
  outputs the element at the specified position in a vector stored in a Memory
- **MemoryAddVector**
  increments by one an element at the specified index in an Integer or Float Vector
- **MemoryAddVectorWithIntOut**
  increments by one an element at the specified index in an Integer or Float Vector and the new value is placed on the output port.
- **MemoryChangeLengthVector**
  change the length of a vector stored in a Memory.
- **MemoryLargestInRangeVector**
  outputs the element with the greatest value in an Integer or Float vector that is stored in a Memory.
- **MemoryLengthVector**
  outputs the length of a vector stored in a Memory.
- **MemorySetVector**
  sets an element at the specified index in a vector stored in a Memory.
- **MemorySmallestInRangeVector**
  outputs the element with the lowest value in an Integer or Float vector that is stored in a Memory.
- **ReadFileVector**
  reads a vector from a file and places it on the output port.
- **SetElementVector**
  sets an element at the specified index in a vector coming on the input port.
- **StringToVector**
  given a string of integer or float values, constructs a corresponding Integer or Float vector and places it on the output port.
- **WriteFileVector**
  writes the incoming vector to a specified file.
Part II

Programming Guide
Chapter 13
Designing Primitives

13.1 Introduction

Before reading this section you must be familiar with the Graphical User Interface (GUI) and know how to create libraries, models and model components. These topics are covered in detail in Modeling with MLDesigner sec. 2.3.

There are different ways to create models or model components using MLDesigner.

- By building a model hierarchically from existing modules using the Model editor. Such a model is called hierarchical module or simply module.
- By creating a Finite State Machine (FSM) representation of the model using the FSM editor. Such a model is called FSM module or simply FSM.
- By defining the external interface of the model using the Model editor and defining the functionality using the Source code editor. Such models are called primitive modules or simply primitives.

**NOTE:** The Ptolemy language is a preprocessor language that allows the designer to use C++ code to define the functionality of primitives. Files written in the Ptolemy language have the extension .pl and contain primitive source code.

The Ptolemy vocabulary differs from that of MLDesigner. The definitions and differences in terminology can be seen in table 13.1. The advantages of each class of models are shown in table 13.2.

Comprehensive libraries of primitive models for the more mature domains are supplied with MLDesigner. These primitives were designed to be as generic as possible and many complex functions can be realized using primitives contained in these libraries. There will, however, always be a need to develop new primitives. You can link the new primitives dynamically or (if the primitives use shared variables) add them to the list of compiled-in primitives by selecting the Load Mode as permanent.

The following steps are necessary when defining a primitive:

1. create the primitive,
2. define the external interface of the primitive using the standard Model editor,
3. define the primitive functionality using the Ptolemy language ptlang, and
4. compile and load the primitive (dynamic or permanent).
### 13 Designing Primitives

<table>
<thead>
<tr>
<th>MLDesigner type</th>
<th>Description</th>
<th>Ptolemy type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>The lowest level model in MLDesigner, with functionality defined in the Ptolemy language. Contains C++ code fragments.</td>
<td>Star</td>
</tr>
<tr>
<td>Module</td>
<td>A model made up of connected primitives and/or embedded modules, with input and/or output ports.</td>
<td>Galaxy</td>
</tr>
<tr>
<td>System</td>
<td>A combination of primitives and modules with connected ports within a model. A complete system that can be executed/simulated.</td>
<td>Universe</td>
</tr>
<tr>
<td>Parameter</td>
<td>Interface element for definition of initial values. In primitives parameters can be used to remember values.</td>
<td>State</td>
</tr>
<tr>
<td>Domain</td>
<td>The model of computation, which defines the behavior of a network of models. In code generation, a domain also corresponds to single target language.</td>
<td>Domain</td>
</tr>
</tbody>
</table>

Table 13.1: Definition of models in MLDesigner

### 13.2 Definition of Primitive Interfaces

After creating a new primitive, MLDesigner automatically opens a **Model Editor Window** containing the empty interface model of the primitive. However, if the primitive was created as a copy of an existing one, you must explicitly open it in a **Model Editor Window**. See ch. 3 for more detail on how to open a model within a **Model Editor Window**. Now, you can use the **Model Editor Window** to define the primitive interface. The toolbar for defining the primitive interface can be seen in fig. 13.1. The external interface model of a primitive consists of:

- model property definitions;
- input/output port definitions;
- parameter definitions;
- annotations or text labels.

![Figure 13.1: Toolbar for primitive interface models](image-url)
13.2 Definition of Primitive Interfaces

<table>
<thead>
<tr>
<th>Modules</th>
<th>FSM models</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>- no knowledge of C/C++ required</td>
<td>- well suited to controller design and protocol specification</td>
<td>- required for fundamental operations</td>
</tr>
<tr>
<td>- automatic consistency / error checking</td>
<td>- suited to certain levels of design abstraction</td>
<td>- often more efficient</td>
</tr>
<tr>
<td>- understandable</td>
<td>- easier to create than primitives</td>
<td>- preexisting C++ simulation modules can be used</td>
</tr>
<tr>
<td>- self-documenting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- easy to modify and extend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- built-in debugging tools can be used</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 13.2: Advantages of Modules, FSM models, and Primitives

13.2.1 Model Property Definitions

As with all hierarchical modules, you can define common model properties of a primitive using the **Property Editor Window**(fig. 13.2). You can define the following model properties:

- **Logical Name**
- **Load Mode**
- **Description**
- **Documentation**
- **Copyright**, and
- **Version**

For each model property, MLDesigner creates an item entry in the primitive source file. The **Logical Name** is used to identify the model component in the **Library View**. The **Logical Name** is shown in the headline of the **Model Editor Window** and is a free text string.

13.2.2 Load Mode

The **Load Mode** property determines whether a primitive is loaded dynamically or permanent. If a primitive is loaded dynamically, a shared library is created that is linked dynamically to MLDesigner. Dynamically linked primitives can be replaced or reloaded anytime. If a primitive is loaded permanent, the primitive is linked statically to MLDesigner. Such primitives are handled like built-in primitives and are added to the built-in primitive list.

Linking a primitive permanently is necessary if the primitive defines resources like variables or functions that are shared with other primitives. To ensure that all symbols of shared resources are known at link time, primitives which define the shared resources have to be loaded before primitives which use these resources. To ensure this prerequisite, you have to set the load mode
13 Designing Primitives

![Property Editor](image)

Figure 13.2: **Property Editor** for primitive example

for these primitives to permanent. If you open a hierarchical model that contains instances of primitives with permanent load mode, these primitives are loaded automatically.

**NOTE:** Changes to permanently linked primitives only become effective when you shut down and restart MLDesigner.

The common model properties *Model Type* and *Domain* are not editable.

### 13.2.3 Input/Output Port Definitions

Use the tool buttons **Add Input Port** and **Add Output Port** to create an arbitrary number of input and output ports. Please refer to ch. 3.4 to get more detailed informations on how to create input/output ports. In contrast to hierarchical modules, where you can define only single input and single output ports, it is possible to define multiple input and multiple output ports for primitives. For each port definition, MLDesigner generates an entry in the primitive source code upon saving. These port items cannot be changed in the primitive source file, but must be changed using the **Property Editor**. Select the port object and change the values of the port properties.

### 13.2.4 Parameter Definitions

For the definition of primitive parameters you have to use the **Property Editor**. As for any hierarchical module, you can create additional properties that define model parameters. Use the menu item **New Parameter** in the context menu of the **Property Editor** to create a new parameter. The parameter is created with default values for all parameter elements. You can define the following elements

- the parameter name,
- the parameter type,
- the default value of the parameter,
- the description of the parameter, and
13.3 Primitive Functionality Definition

- the attributes of the parameter.

For detailed information on the type and significance of parameter elements see sec. 13.5.4. For each defined parameter MLDesigner generates on saving a defparameter item in the primitive source file. These items cannot be changed in the primitive source file; they must be changed using the parameter properties of the primitive interface model.

13.2.5 Annotations

You can place annotations (plain text labels) anywhere within the model background. Use the tool button Add Text Label to create a new text label. You can define the text, the color, the font size, and the justification of the text label. After creation you can change an annotation anytime by selecting the text label and modifying the label properties using the Property Editor Window.

13.3 Primitive Functionality Definition

After defining the primitive interface model, save your changes. MLDesigner then generates the primitive source file that contains items for all model properties and definitions.

If there is already a primitive source file, the primitive properties and definitions that are defined by the primitive interface model are replaced by the properties of the new primitive interface model. All the other definitions like existing functionality are preserved.

The generated primitive source file serves as a template for the definition of the functionality. The primitive source files are written in the Ptolemy language and are identified by the extension .pl.

The Ptolemy language is described in detail in section 13.4.

The Ptolemy language has several language constructs for the definition of the primitive interface. All these constructs are generated using the information from the primitive interface model. Furthermore, the Ptolemy language has some constructs for the definition of methods that describe the functionality of the primitive. These methods are executed at different stages in the life cycle of primitive instances. For example, the Ptolemy language allows for the definition of methods that are executed at:

- instance creation and deletion;
- simulation start-up time;
- during simulation.

The functionality of these methods is defined using C++ code. The Ptolemy language only defines the method structure.

You can edit the primitive source file using the primitive editor. To edit the source code choose Open Source from the context menu or click the Open Source icon on the toolbar.

The primitive editor provides syntax highlighting and all additional source code generated by changes made to the primitive within the Design window, will be formatted automatically (see fig. 13.3). After making your changes, save the file using the Save icon. Since the primitive editor is simply a text editor, changes to the primitive source file do not affect the primitive interface model directly. Changes to open primitives using the MLDesigner GUI are visible in the Source code editor on saving.
13.4 Ptolemy Language Description

The Ptolemy language ptlang is a preprocessor language and was created to make it easier to write and document primitives to run under MLDesigner. Instead of writing all the class definitions and initialization code required for a primitive, the programmer can concentrate on writing the functionality of a primitive and let the preprocessor generate the standard initialization code for ports, parameters, etc. The preprocessor generates standard C++ code, divided into two files, a header file with a .h extension and an implementation file with a .cc extension. It also generates standardized documentation in a file with a .htm extension.

13.4.1 Compiling Primitives

The definition of a primitive named Yyy in domain Xxx should appear in file with the name XxxYyy.pl. The class that implements this primitive will be named XxxYyy. MLDesigner automatically uses the command ptlang XxxYyy.pl to invoke the preprocessor. The preprocessor will produce the files XxxYyy.cc, XxxYyy.h, and XxxYyy.htm in the directory of the primitive source file. The preprocessor does not attempt to parse the parts of the language that consist of C++ code, e.g., the methods. For these parts, it simply counts curly braces to find the ends of the items in question. It generates #line directives so the C++ compiler will print error messages, if any, with respect to the original source file.
13.4.2 Example

To make things more clear consider this example:

1. select the Library tab in the tree view window.
2. open the MLD Libraries/SDF Domain/Nonlinear library.
3. click on the Sin item. From the context menu choose Save As to open the Save as New Model dialog.
4. set the Logical name and Physical Name to MySin.
5. select a writable library in which to save the copied primitive.
6. click the OK button to save the primitive.

You now have a sine function primitive that is described by the file SDFMySin.pl, MySin.mml and SDFMySin.htm as well as a makefile in the selected library. Running the preprocessor (compiling the primitive) produces the three files SDFMySin.h, SDFMySin.cc and SDFMySin.i386-linux.o. The names are determined not by the input filename but by concatenating the domain and name fields. These files define a class named SDFMySin.

At the time of this writing, only one type of declaration may appear at the top level of an MLDdesigner language file, a defprimitive, used to define a whole primitive. The defprimitive section itself is composed of subitems that describe various properties and definitions of the primitive.

All subitems are of the form keyword { body }, where the body may itself be composed of subitems, or may be C++ code, in which case the MLDdesigner language preprocessor checks it only for balanced curly braces.

NOTE: Keywords are not reserved words, they may also be used as identifiers in the C++ code body.
### 13.5 Primitive Language Constructs

The following items can appear in a `defprimitive` item. The items are given in the order in which they typically appear in a primitive source file, although they can appear in any order. An alphabetical listing and summary of items that are frequently used is given in table 13.3. There are additional items for code generation stars, they will be explained in later sections.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Summary</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledge</td>
<td>the names of other contributors to the primitive</td>
<td>no</td>
<td>13-11</td>
</tr>
<tr>
<td>author</td>
<td>the name(s) of the author(s) of the primitive</td>
<td>no</td>
<td>13-11</td>
</tr>
<tr>
<td>begin</td>
<td>C++ code to execute at start time, after the scheduler setup method is called</td>
<td>no</td>
<td>13-18</td>
</tr>
<tr>
<td>ccinclude</td>
<td>specify other files to include in the <code>.cc</code> file</td>
<td>no</td>
<td>13-20</td>
</tr>
<tr>
<td>code</td>
<td>C++ code to include in the <code>.cc</code> file outside the class definition</td>
<td>no</td>
<td>13-20</td>
</tr>
<tr>
<td>codeblock</td>
<td>define a code segment for a code-generation primitive</td>
<td>no</td>
<td>13-21</td>
</tr>
<tr>
<td>conscalls</td>
<td>define constructor calls for members of the primitive class</td>
<td>no</td>
<td>13-17</td>
</tr>
<tr>
<td>constructor</td>
<td>C++ code to include in the constructor for the primitive</td>
<td></td>
<td>13-17</td>
</tr>
<tr>
<td>copyright</td>
<td>copyright information to include in the generated code</td>
<td>no</td>
<td>13-11</td>
</tr>
<tr>
<td>cleanup</td>
<td>method to free all memory when a simulation ends</td>
<td>no</td>
<td>13-18</td>
</tr>
<tr>
<td>deffevent</td>
<td>define an event</td>
<td>no</td>
<td>13-13</td>
</tr>
<tr>
<td>deffmemory</td>
<td>define a memory</td>
<td>no</td>
<td>13-14</td>
</tr>
<tr>
<td>deffresource</td>
<td>define a resource</td>
<td>no</td>
<td>13-14</td>
</tr>
<tr>
<td>deffparameter</td>
<td>define a parameter</td>
<td>no</td>
<td>13-12</td>
</tr>
<tr>
<td>derived</td>
<td>alternative form of <code>derivedfrom</code></td>
<td>no</td>
<td>13-10</td>
</tr>
<tr>
<td>derivedfrom</td>
<td>the base class, which must also be a primitive</td>
<td>no</td>
<td>13-10</td>
</tr>
<tr>
<td>desc</td>
<td>alternative form of <code>descriptor</code></td>
<td>no</td>
<td>13-10</td>
</tr>
<tr>
<td>descriptor</td>
<td>a short summary of the functionality of the primitive</td>
<td>no</td>
<td>13-10</td>
</tr>
<tr>
<td>destructor</td>
<td>C++ code to include in the destructor for the primitive</td>
<td>no</td>
<td>13-17</td>
</tr>
<tr>
<td>domain</td>
<td>the domain, and the prefix of the name of the class</td>
<td>yes</td>
<td>13-9</td>
</tr>
<tr>
<td>explanation</td>
<td>full documentation (see also <code>htmldoc</code>)</td>
<td>no</td>
<td>13-11</td>
</tr>
<tr>
<td>execstime</td>
<td>specify the execution time for a code generation primitive</td>
<td>no</td>
<td>13-21</td>
</tr>
<tr>
<td>go</td>
<td>C++ code to execute when the primitive fires</td>
<td>no</td>
<td>13-18</td>
</tr>
<tr>
<td>header</td>
<td>C++ code to include in the <code>.h</code> file, before the class definition</td>
<td>no</td>
<td>13-20</td>
</tr>
<tr>
<td>hinclude</td>
<td>specify other files to include in the <code>.h</code> file</td>
<td>no</td>
<td>13-20</td>
</tr>
</tbody>
</table>
13.5 Primitive Language Constructs

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Summary</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>htmldoc</td>
<td>full documentation, optionally using HTML items</td>
<td>no</td>
<td>13-11</td>
</tr>
<tr>
<td>inmulti</td>
<td>define a set of inputs</td>
<td>no</td>
<td>13-15</td>
</tr>
<tr>
<td>inout</td>
<td>define a (bidirectional) input and output</td>
<td>no</td>
<td>13-15</td>
</tr>
<tr>
<td>input</td>
<td>define an input to the primitive</td>
<td>no</td>
<td>13-15</td>
</tr>
<tr>
<td>location</td>
<td>an indication of where a user might find the primitive</td>
<td>no</td>
<td>13-11</td>
</tr>
<tr>
<td>method</td>
<td>define a member function for the primitive class</td>
<td>no</td>
<td>13-20</td>
</tr>
<tr>
<td>name</td>
<td>the name of the primitive, and the root of the name of the class</td>
<td>yes</td>
<td>13-9</td>
</tr>
<tr>
<td>outmulti</td>
<td>define a set of outputs</td>
<td>no</td>
<td>13-15</td>
</tr>
<tr>
<td>output</td>
<td>define an output from the primitive</td>
<td>no</td>
<td>13-15</td>
</tr>
<tr>
<td>private</td>
<td>define private data members of the primitive class</td>
<td>no</td>
<td>13-19</td>
</tr>
<tr>
<td>protected</td>
<td>defined protected data members of the primitive class</td>
<td>no</td>
<td>13-19</td>
</tr>
<tr>
<td>public</td>
<td>define public data members of the primitive class</td>
<td>no</td>
<td>13-19</td>
</tr>
<tr>
<td>setup</td>
<td>C++ code to execute at start time, before compile-time scheduling</td>
<td>no</td>
<td>13-18</td>
</tr>
<tr>
<td>version</td>
<td>version number and date</td>
<td>no</td>
<td>13-10</td>
</tr>
<tr>
<td>wrapup</td>
<td>C++ code to invoke at the end of a run (if no error occurred)</td>
<td>no</td>
<td>13-19</td>
</tr>
</tbody>
</table>

Table 13.3: Summary of most important primitive items

13.5.1 Keywords in detail

13.5.1.1 name

This is a required item, and has the syntax

```plaintext
name { identifier }
```

Together with the domain, this provides the name of the class to be defined and the names of the output files. The identifier is case sensitive.

13.5.1.2 domain

This is a required item. It specifies the domain, such as SDF. The syntax is

```plaintext
domain { identifier }
```

where identifier specifies the domain (again, it is case sensitive).
13 Designing Primitives

13.5.1.3 derivedfrom

This optional item indicates that the primitive is derived from another class. The derived and base primitives must be defined in the same domain. Syntax:

    derivedfrom { identifier }

where identifier specifies the base class. The .h file for the base class is automatically included in the output .h file and the directory where the base class is located is added in the include section of the primitive’s makefile.

The derivedfrom statement may also be written derivedfrom or derived. See also page 13-18 for information regarding the go method for derived primitives.

13.5.1.4 descriptor

This item defines a short description of the class. It is displayed in a tooltip when you position the mouse cursor over the element in the Tree View window or over an instance. It has the syntax

    descriptor { text }

where text is simply a section of text that will become the short descriptor of the primitive. A principal use of the short descriptor is to get on-screen help, so the descriptor should not include any troff formatting commands. Unlike the htmldoc (described below), it does not pass through troff. The following are legal descriptors:

    desc { A one line descriptor. }
    desc {
    A multi-line descriptor. The same line breaks and spacing will be used when the descriptor is displayed on the screen.
    }

In these descriptors, inputs, references to the names of parameters (states), and outputs, should be enclosed in quotation marks. Also, each descriptor should begin with a capital letter, and end with a period. If the descriptor is too long, augment it with the htmldoc item, explained below.

13.5.1.5 version

This item contains two entries as shown below

    version { number mm/dd/yy }

where the number is the version number, and the mm/dd/yy is the version date. If you are using SCCS for version control then the following syntax will work well:

    version { %W% %G% }

When the file is checked in by SCCS, the string %W% will be replaced with a string of the form: @(#)filename num, where num is the version number, and %G% will be replaced with a properly formatted date.
13.5.1.6  author

This optional entry identifies the author or authors of the primitive. The syntax is:

    author { author1, author2, author3 }

Any set of characters between the braces will be interpreted as a list of author names.

13.5.1.7  acknowledge

This optional entry attaches an acknowledgment section to the documentation. The syntax is:

    acknowledge { arbitrary single line of text }

13.5.1.8  copyright

This optional entry attaches a copyright note to the .h, and .cc files. The syntax is:

    copyright { copyright information }

The copyright may span multiple lines, just like the descriptor. A typical copyright note is used as follows

    copyright {1990-1994 The Regents of the
                University of California}

13.5.1.9  location

This item describes the location of a primitive definition. The following descriptions can be used:

    location { SDF dsp library }

or

    location { directory }

where directory is the location of the primitive. Please note that this item is for documentation only.

13.5.1.10  explanation

This item is used to give longer explanations of the function of the primitives. This item is obsolete and has been superceded by the htmldoc item.

13.5.1.11  htmldoc

This item is used to give longer explanations, including HTML format directives. MLDesigner uses an HTML viewer (see sec. 3.16) to display primitive documentation.
13 Designing Primitives

13.5.1.12 defparameter

This item is used to define a state or parameter.

**NOTE:** In Ptolemy vocabulary a parameter is by definition only the initial value of a state. However, in context with MLDesigner state definitions are principally called parameter definitions since

1. state definitions specify the interface variables of the primitive as seen as parameters from outside,
2. changes of state variables during simulation are only visible inside the primitive.

Here is an example of a parameter definition:

```plaintext
defparameter
{
  name   { gain }
  type   { int }
  default { 10 }
  desc   { Output gain. }
  attributes { A_CONSTANT | A_SETTABLE }
}
```

There are five types of subitems that may appear in a parameter statement, in any order. The `name` field is the name of the parameter. The `type` field is its type, which may be one of `int`, `float`, `string`, `complex`, `fix`, `intarray`, `floatarray`, `stringarray`, `complexarray`, `fixarray`, `precision`, `boolean`, `enum`, `file`, `expression`, `datastruct`, and many others. Case is ignored for the type argument. The `default` item specifies the initial value of the parameter, its argument is either a string (enclosed in quotation marks) or a numeric value. The above entry could equally have been written:

```plaintext
default { "1.0" }
```

Furthermore, if a particularly long default is required, for example when initializing an array, the string can be broken into a sequence of strings. The following example shows the default for a `ComplexArray`:

```plaintext
{
  "(-.040609,0.0) (-.001628,0.0) (.17853 ,0.0)"
  "(.37665 ,0.0) (.17853 ,0.0) (-.001628,0.0)"
}
```

For complex parameters, the syntax for the default value is

```plaintext
(real, imag)
```

where `real` and `imag` evaluate to integers or floats. The `precision` parameter is used to give the precision of fixed-point values. These values can be other parameters or can be internal to the primitive. The default can be specified in either of two ways:
Method 1: As a string like "3.2", or more generally "m.n", where m is the number of integer bits (to the left of the binary point) and n is the number of fractional bits (to the right of the binary point). Thus length is m+n.

Method 2: A string like "24/32" which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one.

The desc (or descriptor) item, which is optional but highly recommended, attaches a descriptor to the parameter. The same formatting options are available as with the primitive descriptor.

Finally, the attributes keyword specifies parameter attributes. Two attributes are defined for all parameters: A_CONSTANT and A_SETTABLE (along with their complements A_NONCONSTANT and A_NONSETTABLE). If a parameter has the A_CONSTANT attribute, then its value is not modified by the run-time code in the primitive (it is up to you as the primitive writer to ensure that this condition is wanted). Parameters with the A_NONCONSTANT attribute may change when the primitive is run. If a parameter has the A_SETTABLE attribute, you can change the value of the parameter when you instantiate the primitive. Values of parameters without this attribute are not editable for primitive instances. Such parameters will always start with their default values as the initial value. If no attributes are specified, the default is A_CONSTANT|A_SETTABLE. Thus, in the above example, the attributes item is unnecessary. The notation A_CONSTANT|A_SETTABLE indicates a logical "or" of two flags. Confusingly, this means that they both apply A_CONSTANT and A_SETTABLE. Code generation primitives use a great number of attributes, most of them specific to the language model for which code is being generated. Mechanisms for accessing and updating parameters in C++ methods associated with a primitive are explained below, in sec. 13.5.4 and sec. 13.5.5.

An alternative form for the parameter item is state. The subitems of the parameter item are summarized in table 13.4, together with subitems of other items.

13.5.1.13 defevent

This item is used to define an event. Here is an example of an event definition:

```plaintext
defevent
{
    name    { Event1 }
    scope   { Internal }
    type    { Root }
    default { "{Root}" }
    code
    {
        
    }
}
```

There are five types of subitems that appear in an event statement, in any order. The name field is the name of the event. The scope field specifies whether the event is external (visible) or internal
13 Designing Primitives

(hidden). The type field is its type which can be any data structure type derived from Root or Root itself. The default value specifies the initial value of the event shared element that is used to if the event occurs. This item is only used in case of internal events. The syntax has to conform with declarations of data structure value (see ch. 12.3.5). The event definition can contain a code section that defines the code which is executed if the event occurs.

13.5.1.14 defmemory

This item is used to define a memory. Here is an example of a memory definition:

```plaintext
defmemory
{
    name   { Memory1 }
    scope  { Internal }
    type   { Root }
    default { "(Root)" }
}
```

There are four types of subitems that appear in a memory statement, in any order. The name field is the name of the memory. The scope field specifies whether the memory is external or internal, i.e., visible or hidden respectively. The type field is its type which can be any data structure type derived from Root or Root itself. The default specifies the initial value of the memory shared element. This value is used until a model instance will write data into the memory. The default item is only used in case of internal memories. The syntax has to conform with declarations of data structure value (see ch. 12.3.5).

13.5.1.15 defresource

This item is used to define a resource. Here is an example of a resource definition:

```plaintext
defresource
{
    name   { Resorce1 }
    type   { "Quantity" }
    scope  { External }
    code
    {
       // Arguments of the method
       //       int pSuccess
       //       QuantityTransaction* pTrans

       if (pTrans == NULL)
           return;
       if (pSuccess == 0)
           {
           }
       else if (pSuccess == 1)
           {
```
There are four types of subitems that appear in a resource statement, in any order. The name field is the name of the resource. The scope field specifies whether the resource is external or internal, i.e., visible or hidden respectively. The type field specifies whether the resource models a quantity or a server. It defines either the string Quantity or the string Server. The resource definition code defines the code which is executed whenever a transaction exits the resource. When you create a resource with MLDesigner, it creates a code subitem template as shown above automatically.

### 13.5.1.16 input, output, inout, inmulti, outmulti

These keywords are used to define a port also called porthole, which may be an input, output, inout (bidirectional) port or an input/output multiport. Bidirectional ports are not supported in most domains. Like parameters they contain subitems such as:

```plaintext
input
{
    name { signalIn }
    type { complex }
    num { 2 }
    desc { A complex input that consumes two input particles. }
}
```

Here, name specifies the port name. This is a required item. Subitem type specifies the particle type. The scalar types are int, float, fix, complex, message, string, file, datastruct, continuous, or anytype. Again, case does not matter for the type value. The matrix types are int_matrix_env, float_matrix_env, complex_matrix_env, and fix_matrix_env. The type item may be omitted; the default type is anytype. For more information on all of these, please see sec. 14. The numtokens keyword, also be written as num or numTokens, specifies the number of tokens consumed or produced on each firing of the primitive. This only makes sense for certain domains like SDF, DDF, and BDF. In such domains, if the item is omitted, a value of one is used. For primitives where this number depends on the value of a parameter, it is preferable to leave out the num specification and to have the setup method set the number of tokens (in the SDF domain and most code generation domains, this is accomplished with the setSDFParams method). This item is primarily used in the SDF and code generation domains, and is discussed further in the documentation of those domains.

There is an alternative syntax for the type field of a port. This syntax is used in connection with anytype to specify a link between the types of two ports. The syntax is

```plaintext
    type { = name }
```
where `name` is the name of another port. This indicates that this port inherits its type from the specified port. For example, here is a portion of the definition of the `SDFFork` primitive:

```plaintext
input
{
    name { input }
    type { anytype }
}

outmulti
{
    name { output }
    type { = input }
    desc { Type is inherited from the input. }
}
```

<table>
<thead>
<tr>
<th>Item</th>
<th>Subitem</th>
<th>Summary</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>inmulti, inout, input</td>
<td>name</td>
<td>name of the port or group of ports</td>
<td>yes</td>
<td>13-15</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>data type of input/output particles</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>number of tokens consumed by the port (useful only for data flow domains)</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>method, virtual method</td>
<td>name</td>
<td>the name of the method</td>
<td>no</td>
<td>13-20</td>
</tr>
<tr>
<td></td>
<td>access</td>
<td>private, protected, or public</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arglist</td>
<td>the arguments to the method</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>the return type of the method</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td>C++ code defining the method</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if not pure</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>outmulti, output</td>
<td>name</td>
<td>name of the port or group of ports</td>
<td>yes</td>
<td>13-15</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>data type of output particles</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>descriptor</td>
<td>summary of the function of the output</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>number of tokens produced by the port (useful only for data flow domains)</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>defparameter</td>
<td>name</td>
<td>the name of the parameter variable</td>
<td>yes</td>
<td>13-12</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>data type of the parameter variable</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>the default initial value, always a string</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
13.5 Primitive Language Constructs

<table>
<thead>
<tr>
<th>Item</th>
<th>Subitem</th>
<th>Summary</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>descriptor</td>
<td>summary of the function of the parameter</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>attributes</td>
<td>hints to the simulator or code generator</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Table 13.4: Summary of items that have subitems

13.5.1.17 constructor

This item allows the programmer to specify extra C++ code to be executed in the constructor for the class. This code will be executed after any automatically generated code in the constructor that initializes ports, parameters, etc. The syntax is

```
constructor { body }
```

where body is a piece of C++ code. It can be of any length. Note that the constructor is invoked only when the class is first instantiated. Actions that must be performed before every simulation run should appear in the setup or begin methods, not the constructor.

13.5.1.18 conscalls

You may want to have data members in your primitive that have constructors that require arguments. These members would be added by using the public, private, or protected keywords. If you have such members, the conscalls keyword provides a mechanism for passing arguments to the constructors of those members. Simply list the names of the members followed by the list of constructor arguments for each, separated by commas if there is more than one. The syntax is:

```
conscalls { member1(arglist), member2(arglist) }
```

Note that member1 and member2 should have been previously defined in a public, private or protected section, see page 13-19.

13.5.1.19 destructor

This item inserts code into the destructor for the class. The syntax is:

```
destructor { body }
```

You generally need a destructor only if you allocate memory in the constructor, begin method, or setup method. Termination functions that happen with every run should appear in the wrapup function \(^1\). The optional keyword inline may appear before destructor. If it is used, the destructor function definition appears inline, in the header file. Since the destructor for all primitives is virtual, this is only needed if the primitive is used as a base for derivation. See also cleanup.

\(^1\)Note, however, that wrapup is not called if an error occurs
13.5.1.20 cleanup

The cleanup method clears memory regardless of whether a simulation ends normally or abnormally. In situations where the user ends the simulation or where an error is displayed before the wrapup method is called, the cleanup code is executed. An example can be found on 13-34. The syntax is:

```plaintext
cleanup
{
    // delete variable(s)
}
```

If you free in cleanup dynamically allocated variables, make sure that those variables are initialized to zero in the primitive’s constructor.

13.5.1.21 setup

This item defines the setup method, which is called every time the simulation is started, before any compile-time schedule is performed. The syntax is:

```plaintext
setup { body }
```

The optional keyword inline may appear before the setup keyword. It is common for this method to set parameters of input and output ports, and to initialize parameters. The code syntax for this type of method is explained in sec. 13.5.2 on page 13-22. In some domains, with some targets, the setup method may be called more than once during initiation. You must keep this in mind if you use it to allocate or initialize memory.

13.5.1.22 begin

This item defines the begin method, which is called every time the simulation is started, but after the setup method is called, i.e., after any compile-time schedule is performed. The syntax is:

```plaintext
begin { body }
```

This method can be used to allocate and initialize memory. It is especially useful when data structures are shared across multiple instances of a primitive. It is always called exactly once when a simulation is started.

13.5.1.23 go

This item defines the action taken by the primitive when it is fired. The syntax is:

```plaintext
go { body }
```

For derived primitives the go method is empty. In such cases you can either:

- delete the go item. The go method of the base primitive is then called.
- explicitly call the go method of the base primitive.
- write new code for the go method.
A good example is *MLD Libraries/SDF Domain/DSP/LMS*. The *LMS* primitive is derived from *FIR*. The *go* method first updates the taps and then calls the *FIR’s* *go* method explicitly.

```c
    go {
        // First update the taps
        int index = int(errorDelay) * int(decimation) +
                    int(decimationPhase);
        double factor = double(error%0) * double(stepSize);
        for (int i = 0; i < taps.size(); i++)
            taps[i] += factor * double(signalIn%(index));
        index++;
    }

    // Then run FIR’s go method
    SDFFIR :: go();
```

The optional keyword *inline* may appear before the *go* keyword. The *go* method will typically read input particles and write outputs, and will be invoked many times during the course of a simulation. The code syntax for the body is explained in sec. 13.5.2 on page 13-22.

### 13.5.1.24 wrapup

This item defines the *wrapup* method, which is called at the completion of a simulation. The syntax is:

```c
    wrapup { body }
```

The optional keyword *inline* may appear before the *wrapup* keyword. The *wrapup* method might typically display or store final state values. The code syntax for doing this is explained in sec. 13.5.2 on 13-22. Note that the *wrapup* method is not invoked if an error occurs during execution. Thus, the *wrapup* method cannot be used reliably to free allocated memory. Instead, the dynamically allocated memory should be freed in the *cleanup* method or in the primitive’s destructor.

### 13.5.1.25 public, protected, private

These three keywords allow the programmer to declare extra members for the class with the desired protection. The syntax is:

```c
    protkey { body }
```

where *protkey* is *public*, *protected*, or *private*. Example, from the *XMgraph* primitive:

```c
    protected
    {
        XGraph graph;
        double index;
    }
```
This defines an instance of the class XGraph, defined in the MLDesigner kernel, and a double-precision number. If any of the added members require arguments for their constructors, use the conscalls item to specify them.

### 13.5.1.26 ccinclude, hinclude

These items cause the .cc file, or the .h file, to include extra files. A certain number of files are automatically included, when the preprocessor can determine that they are needed, so they do not need to be explicitly specified. If the file to be included is in the kernel it may be necessary to put in the absolute path in future versions of MLDesigner. The syntax is:

```
ccinclude { inclist }
hinclude { inclist }
cccinclude { "kernel/String.h" }
```

where inclist is a comma-separated list of include files. Each filename must be surrounded either by quotation marks or by < and > for system include files like <math.h>.

### 13.5.1.27 code

This keyword allows the programmer to specify a section of arbitrary C++ code. This code is inserted into the .cc file after the include files but before everything else. It can be used to define static non-class functions, declare external variables, or anything else. The outermost pair of curly braces is stripped. The syntax is:

```
code { body }
```

### 13.5.1.28 header

This keyword allows the programmer to specify an arbitrary set of definitions that will appear in the header file. Everything between the curly braces is inserted into the .h file after the include files but before everything else. This can be used, for example, to define classes used by your primitive. The outermost pair of curly braces is stripped.

### 13.5.1.29 method

The method item provides a fully general way to specify an additional method for the class of primitive that is being defined, for example:

```
virtual method
{
    name { exec }
    access { protected }
    arglist { "(const char* extraOpts)" }
    type { void }
    code { // code for the exec method goes here }
}
```
An optional function type specification may appear before the `method` keyword, which must be one of the following:

```plaintext
virtual
pure
inline
static
const
```

The `virtual` keyword makes a virtual member function. If the `pure virtual` keyword is given, a pure virtual member function is declared. There must be no code item in this case. The function type `pure` is a synonym for `pure virtual`. The `inline` function type declares the function to be inline.

If you use multiple type specifiers, the keywords have to be in the following relative order:

```plaintext
[const][inline][pure][virtual]
```

The `static` keyword declares a static method of the primitive class. It cannot be used in conjunction with `[const]` or `[pure] virtual as C++ does not support this. You can therefore use either `static method` or `inline static method` to declare a static member function.

The `method` subitems are:

- `name`: The name of the method. This is a required item.
- `access`: The level of access for the method, one of `public`, `protected` or `private`. If the item is omitted, `protected` is assumed.
- `arglist`: The argument list, including the outermost parentheses, for the method as a quoted string. If this is omitted, the method has no arguments.
- `type`: The return type of the method. If the return type is not a single identifier, you have to put quotes around it. If this is omitted, the return type is `void` (no value is returned).
- `code`: The code that implements the method. This is a required item, unless the `pure` keyword appears, in which case this item cannot appear.

### 13.5.1.30 `execTime`

This item defines the optional `myExecTime` function, which is used in code generation to specify how many time units are required to execute the primitive’s code. The syntax is:

```plaintext
execTime { body }
```

The optional keyword `inline` may appear before the `execTime` keyword. The `body` defines the body of a function that returns an integer value.

### 13.5.1.31 `codeblocks`

Codeblocks are parameterized blocks of code for use in code generation primitives. The syntax is:

```plaintext
codeblock { code }
```
13 Designing Primitives

13.5.2 Writing C++ Code for Primitives

Knowledge of C++ is required when reading this section. Furthermore, reading ch. 13.6 is highly recommended, since it explains some of the more generic and useful classes defined in the MLDesigner kernel. Many of these can be useful in primitives.

C++ code segments are an important part of any primitive definition. They can appear in the setup, begin, go, wrapup, constructor, destructor, execute, header, code, and method items of primitive source code. These items all include a body of arbitrary C++ code, enclosed by curly braces, { and }. In all but the code and header items, the C++ code between braces defines the body of a method of the primitive class. Methods can access any member of the class, including ports for input and output, parameters, and members defined with the public, protected, and private items.

The Structure of an MLDesigner Primitive

In general, the task of an MLDesigner primitive is to receive input particles and/or produce output particles. In addition, there may be side effects (reading or writing files, displaying graphs, or even updating shared data structures). As for all C++ objects, the constructor is called when the primitive is created, and the destructor is called when it is destroyed. In addition, the setup and begin methods -if any- are called every time a new simulation run is started, the go method (which always exists except for primitives like BlackHole and Null) is called each time a primitive is executed, and the wrapup and cleanup methods are called after the simulation run completes.

13.5.3 Reading Inputs and Writing Outputs

The exact mechanism for references to input and output ports depends somewhat on the domain. This is because primitives in the domain XXX use objects of class InXXXPort and OutXXXPort (derived from PortHole) for input and output, respectively. The examples we use here are for the SDF domain. See the appropriate domain chapter for variations that apply to other domains.

13.5.3.1 Portholes and Particles

In the SDF domain, normal inputs and outputs become members of type InSDFPort and OutSDFPort after the preprocessor is finished. These are derived from base class PortHole. For example, given the following item in the defprimitive of an SDF primitive,

```cpp
input
{
  name {in}
  type {float}
}
```

A member named in, of type InSDFPort, will become part of the primitive. We are not usually interested in directly accessing these porthole classes, but rather wish to read or write data through the portholes. All data passing through a porthole is derived from the base class Particle.
Each particle contains data of the type specified in the type subitem of the input or output item.

The operator % operating on a porthole returns a reference to a particle. Consider the following example:

```go
{  
  Particle& currentSample = in%0;  
  Particle& pastSample = in%1;  
  ...  
}
```

The right-hand argument to the % operator specifies the delay of the access. A zero always means the most recent particle. A one means the particle arriving just before the most recent particle. The same rules apply to outputs. Given an output named `out`, the same particles that are read from `in` can be written to `out` in the same order as follows:

```go
{  
  ...  
  out%1 = pastSample;  
  out%0 = currentSample;  
}
```

This works because `out%n` returns a reference to a particle, and hence can accept an assignment. The assignment operator for the class `Particle` is overloaded to make a copy of the data field of the particle.

Operating directly on class `Particle`, as in the above examples, is useful for writing primitives that accept any type of input. The operations need not concern themselves with the type of data contained by the particle. But it is far more common to operate numerically on the data carried by a particle. This can be done using a cast to a compatible type. In the example above, `in` is of type `float`, therefore its data can be accessed by

```go
{  
  Particle& currentSample = in%0;  
  double value = (double)currentSample;  
  ...  
}
```

or more concisely,

```go
{  
  double value = (double)(in%0);  
  ...  
}
```
The expression `(double)(in%0)` can be used anywhere a double can be used. In many contexts, where there is no ambiguity, the conversion operator can be omitted:

```plaintext
double value = in%0;
```

However, since conversion operators are defined to convert particles to several types, it is often necessary to indicate precisely which type of conversion is desired.

To write data to an output porthole, note that the right-hand side of the assignment operator should be of type `Particle`, as shown in the above example. An operator `<<` is defined for particle classes to make this more convenient. Consider the following example:

```plaintext
go{
  float t;
  t = some value to be sent to the output
  out%0 << t;
}
```

Note the distinction between the `<<` operator and the assignment operator. The latter operator copies particles, the former operator loads data into particles. The type of the right-side operand of `<<` may be `int`, `float`, `double`, `Fix`, `Complex` or `Envelope`. The appropriate type of conversion will be performed. For more information on the `Envelope` and `Message` types, please see sec. 14.

### 13.5.3.2 SDF PortHole Parameters

In the example above, where `in%1` was referenced, some special action is required to tell MLDesigner that past input particles are to be saved. A special action is also required to tell the SDF scheduler how many particles will be consumed at each input and produced at each output when a primitive fires. This information can be provided through a call to `setSDFParams` in the `setup` method. This has the syntax

```plaintext
setup{
  portName.setSDFParams(multiplicity, past)
}
```

where `portName` is the name of the input or output porthole, `multiplicity` is the number of particles consumed or produced, and `past` is the maximum value that offset can take in any expression of the form `name%offset`. For example, if the `go` method references `name%0` and `name%1`, then past would have to be at least one. It is zero by default.

### 13.5.3.3 Multiple Portholes

Sometimes a primitive should be defined with `n` input ports and/or `n` output ports, where `n` is a variable. This is supported by the class `MultiPortHole` and its derived classes (`MultiIn<domain>Port` and `MultiOut<domain>Port`). An object of this class has a sequential list of `PortHoles`. For SDF, we have the specialized derived class `MultiInSDFFPort` (which
contains InSDFPorts) and MultiOutSDFPort (which contains OutSDFPorts). Defining a multiple porthole is easy, as illustrated next:

\[
defprimitive
\{
... 
inmulti 
{ 
  name {input_name}
  type {input_type}
}
outmulti 
{ 
  name {output_name}
  type {output_type}
}
... 
\}
\]

To successfully access individual portholes in a MultiPortHole, In<domain>MPHIter or Out<domain>MPHIter iterator class should be used, function of the type of the port. Consider the following code segment from the definition of the SDFFork primitive:

```
input 
{ 
  name {input}
  type {anytype}
}
outmulti 
{ 
  name {output}
  type {=input}
}
... 
go 
{ 
  OutSDFMPHIter nextp(output);
  OutSDFPort* p;
  while ((p = nextp++) != 0)
    (*p)%0 = input%0;
}
```

A single input porthole supplies a particle that gets copied to any number of output portholes. The type of the output MultiPortHole is inherited from the type of the input. The first line of the go method creates an OutSDFMPHIter iterator called nextp, initialized to point to portholes in output. The ++ operator on the iterator returns a pointer to the next porthole in the list, until there are no more portholes, at which time it returns NULL. So the while construct steps through
all output portholes, copying the input particle data to the appropriate output. Consider another example, taken from the \textit{SDFAdd} primitive:

\begin{verbatim}
inmulti
{
  name {input}
  type {float}
}
output
{
  name {output}
  type {float}
}
go
{
  InSDFMPHIter nexti(input);
  InSDFPort* p;
  double sum = 0.0;
  while ((p = nexti++) != 0)
    sum += double((*p)%0);
  output%0 << sum;
}
\end{verbatim}

An \texttt{InSDFMPHIter} iterator named \texttt{nexti} is created and used to access the inputs individually. Occasionally the \texttt{numberPorts} method of class \texttt{MultiPortHole}, which returns the number of ports, is useful. This is called simply as \texttt{portname.numberPorts()}, and returns an integer.

### 13.5.3.4 Type Conversion

The type conversion operators and \texttt{<<} operators are defined as virtual methods in the base class \texttt{Particle}. There are never really objects of class \texttt{Particle} in the system. Instead, there are objects of class \texttt{IntParticle}, \texttt{FloatParticle}, \texttt{ComplexParticle}, and \texttt{FixParticle}, which hold data of type \texttt{int}, \texttt{double} (not \texttt{float!}), \texttt{Complex}, and \texttt{Fix}, respectively (there are also \texttt{MessageParticle}, \texttt{DataStructParticle} and a variety of matrix particles, described later). The conversion and loading operators are designed to “do the right thing” when an attempt is made to convert between mismatched types.

Clearly we can convert an \texttt{int} to a \texttt{double} or \texttt{Complex}, or a \texttt{double} to a \texttt{Complex}, with no loss of information. Attempts to convert in the opposite direction work as follows: conversion of a \texttt{Complex} to a \texttt{double} produces the magnitude of the complex number. Conversion of a \texttt{double} to an \texttt{int} produces the greatest integer that is less than or equal to the \texttt{double} value. There are also operators to convert to or from \texttt{float} and \texttt{Fix}. Each particle also has a virtual \texttt{print} method, so a primitive that writes particles to a file can accept any type.
13.5.4 Parameters

A parameter is defined by the `defparameter` item. The primitive can use a parameter to store data values, remembering them from one invocation to another. They differ from ordinary members of the primitive, which are defined using the `public`, `protected`, and `private` items, in the way that they have a name, and can be accessed from outside the primitive in systematic ways. For instance, the MLDesigner permits the programmer to set any parameter with the `A_SETTABLE` attribute to some value prior to a run. The MLDesigner command interpreter provides similar functionality through the `setparam` command. The parameter attributes are set in the `defparameter` item. A parameter may be modified by the primitive’s code during a run. The attribute `A_NONCONSTANT` is used as a pragma to mark a parameter as one that gets modified during a run. There is currently no mechanism for checking the properness of these attributes.

All parameters are derived from the base class `State`, defined in the MLDesigner kernel. The derived parameter classes currently defined in the kernel are `FloatState`, `IntState`, `ComplexState`, `FixState`, `FloatArrayState`, `IntArrayState`, `ComplexArrayState`, `StringArrayState`, `FixArrayState`, `DataStructState` and `EnumState`.

A parameter can be used in a primitive method just like the corresponding predefined data types. As an example, suppose the primitive definition contains the following directive:

```plaintext
defparameter
{
  name   { myState }  
  type   { float }    
  default { 1.0 }     
  descriptor { Gain parameter. }
}
```

This will define a member of class `FloatState` with default value 1.0. No attributes are defined, so `A_CONSTANT` and `A_SETTABLE`, the default attributes, are assumed. To use the value of a parameter, it should be cast to type `double`, either explicitly by the programmer or implicitly by the context. For example, the value of this parameter can be accessed in the `go` method as follows:

```plaintext
  go
  {
    output%0 << (double)myState * (double)(input%0);
  }
```

The references to `input` and `output` are explained above. The reference to `myState` has an explicit cast to `double`. This cast is defined in `FloatState` class. Similarly, a cast to `int` is available for `IntState`, to `Complex` from `ComplexState`, and to `const char*` for `StringState`. In principle, it is possible to rely on the compiler to automatically invoke this cast.

Attention:
Some compilers (notably some versions of g++) may not choose the expected cast. In particular,
g++ has been known to cast everything to Fix if the explicit cast is omitted in expressions similar to that above. The arithmetic is then performed using fixed-point point computations. This will be dramatically slower than double or integer arithmetic, and may yield unexpected results. It is best to explicitly cast parameters to the desired form. An exception is with simple assignment statements, like

```c
double stateValue = myName;
```

Even g++ gets this right. Explicit casting should be used whenever a parameter is used in an expression. For example, from the setup method of the SDFChop primitive, in which use_past_inputs is an integer parameter,

```c
if ( (int)use_past_inputs )
    input.setSDFParams((int)nread,(int)nread+(int)offset-1);
else
    input.setSDFParams((int)nread,(int)nread-1);
```

Note that the type Complex is not a fundamental part of C++. We have implemented a subset of the Complex class as defined by several library vendors. Using the ComplexState class will automatically ensure the inclusion of the appropriate header files. A member of type Complex can be initialized and operated upon any number of ways. For details, see sec. 14.1.1. A parameter may be updated by an ordinary assignment in C++, as in the following lines

```c
double t = expression;
myState = t;
```

This works because the assignment operator = has been overloaded by the FloatState class definition to set its value from a double. Similarly, an IntState can be set from an int and a StringState can be set from a char* or const char*.

### 13.5.5 Array Parameter

The ArrayState classes (FloatArrayState, IntArrayState, ComplexArrayState, FixArrayState and StringArrayState) are used to store arrays of data. For example,

```c
defparameter
{
    name   { taps }
    type   { FloatArray }
    default { "0.0 0.0 0.0 0.0" }
    descriptor { An array of length four. }
}
```

defines an array of type double with dimension four, with each element initialized to zero. Alternatively, you can specify a filename with a prefix <. If you have a file named foo that contains the default values for an array parameter, you can write,

```c
default { "< foo" }
```
If you expect others to be able to use your primitive, however, you should specify the default filename using a full path. For instance,

```plaintext
default { "< $MLD_USER/My_Library.lib/foo" }
```

For default files installed in the MLDesigner directory tree, this should read:

```plaintext
default { "< $MLD/directory/foo" }
```

The format of the file is also a sequence of data separated by spaces (or newlines, tabs, or commas). File input can be combined with direct data input as in

```plaintext
default { "< foo 2.0" }
default { "0.5 < foo < bar" }
```

A repeat notation is also supported for `ArrayState` objects. The two value strings

```plaintext
default { "1.0 [5]" }
default { "1.0 1.0 1.0 1.0 1.0" }
```

are equivalent. Any integer expression may appear inside the brackets `[]`. The number of elements in an `ArrayState` can be determined by calling its `size` method. The size is not specified explicitly, but is calculated by scanning the default value.

As an example of how to access the elements of an `ArrayState`, suppose `fState` is of type `FloatState` and `aState` is of type `FloatArrayState`. The accesses, like those in the following lines, are common:

```plaintext
fState = aState[1] + 0.5;
aState[1] = (double)fState * 10.0;
aState[0] = (double)fState * aState[2];
```

For a more complete example of the use of `FloatArrayState`, consider the `FIR` primitive defined below. Note that this is a simplified version of the `SDFFIR` primitive that does not permit interpolation or decimation.

```plaintext
defprimitive
{
    name  { FIR }
    domain { SDF }
    desc  { A Finite Impulse Response (FIR) filter. }
    input
    {
        name {signalIn}
        type {float}
    }
    output
    {
        name {signalOut}
        type {float}
    }
}
```

---

If you expect others to be able to use your primitive, however, you should specify the default filename using a full path. For instance,

```plaintext
default { "< $MLD_USER/My_Library.lib/foo" }
```

For default files installed in the MLDesigner directory tree, this should read:

```plaintext
default { "< $MLD/directory/foo" }
```

The format of the file is also a sequence of data separated by spaces (or newlines, tabs, or commas). File input can be combined with direct data input as in

```plaintext
default { "< foo 2.0" }
default { "0.5 < foo < bar" }
```

A repeat notation is also supported for `ArrayState` objects. The two value strings

```plaintext
default { "1.0 [5]" }
default { "1.0 1.0 1.0 1.0 1.0" }
```

are equivalent. Any integer expression may appear inside the brackets `[]`. The number of elements in an `ArrayState` can be determined by calling its `size` method. The size is not specified explicitly, but is calculated by scanning the default value.

As an example of how to access the elements of an `ArrayState`, suppose `fState` is of type `FloatState` and `aState` is of type `FloatArrayState`. The accesses, like those in the following lines, are common:

```plaintext
fState = aState[1] + 0.5;
aState[1] = (double)fState * 10.0;
aState[0] = (double)fState * aState[2];
```

For a more complete example of the use of `FloatArrayState`, consider the `FIR` primitive defined below. Note that this is a simplified version of the `SDFFIR` primitive that does not permit interpolation or decimation.

```plaintext
defprimitive
{
    name  { FIR }
    domain { SDF }
    desc  { A Finite Impulse Response (FIR) filter. }
    input
    {
        name {signalIn}
        type {float}
    }
    output
    {
        name {signalOut}
        type {float}
    }
```
defparameter
{
    name  {taps}
    type  {floatarray}
    default { "-0.04 -.001 .17 .37 .37 .17 -.0018 -.04" }
    desc  { Filter tap values. }
}

setup
{ // tell the PortHole the maximum delay we will use
    signalIn.setSDFParams(1, taps.size() - 1);
}
go
{
    double out = 0.0;
    for (int i = 0; i < taps.size(); i++)
    {
        out += taps[i] * (double)(signalIn%i);
        signalOut%0 << out;
    }
}

Notice the setup method; this is necessary to allocate a buffer in the signalIn input port large enough to hold the particles that are accessed in the go method. Notice the use of the size method of the FloatArrayState.

We now illustrate an PTCL interpreter session using the above FIR primitive. Assume there is a module called singen that generates a sine wave. You can use it with the FIR primitive, as in:

```
primitive foop singen
primitive fir FIR
primitive printer Printer
connect foop output fir signalIn
connect fir signalOut printer input
print fir
```

```
Primitive: mainGalaxy.fir
...
States in the primitive fir:
mainGalaxy.fir.taps type: FloatArray
initial value: 
.current value:
0 0.040609 1 -0.001628 2 .17853 3 .37665 4 .37665 5 .17853
```
13.5 Primitive Language Constructs

Then you can redefine taps by reading them from a file foo, which contains the data:

6 -0.001628
7 -0.040609

The resulting interpreter commands are:

```plaintext
setparam fir taps "<foo 5.5"
print fir
Primitive: mainGalaxy.fir
...
States in the primitive fir:
mainGalaxy.fir.taps type: FloatArray
initial value: <foo 5.5
current value:
0 1.1
1 -2.2
2 3.3
3 -4.4
4 5.5
```

This illustrates that both the contents and the size of a FloatArrayState are changed by a setparam command. Also, notice that file values may be combined with string values. When `< filename occurs in an initial value, it is processed exactly as if the whole file is substituted at that point.

### 13.5.6 Programming Examples

#### 13.5.6.1 Example 1

The following SDF primitive `Ramp` has no inputs, just an output port. The source primitive generates a linear increasing or decreasing sequence of float particles on its output. The parameter value is initialized to define the value of the first output. Each time the primitive’s go method fires, the value parameter is updated to store the next output value. Hence, the attributes of the value parameter are set, so that its value can be overwritten by the primitive’s methods. By default, the primitive will generate the output sequence 0.0, 1.0, 2.0, etc.

defprimitive
{
    name { Ramp }
    domain { SDF }
}
desc
{
    Generates a ramp signal, starting at "value" (default 0) and incrementing by step size "step" (default 1) on each firing.
}

output
{
    name { output }
    type { float }
}

defparameter
{
    name { step }
    type { float }
    default { "1.0" }
    desc { Increment from one sample to the next. }
}

defparameter
{
    name { value }
    type { float }
    default { "0.0" }
    desc { Initial (or latest) value output by Ramp. }
    attrib { A_NONCONSTANT|A_SETTABLE }
}

location { SDF main library }

go
{
    double t = value;
    output%0 << t;
    t += step;
    value = t;
}

13.5.6.2 Example 2

The next example is the SDF primitive *Gain*, which multiplies its input by a constant and outputs the result.

defprimitive
{
    name { Gain }
    domain { SDF }
    desc
    {
        The input multiplied by the parameter "gain" (default 1.0) equals the output.
    }

    htmldoc
    {
    }

    input
    {
        name { input }
        type { float }
    }

    output
    {
        name { output }
        type { float }
    }

    defparameter
    {
        name { gain }
        type { float }
        default { "1.0" }
        desc { Gain of the star. }
    }

    location { SDF main library }

    go
    {
        output%0 << (double)gain * (double)(input%0);
    }
13.5.6.3 Example 3

The following example of the SDF primitive Printer illustrates multiple inputs of type anytype, and the use of the print method of the Particle class. The wrapup method is also used here.

def primitive
{
    name       { Printer }
    domain     { SDF }
    desc
    {
        Print out one sample from each input port per line. The "fileName" parameter specifies the file to be written; the special names <stdout> and <cout>, which specify the standard output stream, and <stderr> and <cerr>, which specify the standard error stream, are also supported.
    }
}

html doc
{
}

in multi
{
    name   { input }
    type   { anytype }
}

def parameter
{
    name   { fileName }
    type   { filename }
    default { "<stdout>" }
    desc   { Filename for output. }
}

def parameter
{
    name   { Title }
    type   { string }
}

def parameter
{
13.5 Primitive Language Constructs

name  { EndCondition }
type  { boolean }
default  { "FALSE" }
desc  { If EndCondition is set to TRUE, the simulation will end when NumberOfItems have been consumed or when the number of cycles in Run Length have been executed, whichever comes first. }

defparameter
{
  name  { NumberOfItems }
type  { int }
default  { "1" }
desc  { The number of particles consumed by an input port. If the primitive receives NumberOfItems input particles and the parameter EndCondition is set to TRUE, then the simulation will be finished. }
}

location  { SDF main library }

hinclude  { "pt_fstream.h" }

code
{
  using std::cout;
  using std::endl;
}

protected
{
  pt_ofstream *p_out;
  int numItemsCounter;
  bool mReqEnd;
}

constructor
{
  p_out = 0;
}

setup
{
  if (EndCondition) {
    willRequestEnd();
  }
if (NumberOfItems < 1)
    Error::abortRun(*this, " the value of the parameter
    NumberOfItems is less than 1");
}
numItemsCounter = 0;
mReqEnd = false;

// in case file was open from previous run w/o wrapup call
LOG_DEL; delete p_out;
LOG_NEW; p_out = new pt_ofstream(fileName);
}
go
{
    if (!mReqEnd)
    {
        pt_ofstream& output = *p_out;
        InSDFMPHIter nexti(input);
        InSDFPort* p;
        output << endl; // Print 
        output << fullName() << endl;
        if (!Title.null())
            output << (const char*)Title << endl;
        while ((p = nexti++) != 0)
            output << (**p%0).print() << "\t";
        output << "\n";
        if (EndCondition && ++numItemsCounter == NumberOfItems)
        {
            mReqEnd = true;
            requestEnd();
        }
    }
}
wrapup
{
    LOG_DEL; delete p_out; // flush output
    p_out = 0;
}
destructor
{
    LOG_DEL; delete p_out; // flush output
}
This primitive is polymorphic since it can operate on any type of input. Note that the default value of the output filename is <stdout>, which causes the output to go to the standard output in this case the Command console. This and other aspects of the ptofstream output stream class are explained below in sec. 13.6.2.1 on page 13-40. The iterator nexti used to scan the input is explained in sec. 13.6.3 on page 13-48.

### 13.5.7 Preventing Memory Leaks in C++ Code

Memory leaks occur when new memory is allocated dynamically and never deallocated. In C programs, new memory is allocated by the malloc or calloc functions, and deallocated by the free function. In C++, new memory is usually allocated by the new operator and deallocated by the delete or the delete [] operator. The problem with memory leaks is that they accumulate over time and may cripple or even crash a program, if left unchecked. One of the most common mistakes leading to memory leaks is applying the wrong delete operator. The delete operator should be used to free a single allocated class or data value, whereas the delete [] operator should be used to free an array of data values. In C programming, the free function does not make this difference.

Another common mistake is overwriting a variable containing dynamic memory without freeing any existing memory first. For example, assume that thestring is a data member of a class, and in one of the methods (other than the constructor), there is the following statement:

```cpp
thestring = new char[buflen];
```

This code should be

```cpp
delete [] thestring;
thestring = new char[buflen];
```

Using delete is not necessary in a class’ constructor because the data member would not have been allocated previously.

In the MLDesigner primitives, the cleanup method should contain code that deletes variables dynamically allocated. In the primitive’s constructor method, the variables containing dynamic memory should be initialized to zero. By freeing memory using the cleanup method, one covers all possible cases of memory leak during simulation. Deallocating memory in the setup method handles the situation where a simulation is restarted, whereas deallocating memory in the cleanup covers the case in which a simulation is ended before or after the wrapup method. This includes cases where error messages are generated and the simulation cannot be continued.

For an example implementation, see the implementation of the SDF primitive Printer given in sec. 13.5.6.3.

Another common mistake is not paying attention to the kinds of strings returned by functions. The function savestring returns a new string dynamically allocated and should be deleted when no longer used. The expandPathName, tempFileName, and makeLower functions return new strings, like the Target::writeFileName method. Therefore, the strings returned by these routines should be deleted when they are no longer needed, and code such as

```cpp
savestring( expandPathName(s) )
```

is redundant and should be simplified to
13 Designing Primitives

expandPathName(s)
to avoid a memory leak due to not keeping track of the dynamic memory returned by the function
expandPathName.
Occasionally, dynamic memory is being used where local memory would be more convenient. For
example, if a variable is only used as a local variable inside a method or function and the value
of the local variable is not returned or passed to outside the method or function, then it would be
better to simply use local memory. For example,

    char* localstring = new char[len + 1];
    if ( person == absent ) return;
    strcpy(localstring, otherstring);
    delete [] localstring;
    return;

could easily return without deallocating localstring. The code should be rewritten to use
either the StringList or InfString class, e.g.,

    InfString localstring;
    if ( person == absent ) return;
    localstring = otherstring;
    return;

Both StringList and InfString can manage the construction of strings of arbitrary size. When a function or method finishes its execution, the destructors of the StringList and InfString variables will automatically be called and will deallocate their memory. Casts have
been defined that can convert StringList to a const char* string and InfString to
a const char* or a char* string, so that instances of the StringList and InfString
classes can be passed as they are into routines that take character array (string) arguments. A
simple example of using the StringList class is a function which builds up an error message
into a single string:

    StringList sl = msg;
    sl << file << ": " << sys_errlist[errno];
    ErrAdd(sl);

The ErrAdd function takes a const char* argument, so sl will converted automatically to a
const char* string by the C++ compiler.
Instead of using the new and delete operators, it is tempting to use constructs like

    char localstring[buflen + 1];

in which buflen is a variable, because the compiler will automatically handle the deallocation
of the memory. Unfortunately, this syntax is a GNU g++ extension and not portable to other C++
compilers. Instead, the StringList and InfString classes should be used, as the previous
example involving localstring illustrates.
Sometimes the return value from a routine that returns dynamic memory is not stored, and there-
fore, the pointer to the dynamic memory gets lost. This occurs, for example, in nested function
calls. Code like
puts( savestring(s) );

should be written as

const char* newstring = savestring(s);
puts( newstring );
delete [] newstring;

Several places in MLDesigner, especially in the schedulers and targets, rely on the hashstring function, which returns dynamic memory. This dynamic memory, however, should not be deallocated because it may be reused by other calls to hashstring. It is the responsibility of the hashstring function to deallocate any memory it has allocated.

## 13.6 Infrastructure for Primitive Definition

The MLDesigner kernel provides a number of C++ classes that often prove useful to primitive writers. Some of these are essential such as those that handle errors. Complete documentation of the kernel classes is given in [BH97]. Here, we summarize only the most generic of these classes, i.e., the ones that are generally useful to primitive programmers. All of these classes described here may be used in primitives, provided that the primitive writer includes the appropriate header files. For instance, the entry

```
cinlude { "pt_fstream.h" }
```

will permit the primitive to create instances of the basic stream classes (described below) in the body of functions that are defined in the primitive.

If the programmer wishes to create an instance as a private, protected, or public member of the primitive, then the header file needs to be included in the .h file, specified in the line

```
hinlude { "pt_fstream.h" }
```

in the SDF primitive Printer defined on page 13-34.

### 13.6.1 Handling Errors

Uniform handling of errors is provided by the Error class. The Error class provides four static methods, summarized in table 13.5. From within a primitive definition, it is not necessary to explicitly include the Error.h header file. A typical use of the class is shown below:

```
Error::abortRun(*this,"this message is displayed");
```

The notation Error::abortRun is the way static methods are invoked in C++ without having a pointer to an instance of the Error class. The first argument tells the Error class which object is flagging the error; this is strongly recommended. The name of the object will be printed along with the error message. Note that the abortRun call does not cause an immediate halt. It simply marks a flag that the scheduler must test for. After an error insert return;

Table 13.5 uses standard C++ notation to indicate how to use the methods. The type of the return value and the type of the arguments is given, together with an explanation of each.
## 13 Designing Primitives

### 13.6.2 I/O Classes

The programmer who is working with primitives often need to communicate with the user. The most flexible way to do this is to build a customized, window-based interface, as described in sec. 16 on page 16-1. However, often it is sufficient to plot some data or to just construct strings and output them to files or to the standard output\(^2\). To do the latter, use the classes `pt_ifstream` and `pt ofstream`, which are derived from the standard C++ stream classes `ifstream` and `ofstream`, respectively. More sophisticated output can be obtained with the `XGraph` class, the `histogram` classes, and classes that interface to Tk for generating animated, interactive displays. All of these classes are summarized in this section.

### 13.6.2.1 Extended Input and Output Stream Classes

The `pt ofstream` class is used in the SDF primitive `Printer` on page 13-34. Include the header file `pt fstream.h`. The `pt ofstream` constructor is invoked in the `setup` method with the call to `new`. It would not work to invoke it in the constructor for the primitive, since the `fileName` parameter would not have been initialized. Notice that the `setup` method reclaims

---

\(^2\)Note that when you run MLDesigner, the standard output is shown in MLDesigner output window. You can start MLDesigner with the `-c` option to get the output within the console window.
the memory allocated in previous runs (or previous invocations of the setup method) before creating a new pt_ofstream object. The classes pt_ifstream and pt_ofstream are only a slight extension of the C++ standard classes ifstream and ofstream. They add the following features:

- First, certain special filenames are recognized as arguments to the constructor or to the open method. These filenames are <cin>, <cout>, <cerr>, or <clog> (the angle brackets must be part of the string), then the corresponding standard stream of the same name is used for input (pt_ifstream) or output (pt_ofstream). Users accustomed to C standard I/O can alternatively use <stdin>, <stdout>, or <stderr>.
- Second, the MLDesigner expandPathName, see table 13.10 on page 13-46, is applied to the filename before it is opened, permitting it to start with ~user or $VAR.
- Finally, if a failure occurs when the file is opened, Error::abortRun is called with an appropriate error message, including the Unix error condition.

These classes can be used for binary character data as well as ASCII.

### 13.6.2.2 Generating Graphs Using the XGraph Class

The XGraph class provides an interface to the 2D plotting functions of MLDesigner. The 2D plotting system and all plotting options are documented in sec. 8.2. An example of the output from MLDesigner 2D plotting system is shown in fig. 13.4.

![Figure 13.4: Example of output of the plotting system using the XGraph class](image)

The most useful methods of the class are summarized in table 13.6. Using the XGraph class involves an invocation of the initialize method, some number of invocations of the addPoint method, followed by an invocation of the terminate method. Multiple datasets (currently up to 64) may be plotted together. They will each be given a distinctive color and/or line pattern. Within each dataset, it is possible to break the connecting lines between points by calling the newTrace method.
13 Designing Primitives

### Class XGraph

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void initialize (...)</code></td>
<td><code>Block* parent</code></td>
<td>start a new plot</td>
</tr>
<tr>
<td></td>
<td><code>int noGraphs</code></td>
<td>pointer to the block using the class</td>
</tr>
<tr>
<td></td>
<td><code>const char* options</code></td>
<td>the number of datasets to plot</td>
</tr>
<tr>
<td></td>
<td><code>const char* title</code></td>
<td>options to pass to the plotting system</td>
</tr>
<tr>
<td></td>
<td><code>const char* saveFile = 0</code></td>
<td>title to put on the graph</td>
</tr>
<tr>
<td></td>
<td><code>int ignore = 0</code></td>
<td>name of a file to save data to</td>
</tr>
<tr>
<td></td>
<td><code>int ignore = 0</code></td>
<td>number of initial points to ignore</td>
</tr>
<tr>
<td><code>void addPoint (...)</code></td>
<td><code>float y</code></td>
<td>add the next point to the first dataset with implicit x position</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the vertical position</td>
</tr>
<tr>
<td><code>void addPoint (...)</code></td>
<td><code>float x</code></td>
<td>add a single point to the first dataset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the horizontal position of the point to plot</td>
</tr>
<tr>
<td></td>
<td><code>float y</code></td>
<td>the vertical position of the point to plot</td>
</tr>
<tr>
<td><code>void addPoint (...)</code></td>
<td><code>int dataSet</code></td>
<td>add a single point to a particular dataset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the number of the dataset (starting with 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the horizontal position of the point to plot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the vertical position of the point to plot</td>
</tr>
<tr>
<td><code>void newTrace (...)</code></td>
<td><code>int dataSet = 1</code></td>
<td>start a new trace independent from the previous trace</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the dataset for the new trace</td>
</tr>
<tr>
<td><code>void terminate ()</code></td>
<td></td>
<td>plot the data using the 2D plotting system</td>
</tr>
</tbody>
</table>

Table 13.6: Summary of methods of class XGraph

13.6.2.3 Classes for Displaying Animated Bar Graphs

The BarGraph class creates a Tk window to displays a bar graph that can be modified dynamically, while a simulation runs. An example with 5 datasets and 8 bars per dataset is shown in fig. 13.5. The most useful methods of the class are summarized in table 13.7. Correspondingly, the class definition source code is in `$MLD/src/pigilib`, rather than the more usual `$MLD/src/kernel`. 
13.6 Infrastructure for Primitive Definition

Figure 13.5: Example of animated bar graph using the BarGraph class

<table>
<thead>
<tr>
<th>Class</th>
<th>BarGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Parameter</td>
</tr>
<tr>
<td>int setup (...)</td>
<td>Block* parent</td>
</tr>
<tr>
<td></td>
<td>char* desc</td>
</tr>
<tr>
<td></td>
<td>int numInputs</td>
</tr>
<tr>
<td></td>
<td>int numBars</td>
</tr>
<tr>
<td></td>
<td>double top</td>
</tr>
<tr>
<td></td>
<td>double bottom</td>
</tr>
<tr>
<td></td>
<td>char* geometry</td>
</tr>
<tr>
<td></td>
<td>double width</td>
</tr>
<tr>
<td></td>
<td>double height</td>
</tr>
<tr>
<td>int update (...)</td>
<td>int dataSet</td>
</tr>
<tr>
<td></td>
<td>int bar</td>
</tr>
<tr>
<td></td>
<td>double y</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 13.7: Summary of methods of class BarGraph
13.6.2.4 Collecting Statistics Using the Histogram Classes

The Histogram class constructs a histogram of data supplied using Tk widget classes. The XHistogram class also constructs a histogram, but then plots it using the 2D plotting system. The most useful methods of both classes are summarized in table 13.8 and table 13.9. The Histogram class counts the number of occurrences of data values that fall within each of a number of bins. Each bin represents a range of numbers. All bins have the same width, and the center of each bin will be an integer multiple of this width. Bin number 0 is always the one with the smallest center. Bins are added if new data arrives that does not fit within any of the existing bins. The getData method is used to read out the contents of a bin. If you start with bin number 0, and proceed until getData returns FALSE, you will have read all the bins.

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>Histogram (...)</td>
<td></td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td>double width = 1.0</td>
<td></td>
<td>the width of each bin; bins are centered at integer multiples of this value</td>
</tr>
<tr>
<td></td>
<td>int maxBins = 1000</td>
<td></td>
<td>since bins are added as needed, it is wise to limit their number</td>
</tr>
<tr>
<td></td>
<td>void add (...)</td>
<td>double x</td>
<td>add to the count of the bin within the given data belongs in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a data point for the histogram</td>
</tr>
<tr>
<td></td>
<td>int numCounts ()</td>
<td></td>
<td>return the number of data values used so far in the histogram</td>
</tr>
<tr>
<td></td>
<td>double mean ()</td>
<td></td>
<td>return the average value of all observed data so far</td>
</tr>
<tr>
<td></td>
<td>double variance ()</td>
<td></td>
<td>return the variance of the observed data so far</td>
</tr>
<tr>
<td></td>
<td>int getData (...)</td>
<td>int binno</td>
<td>get the count for a given bin; return FALSE if the bin is out of range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int&amp; count</td>
<td>starting at 0, the bin number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double&amp; binCenter</td>
<td>place to store the count for the given bin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>place to store the center of the given bin</td>
</tr>
</tbody>
</table>

Table 13.8: Summary of methods of class Histogram
13.6 Infrastructure for Primitive Definition

<table>
<thead>
<tr>
<th>Class XHistogram</th>
<th>#include &quot;Histogram.h&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>void initialize (...)</td>
<td>start a new histogram</td>
</tr>
<tr>
<td>Block* parent</td>
<td>pointer to the block using the class</td>
</tr>
<tr>
<td>double binWidth</td>
<td>the width of each bin; bins are centered at integer multiples of this value</td>
</tr>
<tr>
<td>const char* options</td>
<td>options to pass to the 2D plotting system, in addition to -bar -nl -brw</td>
</tr>
<tr>
<td>const char* title</td>
<td>title to put on the histogram</td>
</tr>
<tr>
<td>const char* saveFile</td>
<td>name of a file to save data to (or 0 if none)</td>
</tr>
<tr>
<td>int maxBins = 1000</td>
<td>limit the number of bins</td>
</tr>
<tr>
<td>void addPoint (...)</td>
<td>add to the count of the bin within the given data belongs in</td>
</tr>
<tr>
<td>double y</td>
<td>a data point for the histogram</td>
</tr>
<tr>
<td>int numCounts ()</td>
<td>return the number of data values used so far in the histogram</td>
</tr>
<tr>
<td>double mean ()</td>
<td>return the average value of all observed data so far</td>
</tr>
<tr>
<td>double variance ()</td>
<td>return the variance of the observed data so far</td>
</tr>
<tr>
<td>void terminate ()</td>
<td>plot the histogram using the 2D plotting system</td>
</tr>
</tbody>
</table>

Table 13.9: Summary of methods of class XHistogram

13.6.3 String Functions and Classes

The MLDesigner kernel defines some ordinary functions (not classes) plus two classes that are useful for building and manipulating C style strings. The non-class string functions are summarized in table 13.10. This includes functions for copying strings, adding strings to a system-wide hash table and creating temporary filenames. The non-class pathname functions are summarized in table 13.11. These functions are for expanding filenames that might begin with a reference to a user’s home directory ~user or contain a shell environment variable $VAR. Also provided is a function for verifying that an external program to be invoked is available, and a function for searching the user’s path.
### String functions

<table>
<thead>
<tr>
<th>Function Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char* savestring (...)</td>
<td>create a new copy of the given text and return a pointer to it; the caller must eventually delete the string.</td>
</tr>
<tr>
<td>const char* text</td>
<td></td>
</tr>
</tbody>
</table>

| const char* hashstring (...) | save a copy of the text in a system-wide hash table, if it isn’t already there, and return a pointer to the entry. |
| const char* text |

| char* tempFileName () | return a new, unique temporary filename; the caller must eventually delete the string. |
| const char* text |

| const char* expandPathName (...) | return an expanded version of the filename argument, which may start with ~, ~user, or $VAR; the expanded result is in static storage, and will be overwritten by the next call. |
| const char* text |

Table 13.10: Summary of String functions

### Path search functions

<table>
<thead>
<tr>
<th>Function Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int progNotFound (...)</td>
<td>flag an error and return TRUE if a program is not found</td>
</tr>
<tr>
<td>const char* program</td>
<td></td>
</tr>
<tr>
<td>const char* extramsg= 0</td>
<td></td>
</tr>
</tbody>
</table>

| const char* pathSearch (...) | find a file in a Unix-style path, returning the directory name |
| const char* file |
| const char* path = 0 |

Table 13.11: Summary of Path search functions

Two classes are provided for manipulating strings, `InfString`, and `StringList`, these
classes are summarized in table 13.12 and table 13.13. The `InfString` class inherits all of the methods from `StringList`, adding only the cast to `char*`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StringList (...)</code></td>
<td></td>
<td>constructors can take one of the following possible arguments</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>return an empty <code>StringList</code></td>
</tr>
<tr>
<td></td>
<td><code>const StringList&amp; s</code></td>
<td>copy 's and return a new, identical <code>StringList</code></td>
</tr>
<tr>
<td></td>
<td><code>char c</code></td>
<td>return a <code>StringList</code> with one string of one character</td>
</tr>
<tr>
<td></td>
<td><code>const char* string</code></td>
<td>copy the string and makes a one element <code>StringList</code> containing it</td>
</tr>
<tr>
<td></td>
<td><code>int i</code></td>
<td>create an ASCII representation of the number and return a one element <code>StringList</code></td>
</tr>
<tr>
<td></td>
<td><code>double x</code></td>
<td>same as above</td>
</tr>
<tr>
<td></td>
<td><code>unsigned int u</code></td>
<td>same as above</td>
</tr>
<tr>
<td><code>StringList&amp; operator =</code></td>
<td></td>
<td>assignment takes the same types of arguments as the constructors, except <code>none</code></td>
</tr>
<tr>
<td><code>StringList&amp; operator &lt;&lt;</code></td>
<td></td>
<td>add one or more elements to a <code>StringList</code>; this takes the same types of arguments as the constructors, except <code>none</code></td>
</tr>
<tr>
<td><code>operator const char*</code></td>
<td></td>
<td>join all elements together and return as a single string</td>
</tr>
<tr>
<td><code>void initialize ()</code></td>
<td></td>
<td>delete all elements, making the <code>StringList</code> empty</td>
</tr>
<tr>
<td><code>int length ()</code></td>
<td></td>
<td>return the length in characters (sum of the lengths of the elements)</td>
</tr>
<tr>
<td><code>int numPieces ()</code></td>
<td></td>
<td>return the number of elements</td>
</tr>
<tr>
<td><code>const char* head ()</code></td>
<td></td>
<td>return the first element</td>
</tr>
<tr>
<td><code>char* newCopy ()</code></td>
<td></td>
<td>return the concatenated elements in a single newly allocated string; the caller must free the memory allocated</td>
</tr>
</tbody>
</table>

Table 13.12: Summary of methods of class `StringList`
13 Designing Primitives

<table>
<thead>
<tr>
<th>Class InfString</th>
<th>#include &quot;InfString.h&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>all StringList methods</td>
<td>see above</td>
</tr>
<tr>
<td>operator char*</td>
<td>join all elements together and return as a single string</td>
</tr>
</tbody>
</table>

Table 13.13: Summary of methods of class InfString

Although these two classes are almost identical in design, their recommended uses are quite different. The first is designed for building up strings without having to be concerned about the current or maximum size of the string. New characters can be appended to the string at any time, and memory will be allocated to accommodate them. When you are ready to use the string, perhaps by passing it to a function that expects the standard character array representation of the string, simply cast the object to char*.

In fact, InfString is publicly derived from StringList, adding only the cast to char*. StringList is implemented as a list of strings, where the size of the list is not bounded ahead of time. StringList is recommended for applications where the list structure is to be preserved. The cast to char* in InfString destroys the list structure, consolidating all its strings into one contiguous string.

A word of warning is necessary: if a function or expression returns a StringList or InfString, and that value is not assigned to a StringList or InfString variable or reference, and the const char* or char* cast is used, it is possible (like under g++) that the StringList or InfString temporary will be destroyed too soon, leaving the const char* or char* pointer pointing to garbage. The solution is to assign the returned value to a local StringList or InfString before performing the cast. Suppose, for example, that the function foo returns an InfString. Further, suppose the function bar takes a char* argument. Then the following code will fail:

    bar(foo());

(Note that the cast to char* is implicit). The following code will succeed:

    InfString x = foo();
    bar(x);

The StringList class is one of several list classes in the MLDesigner kernel. A basic operation on list classes is to sequentially access their members one at a time. This is accomplished using an iterator class, in comparison to the list class. For the StringList class, the iterator is called StringListIter. Its methods are summarized in table 13.14.
An example program fragment using this is given below:

```c++
StringListIter item(myList);
const char* string;
while ((string = item++) != 0) cout << string << "\n";
```

In this code sequence, `myList` is assumed to be a `StringList` previously set up.

### 13.6.4 List Classes

The `StringList` class is derived from the `SequentialList` class. This class is widely used within MLDesigner to provide list functionality. This class implements a linked list with a running count of the number of elements. It uses the generic pointer technique with

```c++
typedef void* Pointer
```

Thus, items in a sequential list can be pointers to any object, with a generic pointer used to access the object. In derived classes, like `StringList`, this generic pointer is converted to a specific type of pointer, like `const char*`. The methods are summarized in table 13.15.

An important point to keep in mind when using a `SequentialList` is that its destructor does not delete the elements in the list. It would not be possible to do so, since it has only a generic pointer. Also, note that random access (by element number, or any other method) can be very inefficient, since it would require sequentially chaining down the list.

`SequentialList` has an iterator class called `ListIter`. The `++` operator (or `next` member function) returns a `Pointer`. In Table 13.16 and Table 13.17 are shown two classes that are privately derived from `SequentialList`, `Queue` and `Stack`. The first of these can implement either a first-in / first-out (FIFO) queue, or a last-in / first-out (LIFO) queue. The second implements a stack, which is equivalent to a LIFO queue.
### Class SequentialList

```cpp
#include "DataStruct.h"
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void append (Pointer p)</td>
<td></td>
<td>add the element p to the end of the list</td>
</tr>
<tr>
<td>Pointer elem (int n)</td>
<td></td>
<td>return the n-th element in the list (zero if there are fewer than n)</td>
</tr>
<tr>
<td>int empty ()</td>
<td></td>
<td>return 1 if empty, 0 if not</td>
</tr>
<tr>
<td>Pointer getAndRemove ()</td>
<td></td>
<td>return and remove the first element in the list (return zero if empty)</td>
</tr>
<tr>
<td>Pointer getTailAndRemove ()</td>
<td></td>
<td>return and remove the last element in the list (return zero if empty)</td>
</tr>
<tr>
<td>Pointer head ()</td>
<td></td>
<td>return the first element in the list (zero if empty)</td>
</tr>
<tr>
<td>void initialize ()</td>
<td></td>
<td>remove all elements from the list</td>
</tr>
<tr>
<td>int member (Pointer p)</td>
<td></td>
<td>return 1 if the list has a pointer equal to p, 0 if not</td>
</tr>
<tr>
<td>void prepend (Pointer p)</td>
<td></td>
<td>add the element p to the beginning of the list</td>
</tr>
<tr>
<td>int remove (Pointer p)</td>
<td></td>
<td>if the list has a pointer equal to p, remove it, and return 1; 0 if not</td>
</tr>
<tr>
<td>int size ()</td>
<td></td>
<td>return the number of elements in the list</td>
</tr>
<tr>
<td>Pointer tail ()</td>
<td></td>
<td>return the last element in the list (zero if empty)</td>
</tr>
</tbody>
</table>

Table 13.15: Summary of methods of class SequentialList

### Class Queue

```cpp
#include "DataStruct.h"
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer getHead ()</td>
<td></td>
<td>return and remove the first element in the queue (return zero if empty)</td>
</tr>
<tr>
<td>Pointer getTail ()</td>
<td></td>
<td>return and remove the last element in the queue (return zero if empty)</td>
</tr>
<tr>
<td>void initialize ()</td>
<td></td>
<td>remove all elements from the queue</td>
</tr>
<tr>
<td>void putHead (Pointer p)</td>
<td></td>
<td>add the element p to the beginning of the queue</td>
</tr>
<tr>
<td>void putTail (Pointer p)</td>
<td></td>
<td>add the element p to the end of the queue</td>
</tr>
<tr>
<td>int size ()</td>
<td></td>
<td>return the number of elements in the queue</td>
</tr>
</tbody>
</table>
### 13.6 Infrastructure for Primitive Definition

#### Class Queue

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><code>#include &quot;DataStruct.h&quot;</code></td>
</tr>
</tbody>
</table>

Table 13.16: Summary of methods of class Queue

#### Class Stack

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer accessTop ()</td>
<td></td>
</tr>
<tr>
<td>void initialize ()</td>
<td></td>
</tr>
<tr>
<td>Pointer popTop ()</td>
<td></td>
</tr>
<tr>
<td>void pushBottom (Pointer p)</td>
<td></td>
</tr>
<tr>
<td>void pushTop (Pointer p)</td>
<td></td>
</tr>
<tr>
<td>int size ()</td>
<td></td>
</tr>
</tbody>
</table>

#include "DataStruct.h"

Table 13.17: Summary of methods of class Stack

### 13.6.5 Hash Tables

Hash tables are lists that are indexed by an ASCII string. A hash function computes the key index from the string to make random accesses reasonably efficient. They are much more efficient, for example, than a linear search over a `SequentialList`. Two of these classes are provided in the MLDesigner kernel. The first, `HashTable`, is generic, in that the table entries are of type `Pointer`, and thus can point to any user-defined data structure. The second, `TextTable`, is more specialized; the entries are strings. It is derived from `HashTable`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include &quot;HashTable.h&quot;</td>
<td></td>
</tr>
</tbody>
</table>

#### Class HashTable

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear ()</td>
<td></td>
</tr>
<tr>
<td>virtual void cleanup (...)</td>
<td>Pointer p</td>
</tr>
<tr>
<td>int hasKey (...)</td>
<td>const char* key</td>
</tr>
</tbody>
</table>

13-51
### Class HashTable

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void insert (...)</td>
<td>const char* key, Pointer data</td>
<td>insert an entry; any previous entry with the same key is replaced, and the cleanup method is called so that in derived classes, its memory can be deallocated</td>
</tr>
<tr>
<td>Pointer lookup (...)</td>
<td>const char* key</td>
<td>look up an entry; in a derived class, this could be overloaded to return a pointer of a more specific type</td>
</tr>
<tr>
<td>int remove (...)</td>
<td>const char* key</td>
<td>remove the entry with the given key from the table; note that the object pointed to by the entry is not deallocated</td>
</tr>
<tr>
<td>int size ()</td>
<td></td>
<td>return the number of entries in the hash table</td>
</tr>
</tbody>
</table>

Table 13.18: Summary of methods of class HashTable

The HashTable class is summarized in table 13.18 and TextTable class is summarized in table 13.19. Only the most useful (and easily used) methods are described. You may want to refer to the source code for more information. The HashTable class has a standard iterator called HashTableIter, where the next method and ++ operator return a pointer to class HashEntry. This class has a const char* key() method that returns the key for the entry, and a Pointer value() method that returns a pointer to the entry. TextTable has an iterator called TextTableIter, where the next method and ++ operator return type const char*.

Sophisticated users will often want to derive new classes from HashTable. The reason is that the methods that look up data in the table can be defined to return pointers of the appropriate type. Moreover, the deallocation of memory when an entry is deleted or the table itself is deleted can be automated. TextTable is a good example of such a derived class. This is not possible with the generic HashTable class, because the Pointer type does not give enough information to know what destructor to invoke. Thus, when using the generic HashTable class, the programmer should explicitly delete the objects pointed to by the Pointer if they were dynamically created and are no longer needed. A detailed example that directly uses the HashTable class, without defining a derived class, is given in the next section. In that example, the Pointer entries point to primitive in a system, so they should not be deleted when the entries in the table are deleted. Their memory will be deallocated when the system is deleted.
13.6 Infrastructure for Primitive Definition

| Class | TextTable | #include "HashTable.h"
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>void clear ()</td>
<td></td>
<td>empty the table</td>
</tr>
<tr>
<td>void cleanup (...)</td>
<td>Pointer p</td>
<td>deallocate the string pointed to by p</td>
</tr>
<tr>
<td>int hasKey (...)</td>
<td>const char* key</td>
<td>return 1 if the given key is in the table, 0 otherwise</td>
</tr>
<tr>
<td>void insert (...)</td>
<td>const char* key, const char* string</td>
<td>create an entry containing a copy of string; a previous entry with the same key is replaced, and the cleanup method is called to deallocate its memory</td>
</tr>
<tr>
<td>const char* lookup (...)</td>
<td>const char* key</td>
<td>look up an entry with the given key; return 0 if there is no such entry</td>
</tr>
<tr>
<td>int remove (...)</td>
<td>const char* key</td>
<td>remove the entry with the given key from the table and deallocated its memory</td>
</tr>
<tr>
<td>int size ()</td>
<td>const char* key</td>
<td>return the number of entries in the hash table</td>
</tr>
</tbody>
</table>

Table 13.19: Summary of methods of class TextTable

13.6.6 Using Random Numbers

The primitives Random* found in the Number Generators library replaces the wide range of primitives that were previously available and have been moved to Compatibility/NumberGenerators. Every time the primitive is executed it generates a value that is a random deviate drawn from the distribution selected by parameter Distribution. The Properties Editor Window of a random number generator instance primitive is shown in fig. 13.6.

The following distributions are available, with their parameters shown in parenthesis:

1. Binomial - (Trials, Probability, Seed)
2. Exponential - (Mean, Seed)
3. Normal - Mean, Variance, Seed)
4. Poisson - (Mean, Seed)
5. Uniform - (Min, Max, Seed)

A separate seed value can be defined for each instance of the primitive which determines whether
13 Designing Primitives

Figure 13.6: Random Number Generator Properties

1. a local seed is used,
2. a generated seed is used, or
3. the global seed is used.

The parameter Seed is used to determine how the given seed value is used by the primitive (see table 13.20).

<table>
<thead>
<tr>
<th>Seed Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>The given seed value is used as local seed, that is, all instances with the same seed generate the same sequence of deviates on each simulation run.</td>
</tr>
<tr>
<td>Negative</td>
<td>The global seed is used. All instances generate different sequences during a simulation run, but each instance generates the same sequence of deviates on consecutive simulation runs.</td>
</tr>
<tr>
<td>Zero</td>
<td>A unique seed value is computed using current system time. All instances generate different sequences during the simulation run and each instance generates a different sequence of deviates on consecutive simulation runs.</td>
</tr>
</tbody>
</table>

Table 13.20: Random Number Generation and Seed Value
The following code must be used (duplicated code indicates that it can be used in either the setup or the go method):

```cpp
#include { <Rng/Normal.h> }

protected
{
    Rng::Normal mRandom;
}

setup
{
    mRandom.setMean(MeanValue);
    mRandom.setVariance(VarianceValue);
    mRandom.setSeed(SeedValue);
}

go
{
    mRandom.setMean(MeanValue);
    mRandom.setVariance(VarianceValue);
    mRandom.setSeed(SeedValue);

    Output%0 << mRandom();
}
```
Chapter 14

Using Data Types

Primitives communicate with each other by sending objects of Particle type. A basic set of types, including scalar and array types, inherited from the Particle class, is built into the MLDesigner kernel. Since all of these particle types are derived from the same base class, it is possible to write primitives that operate on any of them (by referring only to the base class). It is also possible to define new types that contain arbitrary C++ objects.

There are currently eleven key data types defined in the MLDesigner kernel. There are four numeric scalar types - complex, fixed-point, double precision floating-point, and integer - described in sec. 14.1. MLDesigner supports a limited form of user-defined type, the Message type, described in sec. 14.2. Each of the scalar numeric types has an equivalent matrix type, which uses a more complex version of the user-defined type mechanism; they are described in sec. 14.3.

There are two experimental types included in the basic set, containing strings and file references, described in sec. 14.4. MLDesigner allows primitives to be written that will read and write particles of any type. This mechanism is described in sec. 14.5. Finally, some additional experimental types described in sec. 14.6.

NOTE: Experimental classes are currently unsupported and are likely to be changed in the future.

14.1 Scalar Numeric Types

There are four scalar numeric data types defined in the MLDesigner kernel:

- complex
- fixed-point
- floating-point with double precision
- integer

All of these four types can be read from and written to ports as described in sec. 13.5.3 on page 13-22. The floating-point and integer data types are based on the standard C++ double and int types, and need no further explanation. To support the other two types, the MLDesigner kernel contains a Complex and a Fix class, which are described in this section.
14 Using Data Types

14.1 Complex Data Type

The Complex data type in MLDesigner contains real and imaginary components, each of them is specified as a double precision floating-point number. The notation used to represent a complex number is a two number pair (real, imaginary) - for example, \(1.3 - 4.5\) corresponds to the complex number \(1.3 - 4.5j\). Complex implements a subset of the functionality of the complex number classes in the cfront and libg++ libraries, including most of the standard arithmetic operators and a few transcendental functions.

14.1.1 Constructors

Complex()

Create a complex number, initialized with zero \(0.0, 0.0\). For example:

```cpp
Complex c;
```

Complex(double real, double imag)

Create a complex number with the value \((real, imag)\). For example:

```cpp
Complex c(1.3, -4.5);
```

Complex(const Complex& arg)

Create a complex number which is identical to the complex number \(arg\). For example:

```cpp
Complex c(complexSourceNumber);
```

14.1.2 Basic Operators

The following list of arithmetic operators modify the value of the complex number. All functions return a reference to the modified complex number (*this).

```cpp
Complex& operator = (const Complex& arg)
Complex& operator += (const Complex& arg)
Complex& operator -= (const Complex& arg)
Complex& operator *= (const Complex& arg)
Complex& operator /= (const Complex& arg)
Complex& operator *= (double arg)
Complex& operator /= (double arg)
```

These two operators return the real and imaginary parts of the complex number:

```cpp
double real() const
double imag() const
```
### 14.1.1.3 Non-member functions and operators

The following unary and binary operators return a new complex number:

- `Complex operator + (const Complex& x, const Complex& y)`
- `Complex operator - (const Complex& x, const Complex& y)`
- `Complex operator * (const Complex& x, const Complex& y)`
- `Complex operator * (double x, const Complex& y)`
- `Complex operator * (const Complex& x, double y)`
- `Complex operator / (const Complex& x, const Complex& y)`
- `Complex operator / (const Complex& x, double y)`
- `Complex operator - (const Complex& x)`
- `Complex conj (const Complex& x)`
- `Complex sin(const Complex& x)`
- `Complex cos(const Complex& x)`
- `Complex exp(const Complex& x)`
- `Complex log(const Complex& x)`
- `Complex sqrt(const Complex& x)`
- `Complex pow(double base, const Complex& expon)`
- `Complex pow(const Complex& base, const Complex& expon)`

### 14.1.1.4 Other general operators

- `double abs(const Complex& x)`
  
  Return the absolute value, defined as the square root of the norm.

- `double arg(const Complex& x)`
  
  Return the value defined by \(\arctan(x.imag()/x.real())\).

- `double norm(const Complex& x)`
  
  Return the value \(x.real() * x.real() + x.imag() * x.imag()\).

- `double real(const Complex& x)`
  
  Return the real part of the complex number.

- `double imag(const Complex& x)`
  
  Return the imaginary part of the complex number.

### 14.1.1.5 Comparison Operators

- `int operator != (const Complex& x, const Complex& y)`
- `int operator == (const Complex& x, const Complex& y)`
14.1.2 Fixed-point Data Type

The fixed-point data type is implemented in MLDesigner by the Fix class. This class supports a two's complement representation of a finite precision number. In fixed-point notation, the partition between the integer part and the fractional part, the binary point, lies at a fixed position in the bit pattern. Its position represents a trade-off between precision and range. If the binary point lies to the right of all bits, then there is no fractional part.

NOTE: The Fix type is still experimental.

14.1.2.1 Constructing fixed-point variables

Variables of type Fix are defined by specifying the word length and the position of the binary point. At the user interface level, precision is specified either by setting a fixed-point parameter to a (value, precision) pair, or by setting a precision parameter. The former gives the value and precision of some fixed-point value, while the latter is typically used to specify the internal precision of computations in a primitive.

In either case, the syntax of the precision is either \(x.y\) or \(m/n\), where \(x\) is the number of integer bits (including the sign bit), \(y\) and \(m\) are the number of fractional bits, and \(n\) is the total number of bits. Thus, the total number of bits in the fixed-point number (also called its length) is \(x+y\) or \(n\). For example, a fixed-point number with precision 3.5 has a total length of 8 bits, with 3 bits to the left and 5 bits to the right of the binary point.

At the source code level, methods working on Fix objects either have the precision passed as an \(x.y\) or \(m/n\) string, or as two C++ integers that specify the total number of bits and the number of integer bits including the sign bit (that is, \(n\) and \(x\)). For example, suppose you have a primitive with a precision parameter named precision. Consider the following code:

```cpp
Fix x = Fix((const char *) precision);
if (x.invalid())
    Error::abortRun(*this, "Invalid precision");
```

The precision parameter is cast to a string and passed as argument to the Fix class constructor. The error check verifies that the precision was valid.

There is a maximum value for the total length of a Fix object which is specified by the constant `FIX_MAX_LENGTH` in the file `$MLD/src/kernel/Fix.h`. The current value is 64 bits. Numbers in the Fix class are represented using two's complement notation, with the sign bit stored in the bits to the left of the binary point. There must always be at least one bit to the left of the binary point to store the sign.

In addition to its value, each Fix object contains information about its precision and error codes indicating overflow, divide-by-zero, or bad format parameters. The error codes are set when errors occur in constructors or arithmetic operators. There are also fields to specify:

a. whether rounding or truncation should take place when other Fix values are assigned to it, truncation is the default,

b. the response to an overflow or underflow on assignment, the default is saturation.
14.1.2.2 Fixed-point parameter

Parameter variables can be declared as Fix or FixArray. The precision is specified by an associated precision parameter using either one of two possible syntaxes:

- By specifying just a value itself in the dialog box. This creates a fixed-point number with the default length of 24 bits and with the position of the binary point set as required to store the integer value. For example, the value 1.0 creates a fixed-point object with precision 2.22, and the value 0.5 would create one with precision 1.23.
- By specifying a \((value, precision)\) pair. This create a fixed-point number with the specified precision. For example, the value \((2.546, 3.5)\) creates a fixed-point object by casting the double 2.546 to a Fix with precision 3.5.

14.1.2.3 Fixed-point inputs and outputs

Fix types are available in MLDesigner as a type of Particle. The conversion from an int or a double to a Fix takes place using the Fix::Fix(double) constructor which creates a Fix object with the default word length of 24 bits and the number of integer bits as required by the value. For instance, the double 10.3 will be converted to a Fix with precision 5.19, since 5 is the minimum number of bits needed to represent the integer part, 10, including its sign bit.

To use the Fix type in a primitive, the type of the portholes must be declared as fix. Primitives that receive or transmit fixed-point data have parameters that specify the precision of the input and output in bits, as well as the overflow behavior. Here is a simplified version of SDF primitive AddFix, configured for two inputs:

```cpp
defprimitive
{
    name  { AddFix }
    domain { SDF }
    derived { SDFFix }
    input
    {
        name  { input1 }
        type  { fix }
    }
    input
    {
        name  { input2 }
        type  { fix }
    }
    output
    {
        name  { output }
        type  { fix }
    }
    defparameter
    {
```
name  { OutputPrecision }  

type  { precision }  

default { 2.14 }  

desc  

{  
    Precision of the output in bits and precision of the  
    accumulation. When the value of the accumulation  
    extends outside of the precision, the OverflowHandler  
    will be called.  
}

Note that the real AddFix primitive supports any number of inputs. By default, the precision used  
by this primitive during the addition will have 2 bits to the left of the binary point and 14 bits  
to the right. Not shown here is the parameter OverflowHandler, which is inherited from the  
SDFFix primitive and defaults to saturate, that is, if the addition overflows, the result saturates,  
pegging it to either the largest positive or negative number representable. The result value, sum,  
is initialized by the following code:

protected { Fix sum; }  
begin  
{  
    SDFFix::begin();  
    sum = Fix( ((const char *) OutputPrecision) );  
    if ( sum.invalid() )  
        Error::abortRun(*this, "Invalid OutputPrecision");  
    sum.set_ovflow( ((const char*) OverflowHandler) );  
    if ( sum.invalid() )  
        Error::abortRun(*this, "Invalid OverflowHandler");  
}

The begin method checks the specified precision and overflow handler for correctness. Then, in  
the go method, the sum is used to add the input values, thus taking care that the desired precision  
and overflow handling are enforced. For example,

go  
{  
    sum.setToZero();  
    sum += Fix(input1%0);  
    checkOverflow(sum);  
    sum += Fix(input2%0);  
    checkOverflow(sum);  
    output%0 << sum;  
}

The checkOverflow method is inherited from SDFFix primitive. The protected member sum  
is an uninitialized Fix object until the begin method runs. In the begin method, it is given
the precision specified by OutputPrecision. The go method initializes it to zero. If the go method had assigned a value specified by another Fix object instead, it would acquire the precision of that other object, at the point it had be initialized.

### 14.1.2.4 Assignment and overflow handling

Once a Fix object has been initialized, its precision does not change as long as the object exists. The assignment operator is overloaded so it checks whether the value of the object to the right of the assignment fits into the precision of the left object. If not, it then takes the appropriate overflow response and sets the overflow error bit.

If a Fix object is with no arguments in the constructor, as in the protected declaration above, then that object is an uninitialized Fix. It can accept any assignment, acquiring not only its value, but also its precision and overflow handler.

The behavior of a Fix object on an overflow depends on the specifications and the behavior of the object itself. Each object has a private data field that is initialized by the constructor. If there is an overflow, the overflow handler looks at this field and uses the specified method to handle the overflow. This data field is set to saturate by default, and can be set explicitly to any other desired overflow handling method using a function called set_ovflow(<keyword>). The keywords for overflow handling methods are: saturate (default), zero_saturate, wrapped, warning. saturate replaces the original value is replaced by the maximum (for overflow) or minimum (for underflow) value representable given the precision of the Fix object. zero_saturate sets the value to zero.

### 14.1.2.5Explicitly casting inputs

In the above example, in the second line of the go method the input is added to the protected member sum, which has the side-effect of quantizing the input value to the precision of sum. The programmer also could have written the go method as follows:

```go
{
    sum = Fix(input1%0) + Fix(input2%0);
    output%0 << sum;
}
```

The behavior here is significantly different: the inputs are added using their own native precision, and only the result is quantized to the precision of sum.

Some primitives allow the programmer to select between these two different behaviors with a parameter called ArrivingPrecision. If set to YES, the input particles are not explicitly cast. They are used as they are. If set to NO, the input particles are cast to an internal precision, which is usually specified by another parameter.

Here is the (abbreviated) source of the SDF primitive GainFix, which demonstrates this point:

```go
defprimitive
{
    name { GainFix }
    domain { SDF }
```
This is an amplifier; the fixed-point output is the fixed-point input multiplied by the "gain" (default 1.0). The precision of "gain", the input, and the output can be specified in bits.

input
{
    name { input }
    type { fix }
}

output
{
    name { output }
    type { fix }
}

defparameter
{
    name { gain }
    type { fix }
    default { 1.0 }
    desc { Gain of the primitive. }
}

defparameter
{
    name { ArrivingPrecision }
    type { int }
    default { "YES" }
    desc { Flag indicating whether or not to use the arriving particles as they are: YES keeps the same precision, and NO casts them to the precision specified by the parameter "InputPrecision". }
}

defparameter
{
    name { InputPrecision }
    type { precision }
    default { 2.14 }
    desc { Precision of the input in bits. The input }
particles are only cast to this precision if the parameter "ArrivingPrecision" is set to NO.

}
}
defparameter
{
    name { OutputPrecision }
    type { precision }
    default { 2.14 }
    desc
    {
        Precision of the output in bits. This is the precision that will hold the result of the arithmetic operation on the inputs. When the value of the product extends outside of the precision, the OverflowHandler will be called.
    }
}
}

protected
{
    Fix fixIn, out;
}

begin
{
    SDFFix::begin();
    if ( ! int(ArrivingPrecision) )
    {
        fixIn = Fix( ((const char *) InputPrecision) );
        if(fixIn.invalid())
            Error::abortRun( *this, "Invalid InputPrecision" );
    }
    out = Fix( ((const char *) OutputPrecision) );
    if ( out.invalid() )
        Error::abortRun( *this, "Invalid OutputPrecision" );
    out.set_ovflow( ((const char *) OverflowHandler) );
    if(out.invalid())
        Error::abortRun( *this,"Invalid OverflowHandler" );
}

go
{ // all computations should be performed with out since // that is the Fix variable with the desired overflow // handler
    out = Fix(gain);
if ( int(ArrivingPrecision) )
{
    out *= Fix(input%0);
}
else
{
    fixIn = Fix(input%0);
    out *= fixIn;
}
checkOverflow(out);
output%0 << out;
}
}

Note that the SDFGainFix primitive and many of the Fix primitives are derived from the primitive SDFFix. SDFFix implements commonly used methods and defines two parameters: Overflow-Handler selects one of four overflow handlers to be called each time an overflow occurs, and ReportOverflow, which, if true, causes the number and percentage of overflows that occurred for that primitive during a simulation run to be reported in the wrapup method.

14.1.2.6 Constructors

Fix()
Create a Fix number with unspecified precision and value zero.

Fix(int length, int intbits)
Create a Fix number with total word length of length bits and intbits bits to the left of the binary point. The value is set to zero. If the precision parameters are not valid, an error bit is internally set so that the invalid method will return TRUE.

Fix(const char* precisionString)
Create a Fix number whose precision is determined by precisionString, which has the syntax leftbits.rightbits, where leftbits is the number of bits to the left of the binary point and rightbits is the number of bits to the right of the binary point, or rightbits/totalbits, where totalbits is the total number of bits. The value is set to zero. If the precisionString is not in the proper format, an error bit is internally set so that the invalid method will return TRUE.

Fix(double value)
Create a Fix number with the default precision of 24 total bits for the word length and set the number of integer bits to the minimum needed to represent the integer part of the number value. If the given value needs more than 24 bits to be represent, the value will be clipped and the number stored will be the largest possible under the default precision (i.e. saturation occurs). In this case an internal error bit is set so that the ovf_occurred method will return TRUE.
14.1 Scalar Numeric Types

Fix(int length, int intbits, double value)
Create a Fix with the specified precision and set its value to the given value. The number is rounded to the closest representable number given the precision. If the precision parameters are not valid, then an error bit is internally set so that the invalid method will return TRUE.

Fix(const char* precisionString, double value)
Same as the previous constructor except that the precision is specified by the given precisionString instead of as two integer arguments. If the precision parameters are not valid, an error bit is internally set so that the invalid() method will return TRUE when called on the object.

Fix(const char* precisionString, uint16* bits)
Create a Fix with the specified precision and set the bits precisely to the ones in the given bits argument. The first word pointed to by bits contains the most significant 16 bits of the representation. Only as many words as necessary to fetch the bits will be referenced from the bits argument. For example: Fix("2.14",bits) will only reference bits[0].

This constructor gets very close to the representation and is meant mainly for debugging. It may be removed in the future.

Fix(const Fix& arg)
Copy constructor. Produces an exact duplicate of arg.

Fix(int length, int intbits, const Fix& arg)
Read the value from the arg and set to a new precision. If the precision parameters are not valid, an error bit is internally set so that the invalid method will return TRUE when called on the object. If the value from the source will not fit, an error bit is set so that the ovf_occurred method will return TRUE.

14.1.2.7 Functions to set or display information about the Fix number

int len() const
Return the total word length of the Fix number.

int intb() const
Return the number of bits to the left of the binary point.

int precision() const
Return the number of bits to the right of the binary point.

int overflow() const
Return the code of the type of overflow response for the Fix number. The possible codes are:

- 0 - ovf_saturate,
- 1 - ovf_zero_saturate,
- 2 - ovf_wrapped,
- 3 - ovf_warning,
- 4 - ovf_n_types.

```cpp
int roundMode() const
    Return the rounding mode: 1 for rounding, 0 for truncation.
```

```cpp
int signBit() const
    Return TRUE if the value of the Fix number is negative, FALSE if it is positive or zero.
```

```cpp
int is_zero()
    Return TRUE if the value of the Fix number is zero.
```

```cpp
double max()
    Return the maximum value representable using the current precision.
```

```cpp
double min()
    Return the minimum value representable using the current precision.
```

```cpp
double value()
    The value of the Fix number as a double.
```

```cpp
void setToZero()
    Set the value of the Fix number to zero.
```

```cpp
void set_overflow(int value)
    Set the overflow type.
```

```cpp
void set_rounding(int value)
    Set the rounding type: TRUE for rounding, FALSE for truncation.
```

```cpp
void initialize()
    Discard the current precision format and set the Fix number to zero.
```

There are a few functions for backward compatibility:

```cpp
void set_ovflow(const char*)
    Set the overflow using a name.
```

```cpp
void Set_MASK(int value)
    Set the rounding type. Same functionality as set_rounding().
```
14.1 Scalar Numeric Types

14.1.2.8 Comparison function

```c
int compare (const Fix& a, const Fix& b)

Compare two Fix numbers. Return -1 if \( a < b \), 0 if \( a = b \), 1 if \( a > b \).
```

The following functions are for use with the error condition fields:

```c
int ovf_occurred()

Return TRUE if an overflow has occurred as the result of some operation like addition or assignment.
```

```c
int invalid()

Return TRUE if the current value of the Fix number is invalid due to it having an improper precision format, or if some operation caused a divide by zero.
```

```c
int dbz()

Return TRUE if a divide by zero error occurred.
```

```c
void clear_errors()

Reset all error bit fields to zero.
```

14.1.2.9 Operators

```c
Fix& operator = (const Fix& arg)

Assignment operator. If *this does not have its precision format set (i.e. it is uninitialized), the source Fix is copied. Otherwise, the source Fix value is converted to the existing precision. Either truncation or rounding takes place, based on the value of the rounding bit of the current object. Overflow results either in saturation, ”zero saturation” (replacing the result with zero), or a warning error message, depending on the overflow field of the object. In these cases, ovf_occurred will return TRUE on the result.
```

```c
Fix& operator = (double arg)

Assignment operator. The double value is first converted to a default precision Fix number and then assigned to *this.
```

The function of these arithmetic operators should be self-explanatory:

```c
Fix& operator += (const Fix&)
Fix& operator -= (const Fix&)
Fix& operator *= (const Fix&)
Fix& operator **= (int)
Fix& operator /= (const Fix&)
Fix operator + (const Fix&, const Fix&)
Fix operator - (const Fix&, const Fix&)
Fix operator * (const Fix&, const Fix&)
Fix operator * (const Fix&, int)
```
Fix operator * (int, const Fix&)  
Fix operator / (const Fix&, const Fix&)  
Fix operator - (const Fix&) // unary minus  
int operator == (const Fix& a, const Fix& b)  
int operator != (const Fix& a, const Fix& b)  
int operator >= (const Fix& a, const Fix& b)  
int operator <= (const Fix& a, const Fix& b)  
int operator > (const Fix& a, const Fix& b)  
int operator < (const Fix& a, const Fix& b)

Note:

- These operators are designed so that overflow does not occur in normal cases (the return value has a wider format than that of its arguments). The exception is when the result cannot be represented in a Fix with all 64 bits before the binary point.
- The output of any operation will have error codes that are the logical OR of those of the arguments to the operation, plus any additional errors that occurred during the operation (like divide by zero).
- The division operation is currently a hack: it converts to double and computes the result, converting back to Fix.
- The relational operators ==, !=, >=, <=, >, < are all written in terms of a function:
  ```cpp
  int compare(const Fix& a, const Fix& b)
  ```
  This function returns -1 if \( a < b \), 0 if \( a = b \), and 1 if \( a > b \). The comparison is exact (every bit is checked) if the two values have the same precision format. If the precisions are different, the arguments are converted to doubles and compared. Since double values only have an accuracy of about 53 bits on most machines, this may cause false equality reports for Fix values with many bits.

### 14.1.2.10 Conversions

```cpp
operator int() const
```

Return the value of the Fix number as an integer, truncating towards zero.

```cpp
operator float() const
```

```cpp
operator double() const
```

Convert to a float or a double, creating an exact result when possible.

```cpp
void complement()
```

Replace the current value by its complement.

### 14.1.2.11 Fix overflow, rounding, and errors

The Fix class defines the following enumeration values for overflow handling:

```cpp
Fix::ovf_saturate  
Fix::ovf_zero_saturate  
Fix::ovf_wrapped  
Fix::ovf_warning
```
They may be used as arguments to the `set_overflow` method, as in the following example:

```cpp
out.set_overflow(Fix::ovf_saturate);
```

The member function

```cpp
int overflow() const;
```

returns the overflow type. This returned result can be compared against the above enumerated values. Overflow types may also be specified as strings, using the method

```cpp
void set_ovflow(const char* overflow_type);
```

the `overflow_type` argument may be one of `saturate`, `zero_saturate`, `wrapped`, or `warning`.

The rounding behavior of a `Fix` value may be set by calling

```cpp
void set_rounding(int value);
```

If the argument is false, or has the value `Fix::mask_truncate`, then truncation will occur. If the argument is nonzero, for example, if it has the value `Fix::mask_truncate_round`, rounding will occur. The older name `Set_MASK` is a synonym for `set_rounding`.

The following functions access the error bits of a `Fix` result:

```cpp
int ovf_occurred() const;
int invalid() const;
int dbz() const;
```

The first function returns `TRUE` if there have been any overflows in computing the value. The second returns `TRUE` if the value is invalid, because of invalid precision parameters or a divide by zero. The third returns `TRUE` only for divide by zero.

## 14.2 Defining New Data Types

The MLDesigner contains a heterogeneous message interface, which provides a mechanism for primitives to transmit arbitrary objects to other primitives. It's design fulfills the following requirements:

- Existing primitives (primitives that were written before the message interface was added) that handle `ANYTYPE` work with message particles without change.
- Message portholes can send different types of messages during the same simulation. This is especially useful for modeling communication networks.
- It avoids copying large messages by using a reference count mechanism, as in many C++ classes (for example, string classes).
- It is possible to safely modify large messages without excessive memory allocation and deallocation.
- It is (relatively) easy for users to define their own message types. No change to the kernel is required to support new message types.
14 Using Data Types

The message type is interpreted by MLDesigner as a particle containing a message. There are three classes that implement the support for message types:

- The **Message** class is the base class from which all other message data types are derived. A user who wishes to define an application-specific message type derives a new class from Message.
- The **Envelope** class contains a pointer to a derived class from Message. When an Envelope object is copied or duplicated, the new envelope simply sets its own pointer to the pointer contained in the original. Thus, several envelopes can reference the same Message object. Each Message object contains a reference count, which tracks how many Envelope objects reference it. When the last reference is removed, the Message is deleted.
- The **MessageParticle** class is a type of Particle (like IntParticle, FloatParticle, etc.). It contains an Envelope. Ports of type message transmit and receive objects of this type.

Class **Particle** contains two member functions for message support: getMessage, to receive a message, and the << operator with an Envelope as the right argument, to load a message into a particle. These functions return errors in the base class. They are overridden in the MessageParticle class with functions that perform the expected operation.

### 14.2.1 Defining a New Message Class

Every user-defined message is derived from class Message. Certain virtual functions defined in that class must be overridden, others may optionally be overridden. Here is an example of a user-defined message type:

```cpp
#include "Message.h"

class IntVecData: public Message
{
  private:
    int len;
  init(int length, int *srcData)
  {
    len = length;
    data = new int[len];
    for (int i = 0; i < len; i++)
      data[i] = *srcData++;
  }
  public:// the pointer is public for simplicity
    int *data;
```
int length() const { return len;}

// functions for type-checking
const char* dataType() const { return "IntVecData";}  

// isA responds TRUE if given the name of the class or
// of any baseclass.
int isA(const char* typ) const
{
    if (strcmp(typ,"IntVecData") == 0) return TRUE;
    else return Message::isA(typ);
}

// constructor: makes an uninitialized array
IntVecData(int length): len(length)
{
    data = new int[length];
}

// constructor: makes an initialized array from an int array
IntVecData(int length,int *srcData) { init(length,srcData);}  

// copy constructor
IntVecData(const IntVecData& src) { init(src.len,src.data);}  

// clone: make a duplicate object
Message* clone() const { return new IntVecData(*this);}  

// destructor
"IntVecData() { delete []data; }

This message object can contain a vector of integers of arbitrary length. Some functions in the class
are arbitrary and the programmer may define them in the way that is most convenient. However,
there are some requirements.
The class must redefine the dataType method from class Message. This function returns a
string identifying the message type. This string should be identical to the name of the class. In
addition, the isA method must be defined. The isA method responds with TRUE (or 1) if given
the name of the class or of any base class. Otherwise, it returns FALSE (or 0). This mechanism
permits primitives to handle any of a whole group of message types, even for classes that are
defined after the primitive is written.
Because of the regular structure of isA function bodies, macros are provided to generate them.
The ISA_INLINE macro expands to an inline definition of the function; for example,

ISA_INLINE(IntVecData,Message)
could have been written like the example above, instead of the definition of isA to generate exactly
the same code. Alternatively, to put the function body in a .cc file, the programmer can write
int isA(const char*) const;

in the class declaration and put

    ISA_FUNC(IntVecData, Message)

in the .cc file (or wherever the methods are defined).
The class must define a copy constructor, unless the default copy constructor, generated by the
compiler which does member-wise copying, will do the job.
The class must redefine the clone method of class Message. In the example above, where
a new object is created using the new operator and the copy constructor, the form will suffice,
assuming that the copy constructor is defined.
In addition, the programmer may optionally define type conversions and printing functions if they
make sense. If a primitive which produces messages is connected to a primitive which expects
integers (or floating values, or complex values), the appropriate type conversion function is called.
The base class, Message, defines the virtual conversion functions asInt(), asFloat(),
and asComplex() and the printing method print(), see the file $MLD/src/kernel/-
Message.h for their exact types. The base class conversion functions assert a run-time error,
and the default print function returns a StringList saying

    <type>: no print method

where type is whatever is returned by dataType(). By redefining these methods, the programmer
can make it legal to connect a primitive that produces messages to a primitive that expects integer,
float, or complex particles, or he can connect to a Printer or XMgraph primitive. For the XMgraph
to work, you must define the asFloat function, for Printer to work, you must define the print
method.

14.2.2 Use of the Envelope Class

The Envelope class references objects of the Message class or derived classes. Once a mes-
sage object is placed into an envelope object, the envelope takes over the responsibility for man-
aging its memory: maintaining reference counts and deleting the message when it is no longer
needed.
The constructor, which takes a reference to a Message as its argument, copy constructor, assign-
ment operator, and the destructor of Envelope manipulate the reference counts of the referenced
Message object. An assignment simply copies a pointer and increments the reference count.
When the destructor of an Envelope is called, the reference count of the Message object is
decremented. If it becomes zero, the Message object is deleted. Because of this deletion, a
Message must never be put inside an Envelope unless it was created with the new operator.
Once a Message object is put into an Envelope it doesn’t need to be explicitly deleted. It
will "live" as long as there is at least one Envelope that contains it, and it will then be deleted
automatically.
It is possible for an Envelope to be "empty". If it is so, the empty method will return TRUE,
and the data field will point to a special "dummy message" of the DUMMY type that has no data in
it.
The dataType method of Envelope returns the datatype of the contained Message object. The methods asInt(), asFloat(), asComplex(), and print() are also "passed
through”, in a similar way, to the contained object.

Two Envelope methods are provided for convenience to make type checking simpler: type-
Check and typeError. A simple example illustrates their use:

```cpp
if (!envelope.typeCheck("IntVecData"))
{
    Error::abortRun(*this, envelope.typeError("IntVecData"));
    return;
}
```

The method typeCheck calls isA on the message contents and returns the result, so an er-
ror will be reported if the message contents are not IntVecData and are not derived from
IntVecData. Since the above code segment is common in primitives, a macro is included in
Message.h to generate it. The macro

```cpp
TYPE_CHECK(envelope,"IntVecData");
```

expands to essentially the same code as above. The typeError method generates an appropriate
error message:

```
Expected message type 'arg', got 'type'
```

To access the data, two methods are provided, myData() and writableCopy(). The myData
function returns a pointer to the contained Message-derived object. The data pointed to by this
pointer must not be modified, since other Envelope objects in the program may also contain it.
If the programmer convert its type, he should always make sure that the converted type is a const
pointer, see the programming example for UnPackInt below. This ensures that the compiler will
complain if the programmer is doing anything illegal.

The writableCopy function also returns a pointer to the contained object, but with a difference.
If the reference count is one, the envelope is emptied (set to the dummy message) and the contents
are returned. If the reference count is greater than one, a clone of the contents is made by calling
its clone() function and returned. Again the envelope is set to zero to prevent the making of
additional clones later on.

In some cases, a primitive writer will need to keep a received Message object around between
executions. The best way to do this is to have the primitive contain a member of type Envelope,
and to use this member object to hold the message data between executions. Messages should
always be kept in envelopes so that the programmer does not have to worry about memory man-
agement.

### 14.2.3 Use of the MessageParticle Class

If a porthole is of type message, its particles are objects of the class MessageParticle. A
MessageParticle is simply a particle whose data field is an Envelope, which means that
it can hold a Message in the same way that Envelope objects do.

Many methods of the Particle class are redefined in the MessageParticle class to cause
a run-time error. For example, it is illegal to send an integer, floating, or complex number to the
particle with the << operator. The conversion operators (conversion to type int, double, or
Complex) return errors by default, but can be made legal by redefining the asInt, asFloat,
or asComplex methods for a specific message type.
The principal operations on MessageParticle objects are << with an argument of type Envelope, to load a message into the particle, and getMessage(Envelope&), to transfer message contents from the particle into a user-supplied message. The getMessage method removes the message contents from the particle\(^1\). In cases where the destructive behavior of getMessage cannot be tolerated, an alternative interface, accessMessage(Envelope&), is provided. It does not remove the message contents from the particle. Promiscuous use of accessMessage in systems with large-sized messages can increase the amount of virtual memory to be reserved (though all message will be deleted eventually).

### 14.2.4 Use of Messages in Primitives

Here are a couple of simple examples of primitives that produce and consume messages. For more advanced samples, look in the MLDesigner distribution for primitives that produce or consume messages. The image processing classes and primitives, which are briefly described below in sec. 14.6.2 on page 14-44, provide a particularly rich set of examples. The matrix classes described in sec. 14.3 on page 14-23 are also good examples. The matrix classes are recognized in the MLDesigner kernel.

```cpp
defprimitive
{
    name   { PackInt }
    domain { SDF }
    desc   { Accept integer inputs and produce IntVecData messages.}

defparameter
{
    name   { length }
    type   { int }
    default { 10 }
    desc   { number of values per message }
}
input
{
    name   { input }
    type   { int }
}
output
{
    name   { output }
    type   { message }
}
ccinclude { "Message.h", "IntVecData.h" }\(^1\)
```

\(^1\)The reason for this "aggressive reclamation" policy (both here and in other places) is to minimize the number of unused messages in the system and to prevent unnecessary clones from being generated by writableCopy() by eliminating references to Message objects as soon as possible.
14.2 Defining New Data Types

```c
setup
{
    input.setSDFParams(int(length), int(length-1));
}

go
{
    int l = length;
    IntVecData * pd = new IntVecData(l);
    // Fill in message. input%0 is newest, must reverse
    for (int i = 0; i < l; i++)
        pd->data[l-i-1] = int(input%i);
    Envelope pkt(*pd);
    output%0 << pkt;
}
```

Since this is an SDF primitive, it must produce and consume a constant number of tokens on each step, so the message length must be fixed (though it is controllable with a parameter). See sec. 17.1 on page 17-1 for an explanation of the `setSDFParams` method. Notice that the `output` port-hole is declared to be of type `message`. Notice also the `ccinclude` statement, the programmer has to include the file `Message.h` in all message-manipulating primitives, and he also has to include the definition of the specific message type he wishes to use.

The code itself is fairly straightforward. An `IntVecData` object is created with the `new` command, it is filled in with data, put into an `Envelope` and is sent. Resist the temptation to declare the `IntVecData` object as a local variable: it will not work. It must reside on the heap. Here is a primitive to do the inverse operation:

```c
defprimitive
{
    name { UnPackInt }
    domain { SDF }
    desc
    {
        Accept IntVecData messages and produce integers. The first 'length' values from each message are produced.
    }
    defparameter
    {
        name { length }
        type { int }
        default { 10 }
        desc { number of values output per message }
    }
```
Because the domain is SDF, we must always produce the same number of outputs regardless of the size of the messages. The simple approach taken here is to require at least a certain amount of data or else to trigger an error and abort the execution.

The operations here are to declare an `Envelope` object `pkt`, get the data from the particle into `pkt` variable by calling the `getMessage` method, check the type, and then access the contents. Notice the cast operation, this is needed because `myData` returns a const pointer to class `Message`. It is important that the programmer converts the pointer to `const IntVecData *` and not `IntVecData *` because we have no right to modify the message through this pointer.
Many C++ compilers will not warn by default about "casting away const". We recommend turning on compiler warnings when compiling code that uses messages to avoid getting into trouble. For g++, say -Wcast-qual, for cfront-derived compilers, say +w.

If the programmer wishes to modify the message and then send the result as an output, he would call writableCopy instead of myData, modify the object, then send it on its way as seen in the previous primitive.

14.3 Matrix Data Types

The primary support for matrix types in MLDesigner is the PtMatrix class. PtMatrix is derived from the Message class, and uses the various kernel support functions for working with the Message data type as described in sec. 14.2 on page 14-15. This section discusses the PtMatrix class and how to write primitives and programs using this class.

14.3.1 Design philosophy

The PtMatrix class implements two dimensional arrays. There are four key classes derived from PtMatrix: ComplexMatrix, FixMatrix, FloatMatrix, and IntMatrix. Note that FloatMatrix is a matrix of C++ doubles. A review of matrix classes implemented by other programmers reveals two main styles of implementation: a vector of vectors, or a simple array. In addition, there are two main formats of storing the entries: column ordering, where all the entries in the first column are stored before the entries of the second column, and row ordering, where the entries are stored starting with the first row. Column ordering is how Fortran stores arrays whereas row-major ordering is the way C stores arrays.

The MLDesigner PtMatrix class stores data as a simple C array, and therefore uses row ordering. Row ordering also seems more natural for operations such as image and video processing, but it might make it more difficult to interface PtMatrix class with Fortran library calls. The limits of interfacing PtMatrix class with other software is discussed in sec. 14.3.5 on page 14-36.

The design decision to store data entries in a C array rather than in an array of vector objects has a greater effect on performance than the decision whether to use row or column ordering. There are a couple of advantages in implementing a matrix class as an array of vector class objects: referencing an entry may be faster, and it is easier to do operations on a whole row or column of the matrix, depending on whether the format is an array of column vectors or an array of row vectors. An entry lookup in an array of row vectors requires two index lookups: one to find the desired row vector in the array and one to find the desired entry of that row. A linear array, in contrast, requires a multiplication to find the location of first element of the desired row and then an index lookup to find the column in that row. For example, A[row][col] is equivalent to looking up &data + (row*numRows + col) if the entries are stored in a C array data[], whereas it is *(&rowArray + row) + col if looking up the entry in an array of vectors format.

Although the array of vectors format has faster lookups, it is also more extensive to create and delete the matrix. Each vector of the array must be created in the matrix constructor, and each vector must also be deleted by the matrix destructor. The array of vectors format also requires more memory to store the data and the extra array of vectors.

With the advantages and disadvantages of the two systems in mind, we chose to implement the PtMatrix class with the data stored in a standard C array. MLDesigner’s environment is such
that matrices are created and deleted constantly as needed by the primitives: this negates much of
the speedup gained from faster lookups. Also, it was useful to keep the design of the class simple
and the memory usage efficient because of MLDesigner’s increasing size and complexity.

14.3.2 PtMatrix Class

The PtMatrix base class is derived from the Message class so the programmer can work
with matrices using Envelope class and message-handling system. However, the Message-
Particle class is not used by the PtMatrix class. Instead, there are special MatrixEnv-
Particle classes defined to handle the type checking between the various types of matrices.
This allows the system to automatically detect incorrect connections between two primitives with
different matrix type inputs and outputs\(^2\). Also, the MatrixEnvParticle class has some
special functions not found in the standard MessageParticle class to allow easier handling
of PtMatrix class messages. A discussion of how to pass PtMatrix class objects using the
MatrixEnvParticle can be found later in this documentation.

As previously explained, there are currently four data-specific matrix classes: ComplexMatrix,
FixMatrix, FloatMatrix, and IntMatrix. Each of these classes stores its entries in a
standard C array named data, which is an array of data objects corresponding to the PtMatrix
type: Complex, Fix, double, or int. These four matrix classes implement a common set
of operators and functions. In addition, the ComplexMatrix class has a few special methods
such as conjugate() and hermitian() and the FixMatrix class has a number of special
constructors that allow the programmer to specify the precision of the entries in the matrix. As a
rule, all entries of a FixMatrix will have the same precision.
The matrix classes were designed to take full advantage of operator overloading in C++. The
result is that matrix objects can be written much like operations on scalar ones. For example, the
two-operand multiply operator * has been defined so that if A and B are matrices, A*B will return
a third matrix that is the matrix product of A and B.

14.3.3 Public Functions and Operators for the PtMatrix Class

The functions and operators listed below are implemented by all matrix classes ComplexMatrix,
FixMatrix, FloatMatrix, and IntMatrix unless otherwise noted. The symbols used are:

- XXX refers to one of the following: Complex, Fix, Float, or Int
- xxx refers to one of the following: Complex, Fix, double, or int

14.3.3.1 Functions and Operators to access entries of the Matrix

\[ \text{xxx} & \text{ entry(int i)} \]

\(^2\)We recommend, however, not to adapt this method to own types, the standard method of adding new message types
described in sec. 14.2 should be used instead. The method currently used for the matrix classes may not be supported
in future releases.
Example: A.entry(i)
Return the i-th entry of the matrix if its data storage is considered to be a linear array. This is useful for quick operations on every entry of the matrix without regard for the specific (row, column) position of that entry. The total number of entries in the matrix is defined to be numRows() * numCols(), with indices ranging from 0 to numRows() * numCols() - 1. This function returns a reference to the actual entry in the matrix so that assignments can be made to that entry. In general, functions that wish to use linear references to each entry of a matrix A should use this function instead of the expression A.data[i] because classes which are derived from PtMatrix can then overload the entry() method and reuse the same functions.

```cpp
operator [] (int row)
```
Example: A[row][column]
Return a pointer to the start of the row in the matrices data storage. (This operation is different to matrix classes defined as arrays of vectors, in which the [] operator returns the vector representing the desired row.) This operator is generally not used alone but with the [] operator defined on C arrays, so that A[i][j] will give you the entry of the matrix in the i-th row and j-th column of the data storage. The range of rows is from 0 to numRows()-1 and the range of columns is from 0 to numCols()-1.

### 14.3.3.2 Constructors

```cpp
XXXMatrix()
```
Example: IntMatrix A;
Create an uninitialized matrix. The number of rows and columns are set to zero and no memory is allocated for the storage of data.

```cpp
XXXMatrix(int numRows, int numCols)
```
Example: FloatMatrix A(3,2);
Create a matrix with dimensions numRows by numCols. Memory is allocated for the data storage but the entries are uninitialized.

```cpp
XXXMatrix(int numRows, int numCols, PortHole& portHole)
```
Example: ComplexMatrix(3,3,myPortHole);
Create a matrix of the given dimensions and initialize the entries by assigning to them values taken from the port portHole. The entries are assigned in a rasterized sequence so that the value of the first particle removed from the porthole is assigned to entry (0,0), the second particle’s value to entry (0,1), etc. It is assumed that the port has enough particles in its buffer to fill all the entries of the new matrix.

```cpp
XXXMatrix(int numRows, int numCols, XXXArrayState& dataArray)
```
Example: `IntMatrix A(2, 2, myIntArrayState);`
Create a matrix with the given dimensions and initialize the entries to the values in the given ArrayState. The values of the ArrayState fill the matrix in rasterized sequence so that entry (0, 0) of the matrix is the first entry of the ArrayState, entry (0, 1) of the matrix is the second, etc. An error is generated if the ArrayState does not have enough values to initialize the whole matrix.

XXXMatrix(const XXXMatrix& src)
Example: `FixMatrix A(B);`
This is the copy constructor. A new matrix is formed with the same dimensions as the source matrix and the data values are copied from the source.

XXXMatrix(const XXXMatrix& src, int startRow, int startCol, int numRows, int numCols)
Example: `IntMatrix A(B, 2, 2, 3, 3);`
This special "submatrix" constructor creates a new matrix whose values came from a submatrix of the source. The arguments `startRow` and `startCol` specify the starting row and column of the source matrix. The values `numRows` and `numCols` specify the dimensions of the new matrix. The sum `startRow + numRows` must not be greater than the maximum number of rows in the source matrix; similarly, `startCol + numCols` must not be greater than the maximum number of columns in the source. For example, if `B` is a matrix with dimension (4, 4), then `A(B, 1, 1, 2, 2)` would create a new matrix `A` that is a (2, 2) matrix with data values from the center quadrant of matrix `B`, so that `A[0][0] == B[1][1], A[0][1] == B[1][2], A[1][0] == B[2][1],` and `A[1][1] == B[2][2].`

The following are special constructors for the FixMatrix class that allow the programmer to specify the precision of the entries of the FixMatrix.

FixMatrix(int numRows, int numCols, int length, int intBits)
Example: `FixMatrix A(2, 2, 14, 4);`
Create a FixMatrix with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` arguments.

FixMatrix(int numRows, int numCols, int length, int intBits, PortHole& portHole)
Example: `FixMatrix A(2, 2, 14, 4);`
Create a FixMatrix with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` arguments and initialized with the values that are read from the particles contained in the porthole `portHole`.

FixMatrix(int numRows, int numCols, int length, int intBits, FixArrayState& dataArray)
Example: `FixMatrix A(2,2,14,4);`
Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` arguments and initialized with the values in the given `FixArrayState`.

There are also special copy constructors for the `FixMatrix` class that allow the programmer to specify the precision of the entries of the `FixMatrix` as they are copied from the sources. These copy constructors are usually used for easy conversion between the other matrix types. The last argument specifies the type of masking function (truncate, rounding, etc.) to be used when doing the conversion.

```
FixMatrix(const XXXMatrix& src, int length, int intBits, int round)
```

Example: `FixMatrix A(CxMatrix,4,14,TRUE);`
Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` arguments. Each entry of the new matrix is copied from the corresponding entry of the `src` matrix and converted as specified by the `round` argument.

### 14.3.3.3 Comparison operators

```
int operator == (const XXXMatrix& src)
```

Example: if `(A == B)` then ... Return `TRUE` if the two matrices have the same dimensions and every entry in `A` is equal to the corresponding entry in `B`. Return `FALSE` otherwise.

```
int operator != (const XXXMatrix& src)
```

Example: if `(A != B)` then ... Return `TRUE` if the two matrices have different dimensions or if any entry of `A` differs from the corresponding entry in `B`. Return `FALSE` otherwise.

### 14.3.3.4 Conversion operators

Each matrix class has a conversion operator so that the programmer can explicitly cast one type of matrix to another (the casting is done by copying). It would have been possible to make conversions occur automatically when needed, but because these conversions can be quite extensive for large matrices and because unexpected results might occur if the programmer did not intend for a conversion to occur, the conversions have to be used explicitly.

```
operator XXXMatrix () const
```

Example: `FloatMatrix C = A * (FloatMatrix)B;`
Convert a matrix of one type into another. These conversions allow the various arithmetic operators, such as `*` and `+`, to be used on matrices of different type. For example, if `A` in the example above is a `(3,3) FloatMatrix` and `B` is a `(3,2) IntMatrix`, then `C` is a `FloatMatrix` with dimensions `(3,2)`.
14.3.3.5 Invasive replacement operators

These operators are member functions that modify the current value of the object. In the following examples, A is usually the value (*this). All operators return *this.

```
XXXMatrix& operator = (const XXXMatrix& src)
   Example: A = B;
   This is the assignment operator: make A into a matrix that is a copy of B. If A already has allocated data storage, then the size of this data storage is compared to the size of B. If they are equal, then the dimensions of A are simply set to those of B and the entries copied. If they are not equal, the data storage is freed and reallocated before copying.

XXXMatrix& operator = (xxx value)
   Example: A = value;
   Assign each entry of A to have the given value. Memory management is handled as in the previous operator. Note: this operator is targeted for deletion. Do not use it.

XXXMatrix& operator += (const XXXMatrix& src)
   Example: A += B;
   Perform the operation A.entry(i) += B.entry(i) for each entry in A. A and B must have the same dimensions.

XXXMatrix& operator += (xxx value)
   Example: A += value;
   Add the scalar value to each entry in the matrix.

XXXMatrix& operator -= (const XXXMatrix& src)
   Example: A -= B;
   Perform the operation A.entry(i) -= B.entry(i) for each entry in A. A and B must have the same dimensions.

XXXMatrix& operator -= (xxx value)
   Example: A -= value;
   Subtract the scalar value from each entry in the matrix.

XXXMatrix& operator *= (const XXXMatrix& src)
   Example: A *= B;
   Perform the operation A.entry(i) *= B.entry(i) for each entry in A. A and B must have the same dimensions. Note: this is an element-wise operation and is not equivalent to A = A * B.

XXXMatrix& operator *= (xxx value)
   Example: A *= value;
   Multiply each entry of the matrix by the scalar value.
```
14.3 Matrix Data Types

XXXMatrix& operator /= (const XXXMatrix& src)
Example: A /= B;
Perform the operation A.entry(i) /= B.entry(i) for each entry in A
(Student multiplication). A and B must have the same dimensions.

XXXMatrix& operator /= (xxx value)
Example: A /= value
Divide each entry of the matrix by the scalar value. The scalar value must be
non-zero.

XXXMatrix& identity()
Example: A.identity();
Change A to be an identity matrix so that each entry on the diagonal is 1 and all
off-diagonal entries are 0.

14.3.3.6 Non-invasive operators (these return a new matrix)

XXXMatrix operator - ()
Example: B = -A;
Return a new matrix such that each element is the negative of the element of the
source.

XXXMatrix operator ~() 
Example: B = ~A;
Return a new matrix that is the transpose of the source.

XXXMatrix operator ! ()
Example: B = !A;
Return a new matrix which is the inverse of the source.

XXXMatrix operator ^(int exponent)
Example: B = A^2;
Return a new matrix which is the source matrix to the given exponent power. The
exponent can be negative, in which case the exponent is first treated as a positive
number and the final result is then inverted. So A^2 == A*A and A^(-3) ==

XXXMatrix transpose()
Example: B = A.transpose();
This is the same as the ~operator but called by a function name instead of an
operator.

XXXMatrix inverse()
Example: B = A.inverse();
This is the same as the ! operator but called by a function name instead of an
operator.
ComplexMatrix conjugate()
Example: ComplexMatrix B = A.conjugate();
Return a new matrix such that each element is the complex conjugate of the
source. This function is defined for the ComplexMatrix class only.

ComplexMatrix hermitian()
Example: ComplexMatrix B = A.hermitian();
Return a new matrix which is the Hermitian Transpose (conjugate transpose) of
the source. This function is defined for the ComplexMatrix class only.

14.3.3.7 Non-member binary operators

XXXMatrix operator + (const XXXMatrix& left,
const XXXMatrix& right)
Example: A = B + C;
Return a new matrix which is the sum of the first two. The left and right source
matrices must have the same dimensions.

XXXMatrix operator + (const xxx& scalar,
const XXXMatrix& matrix)
Example: A = 5 + B;
Return a new matrix that has entries of the source matrix added to a scalar value.

XXXMatrix operator + (const XXXMatrix& matrix,
const xxx& scalar)
Example: A = B + 5;
Return a new matrix that has entries of the source matrix added to a scalar value.
(This is the same as the previous operator but with the scalar on the right.)

XXXMatrix operator - (const XXXMatrix& left,
const XXXMatrix& right)
Example: A = B - C;
Return a new matrix which is the difference of the first two. The left and right source
matrices must have the same dimensions.

XXXMatrix operator - (const xxx& scalar,
const XXXMatrix& matrix)
Example: A = 5 - B;
Return a new matrix that has the negative of the entries of the source matrix added
to a scalar value.

XXXMatrix operator - (const XXXMatrix& matrix,
const xxx& scalar)
Example: A = B - 5;
Return a new matrix such that each entry is the corresponding entry of the source
matrix minus the scalar value.
XXXMatrix operator * (const XXXMatrix& left,
    const XXXMatrix& right)
  Example: A = B * C;
  Return a new matrix which is the matrix product of the first two. The left and
  right source matrices must have compatible dimensions, i.e., A.numCols() ==
  B.numRows().

XXXMatrix operator * (const xxx& scalar,
    const XXXMatrix& matrix)
  Example: A = 5 * B;
  Return a new matrix that has entries of the source matrix multiplied by a scalar
  value.

XXXMatrix operator * (const XXXMatrix& matrix,
    const xxx& scalar)
  Example: A = B * 5;
  Return a new matrix that has entries of the source matrix multiplied by a scalar
  value. (This is the same as the previous operator but with the scalar on the right.)

### 14.3.3.8 Miscellaneous functions

- **int numRows()**
  Return the number of rows in the matrix.

- **int numCols()**
  Return the number of columns in the matrix.

- **Message* clone()**
  Example: IntMatrix *B = A.clone();
  Return a copy of *this.

- **StringList print()**
  Example: A.print()
  Return a formatted StringList that can be printed to display the contents of
  the matrix in a reasonable format.

- **XXXMatrix& multiply (const XXXMatrix& left,
    const XXXMatrix& right,
    XXXMatrix& result)**
  Example: multiply(A,B,C);
  This is a faster 3 operand form of matrix multiply such that the result matrix is
  passed as an argument so that we avoid the extra copy step that is involved when
  we write C = A * B.

- **const char* dataType()**
  Example: A.dataType()
  Return a string that specifies the name of the type of matrix. The strings are
  ComplexMatrix, FixMatrix, FloatMatrix, and IntMatrix.

- **int isA(const char* type)**
  Example: if(A.isA("FixMatrix")) then ...
  Return TRUE if the argument string matches the type string of the matrix.
14.3.4 Writing Primitives Using the PtMatrix Class

This section describes how to use the matrix data classes when writing primitives. Some examples are given here but the programmer should refer to the primitives in $MLD/MLD_Libraries/sdf/matrix/*.pl and $MLD/MLD_Libraries/sdf/image/*.pl for more examples.

14.3.4.1 Memory management

The most important thing to understand about the use of matrix data classes in the MLDesigner environment is that primitives that intend to output the matrix in a particle should allocate memory for the matrix but never delete that matrix. Memory reclamation is done automatically by the reference-counting mechanism of the Message class. Strange errors will occur if the primitive deletes the matrix before it is used by another primitive later in the execution sequence.

14.3.4.2 Naming conventions

Primitives that implement general-purpose matrix operations usually have names with the _M suffix to distinguish them from primitives that operate on scalar particles. For example, the SDF primitive Gain_M multiplies an input matrix by a scalar value and outputs the resulting matrix. This is in contrast to SDF primitive Gain, which multiplies an input value held in a FloatParticle by a double and puts that result in an output FloatParticle.

14.3.4.3 Include files

To use the PtMatrix classes in a primitive, the file Matrix.h has to be included either in its .h or .cc file. If the primitive has a matrix data member, then the declaration

```c
hinclude { "Matrix.h" }
```

needs to be in the Primitive definition. Otherwise, the declaration

```c
ccinclude { "Matrix.h" }
```

is sufficient.

14.3.4.4 Input portholes

To declare an input port that accepts matrices, the following syntax is used:

```c
input
{
  name { inputPortHole }
  type { FLOAT_MATRIX_ENV }
}
```

The syntax is the same for output ports. The type field can be COMPLEX_MATRIX_ENV, FLOAT_MATRIX_ENV, FIX_MATRIX_ENV, or INT_MATRIX_ENV. The icons created by MLDesigner will have terminals that are thicker and that have larger arrow points than the terminals for scalar particle types. The colors of the terminals follow the pattern of colors for scalar data types, e.g., blue represents float and FloatMatrix.

The syntax to extract a matrix from the input porthole is:
14.3 Matrix Data Types

Envelope inPkt;
(inputPortHole%0).getMessage(inPkt);
const FloatMatrix& inputMatrix = *(const FloatMatrix *)inPkt.myData();

The first line declares an Envelope, which is used to access the matrix. Details of the Envelope class are given in sec. 14.2.2 on page 14-18. The second line fills the envelope with the input matrix. Note that, because of the reference-counting mechanism, this line does not make a copy of the matrix. The last two lines extract a reference to the matrix from the envelope. It is up to the programmer to make sure that the cast agrees with the definition of the input port. Because multiple envelopes might reference the same matrix, a primitive is generally not permitted to modify the matrix held by the Envelope. Thus, the function myData() returns a const Message *. This can be cast to be a const FloatMatrix * and then be de-referenced and assigned the value to inputMatrix. It is generally better to handle matrices by reference instead of pointers because it is easier to write A + B rather than *A + *B when working with matrix operations. Primitives that wish to modify an input matrix should access it by using the writableCopy method, as explained in sec. 14.2.2 on page 14-18.

14.3.4.5 Allowing delays on inputs

The cast to const FloatMatrix * above is not always safe. Even if the source primitive is known to provide matrices of the appropriate type, a delay on the arc connecting the two primitives can cause problems. In particular, delays in data flow domains are implemented as initial particles on the arcs. These initial particles are given the value "zero" as defined by the type of particle. For Message particles, "zero" is an uninitialized Message particle containing a "dummy" data value. This dummy Message will be returned by the myData method in the third line of the above code fragment. The dummy message is not a FloatMatrix, rendering the above cast invalid. A primitive that expects matrix inputs has to have code to handle empty particles. An example is:

```cpp
if(inPkt.empty())
{
    FloatMatrix& result = *(new FloatMatrix(int(numRows),
                                             int(numCols)));
    result = 0.0;
    output%0 << result;
}
```

There are many ways that an empty input can be interpreted by a primitive which operates on matrices. For example, a primitive multiplying two matrices can simply output a zero matrix if either input is empty. A primitive adding two matrices can output whichever input is not empty. In the example above, the output matrix has the dimensions as set by the parameters of the primitive so that any primitive that uses this output will have valid data. A possible alternative to outputting a zero matrix is to simply pass that empty MessageParticle along. This approach, however, can lead to counterintuitive results. Suppose that the empty message reaches a display primitive like TkText, which will attempt to call the print() method of the object. An empty message has a print() method that results in a message like
<type>: no print method

This is most likely to prove extremely confusing to users, so it is strongly recommend that each matrix primitive handle the empty input in a reasonable way, and produce a non-empty output.

### 14.3.4.6 Matrix outputs

To put a matrix into an output porthole, the syntax is:

```cpp
FloatMatrix& outMatrix = *(new FloatMatrix(someRow, someCol));
// ... do some operations on the outMatrix
outputPortHole%0 << outMatrix;
```

The last line is similar to outputting a scalar value. This is because the `<<` operator has been overloaded for `MatrixEnvParticle` to support `PtMatrix` class inputs. The standard use of the `MessageParticle` class requires the programmer to put his message into an envelope first and then use `<<` on the envelope (see sec. 14.2.2 on page 14-18), but this has been specialized so that the extra operation of creating an envelope first is not explicit.

Here is an example of a complete primitive definition with input and output matrices:

```ml
defprimitive
{
    name { Mpy_M } 
    domain { SDF } 
    desc {
        Does a matrix multiplication of two input Float matrices A and B to produce matrix C. 
        Matrix A has dimensions (numRows,X). 
        Matrix B has dimensions (X,numCols). 
        Matrix C has dimensions (numRows,numCols). 
        the programmer need only specify numRows and numCols. 
        An error will be generated automatically if the number of columns in A does not match the number of columns in B. 
    }
    input 
    {
        name { Ainput } 
        type { FLOAT_MATRIX_ENV } 
    }
    input 
    {
        name { Binput } 
        type { FLOAT_MATRIX_ENV } 
    }
    output 
```
14.3 Matrix Data Types

```c
{  
  name { output }  
  type { FLOAT_MATRIX_ENV }  
}
defparameter  
{  
  name { numRows }  
  type { int }  
  default { 2 }  
  desc { The number of rows in Matrix A and Matrix C. }  
}
defparameter  
{  
  name { numCols }  
  type { int }  
  default { 2 }  
  desc { The number of columns in Matrix B and Matrix C }  
}
ccinclude { "Matrix.h" }  

go  
{  
 // get inputs  
 Envelope Apkt;  
 (Ainput%0).getMessage(Apkt);  
 const FloatMatrix& Amatrix =  
       *(const FloatMatrix *)Apkt.myData();  

 Envelope Bpkt;  
 (Binput%0).getMessage(Bpkt);  
 const FloatMatrix& Bmatrix =  
       *(const FloatMatrix *)Bpkt.myData();  

 // check for "null" matrix inputs, which could be  
 // caused by delays on the input line  
 if(Apkt.empty() || Bpkt.empty())  
 {  
   // if either input is empty, return a zero  
   // matrix with the state dimensions  
   FloatMatrix& result = *(new FloatMatrix(int(numRows),  
               int(numCols)));  

   result = 0.0;  
   output%0 << result;  
 }  
 else  
 {  
   // Amatrix and Bmatrix are both valid  
```
if((Amatrix.numRows() != int(numRows)) || 
    (Bmatrix.numCols() != int(numCols)))
{
    Error::abortRun(*this,"Dimension size of FloatMatrix",
                   " inputs do not match the given ",
                   "state parameters.");
    return;
}
// do matrix multiplication
FloatMatrix& result = *(new FloatMatrix(int(numRows),
                                          int(numCols)));
// we could write
//       result = Amatrix * Bmatrix;
// but the following is faster
multiply(Amatrix,Bmatrix,result);
output%0 << result;
}

14.3.5 Future Extensions

After reviewing the libraries of numerical analysis software that is freely available on the Internet, it is clear that it would be beneficial to extend the PtMatrix class by adding those well-tested libraries as callable functions. Unfortunately, many of those libraries are currently only available in Fortran, and there are some incompatibilities with Fortran’s column ordering and C’s row major ordering. Those problems will still exist even if the Fortran code will be converted to C. There are a few groups which are currently working on C++ ports of the numerical analysis libraries. One notable group is the Lapack++[DP93] project which is developing a flexible matrix class of their own, besides porting the Fortran algorithms of Lapack into C++. This might possibly be included in a future release.

14.4 File and String Types

There are two experimental types in MLDesigner that support non-numeric computation. These types represent the beginnings of an effort to extend MLDesigner’s data flow model to ”non-data flow” problems such as scheduling and design flow. Their interfaces are still being developed, therefore they should be expected to be changed in future releases. Any suggestion on how to improve the interface and functionality of these two types will be welcomed.

14.4.1 File Type

The file type is implemented by the classes FileMessage and FileParticle, which are derived from Message and Particle. It uses the reference-counting mechanism of the
Message and Envelope classes to ensure that files are not deleted until no longer needed. It is, however, recommend that the programmer creates the Message interface described in sec. 14.2 for his own types. The File type adds the following functions to Message.

### 14.4.1 Constructors

**FileMessage()**
Create a new file message with a unique filename. By default, the file will be deleted if no file messages reference it.

**FileMessage(const char* name)**
Create a new file message with the given filename name. By default, the file will not be deleted if no file messages reference it.

**FileMessage(const FileMessage& src)**
Create a new file message containing the same filename as the given file message. By default, the file will not be deleted if no file messages reference it.

### 14.4.2 Operations

**const char* fileName()**
Return the filename contained in this message.

**StringList print()**
Return the filename contained in this message in a StringList object.

**void setTransient(int transient)**
Set the status of the file. If transient is TRUE, the file will be deleted if no file messages are reference it; if FALSE, it will not be deleted.

### 14.4.2 String Type

The string type is implemented by the classes StringMessage and StringParticle, which are derived from Message and Particle. It contains an InfString object. InfString is a version of StringList that allows limited modification and is used to interface C++ to Tcl. Again, it uses the reference-counting mechanism of the Message and Envelope classes to ensure that strings are not deleted until no longer needed. StringMessage is currently very simple, it adds the following functions to Message.

#### 14.4.2.1 Constructors

**StringMessage()**
Create a new string message with an empty string.

**StringMessage(const char* name)**
Create a new string message with a copy of the given string. The given string can be deleted, since the new message does not reference it.

**StringMessage(const StringMessage& src)**
Create a new string message containing the same string as the given string message. Again, the string is copied.
14.4.2.2 Operations

StringList print()

Return the string contained in this message in a StringList object.

14.5 Manipulating Particles of Type anytype

MLDesigner allows primitives to declare that inputs and outputs are of type anytype. A primitive may need to do this, for example, if it simply copies its inputs without regard to type, as in the case of a Fork primitive, or if it calls a generic function that is overloaded by every data type, such as sink primitives which call the print method of the type. The following is an example of a primitive that operates on anytype particles:

```c
defprimitive
{
  name {Fork}
  domain {SDF}
  desc { Copy input particles to each output. }

  input
  {
    name { input }
    type { anytype }
  }

  outmulti
  {
    name { output }
    type { =input }
  }

  location { SDF main library }

  ccinclude {"Message.h","Type.h" }

  go
  {
    OutSDFMPHIter nextp(output);
    OutSDFPort* p;
    Particle& pp = input%0;
    Type *tType;
    if((pp.type() == DATASTRUCT) && (tType = (Type*)pp))
    {
      while ((p = (OutSDFPort*)nextp.next()) != 0)
        (*p)%0 << tType->clone();
    }
```
else
{
    while ((p = (OutSDFPort*)nextp.next()) != 0)
    { (*p)%0 = pp; }
}

It should be noticed that in the definition of the output port, the primitive simply says that its output type will be the same as the input type. Ptlang translates this definition into an ANYTYPE output porthole and a statement in the primitive constructor that reads

    output.inheritTypeFrom(input);

During module setup, the MLDesigner kernel assigns actual types to ANYTYPE ports, making use of the types of connected ports and inheritTypeFrom connections. For example, if a Fork input is connected to an output port of type INT, the Fork input’s type becomes INT, and, because of the inheritTypeFrom connection, the output will do, too. At runtime, there is no such thing as an ANYTYPE porthole. Every port has been resolved to some specific data type, which can be obtained from the porthole using the resolvedType() method. However, this mechanism does not differ among the various subclasses of Message, so if the programmer is using Message particles, he still needs to check the actual type of each Message received.

Porthole type assignment is really a fairly complex and subtle algorithm, which is discussed further in [BH97]. The important properties for a primitive writer to know are these:

- If an input port has a specific declared type, it is guaranteed to receive particles of that type. For reasons mentioned in sec. 13.5.3 on page 13-22, it should be preferred to explicitly cast input particles to the desired type, as in

  ```
  go
  {
      double value = double(in%0);
      ...
  }
  ```

  but this is not strictly necessary in the current system.

- In simulation domains, an output port is NOT guaranteed to transmit particles of its declared type. The actual resolved type of the porthole will be determined by the connected input porthole. Therefore, the programmer should always allow type conversion of the value computed by the primitive into the actual type of the output particle. This happens implicitly when he writes something like

  ```
  out%0 << t;
  ```

  because this expands into a call of the particle’s virtual method for loading a value of the given type. But assuming that the programmer knows the exact type of particle in the porthole, say by writing something like `(FloatParticle&), (out%0)` is very unsafe.
14 Using Data Types

- In code generation domains, it is usually critical that the output porthole’s actual type be what the primitive writer expected. Most codegen domains therefore splice type conversion primitives into the schematic when input and output ports of different declared types are connected. In this way, both connected primitives will see the data type they expect, and the necessary type conversion is handled transparently.

- The component portholes of a multiporthole are separately type-resolved. Thus, if an input multiporthole is declared ANYTYPE, its component portholes might have different types at runtime. The component portholes of an output multiporthole can have different resolved types in any case, because they might be connected to inputs of different types.

- It is rarely a good idea to declare a pure ANYTYPE output porthole. Its type should be equated to some input porthole using the plang = port notation instead or an explicit inheritTypeFrom call. This ensures that the type resolution algorithm can succeed. A "pure ANYTYPE" output will work only while connected to an input of determinable type. If it’s connected to an ANYTYPE input, the kernel will be unable to resolve a type for the connection. By providing a = type declaration, the programmer allows the kernel to choose an appropriate particle type for an ANYTYPE-to-ANYTYPE connection.

14.6 Unsupported Types

There are a number of data types in MLDesigner that are NOT recommended to be used by external developers because they are either insufficiently mature or most likely to be changed in the future. This section briefly describes those classes.

14.6.1 Sub-Matrices

The MLDesigner kernel contains a set of matrices to support efficient computation with sub-matrices. These classes were developed specifically for the experimental multidimensional SDF (MDSDF) domain and will probably be implemented differently in a future release.

There are four sub-matrix classes, one for each concrete matrix class: ComplexSubMatrix, FixSubMatrix, FloatSubMatrix, and IntSubMatrix. Each of them inherits from the corresponding PtMatrix class. A sub-matrix contains a reference to a “parent” matrix of the same type, and modifies its internal data pointers and matrix size parameters to reference a rectangular region of the parent’s data. The constructors for the submatrix classes have arguments that specify the region of the parent matrix referenced by the sub-matrix.

As for matrices, the following descriptions of sub-matrices uses the writing convention that XXX means Complex, Fix, Float, or Int, and xxx means Complex, Fix, double, or int.

14.6.1.1 Submatrix constructors

XXXSubMatrix()
Create an uninitialized matrix.

XXXSubMatrix(int numRows, int numCols)
Create a regular matrix with dimensions numRows by numCols; return a new submatrix with this matrix as its parent. Memory is allocated for the data storage but the entries are uninitialized.
XXXSubMatrix(XXXSubMatrix& src, int sRow, int sCol, int nRows, int nCols)

Create a sub-matrix and initialize it to reference the region of the parent matrix starting at (sRow, sCol) and of size (nRows, nCols). The parent matrix is the same as the parent matrix of src. The given dimensions must fit into the parent matrix, or an error will be flagged. Unlike the ”sub-matrix” constructors in the regular matrix classes, this constructor does not copy matrix data.

XXXSubMatrix(const XXXSubMatrix& src)

Make a duplicate of the src sub-matrix. The parent of the new matrix is the same as the parent of src.

14.6.1.2 Operations

Sub-matrices support all operations supported by the regular matrix classes. Because the matrix classes uniformly use only the entry() and operator [] member functions to access the data, the sub-matrix classes need only to override these functions, and all matrix operations become available on sub-matrices.

xxx& entry(int i)

Return the i-th entry of the sub-matrix when its data storage is considered to be a linear array.

xxx* operator [](int row)

Return a pointer to the start of the row of the sub-matrices data storage.

14.6.1.3 Using sub-matrices in primitives

Sub-matrices are not currently useful in general-purpose data flow primitives. Rather, they were developed to provide an efficient means of referencing portions of a single larger matrix in the multi-dimensional synchronous data flow (MDSDF) domain.

A short summary: unlike other domains, the MDSDF kernel does not transfer particles through FIFO buffers. Instead, each geodesic keeps a single copy of a ”parent” matrix, that represents the ”current” two-dimensional datablock. Each time a primitive fires, it obtains a sub-matrix that references this parent matrix with the getOutput() function of the MDSDF input port class. For example, a primitive might contain:

FloatSubMatrix* data = (FloatSubMatrix*) (input.getInput());

Note that this is not really getting a matrix, but a sub-matrix that references a region of the current data matrix. The size of the sub-matrix has been set by the primitive in its initialization code by calling the setMDSDFParams() function of the port. To write data to the output matrix, the primitive gets a sub-matrix which references a region of the current output matrix and writes to it with a matrix operator. For example,

FloatSubMatrix* result = (FloatSubMatrix*) (output.getOutput());
result = -data;
Because the sub-matrices are only references to the current matrix on each arc, they must be deleted after use:

```cpp
delete &input;
delete &result;
```

Here is a simplified example of a complete MDSDF primitive:

```cpp
defprimitive
{
    name { Add }
    domain { MDSDF }
    desc
    {
        Matrix addition of two input matrices A and B to produce matrix C. All matrices must have the same dimensions.
    }
    version { %W% %G% }
    author { Mike J. Chen }
    location { MDSDF library }

    input
    {
        name { Ainput }
        type { FLOAT_MATRIX }
    }
    input
    {
        name { Binput }
        type { FLOAT_MATRIX }
    }
    output
    {
        name { output }
        type { FLOAT_MATRIX }
    }
    defparameter
    {
        name { numRows }
        type { int }
        default { 2 }
        desc { The number of rows in the input/output matrices. }
    }
    defparameter
    {
        name { numCols }
```
14.6 Unsupported Types

declare { int }
default { 2 }
desc { The number of columns in the input/output matrices. }
}
ccinlude { "SubMatrix.h" }

setup
{
    Ainput.setMDSDFParams(int(numRows), int(numCols));
    Binput.setMDSDFParams(int(numRows), int(numCols));
    output.setMDSDFParams(int(numRows), int(numCols));
}

go
{
    // get a SubMatrix from the buffer
    FloatSubMatrix& input1 = *(FloatSubMatrix*)(Ainput.getInput());
    FloatSubMatrix& input2 = *(FloatSubMatrix*)(Binput.getInput());
    FloatSubMatrix& result = *(FloatSubMatrix*)(output.getOutput());

    // compute addition, putting result into output
    result = input1 + input2;

    delete &input1;
    delete &input2;
    delete &result;
}

14.6.1.4 Sub-Matrix "Particles"

The Ptolemy language type of submatrices is FLOAT_MATRIX, INT_MATRIX, and so on. This is not documented in the Modeling Guide and is likely to change in a future release. Each of these Ptolemy language types is implemented by a sub-class of Particle: IntMatrixParticle, FloatMatrixParticle, FixMatrixParticle and ComplexMatrixParticle. These particle classes exist only for setting up the portholes and performing type-checking. They are never created or passed around during a simulation. Instead, sub-matrices are created and destroyed by the MDSDF kernel and the primitives as described above.
14 Using Data Types

14.6.2 Image Particles

A set of experimental image data types, designed to make it convenient to manipulate images and video sequences in MLDesigner, were defined by Paul Haskell. They are based on MLDesigner’s built-in Message type, described above. A library of primitives that uses these image data types can be found in the image library of the DE domain.

This set of classes is being replaced by the $\text{PtMatrix}$ classes, and the SDF image classes now all use $\text{PtMatrix}$. We give here a brief introduction to the image data types used in the DE domain, although a new user should consider using $\text{PtMatrix}$ classes instead. Class definitions can be found in $\text{MLD/MLD_Libraries/DE/kernel}$.

The base class of all the image classes is called $\text{BaseImage}$. It has some generic methods and members for manipulating images. Most of the methods are redefined in the derived classes. The $\text{fragment}$ method partitions an image into many smaller images, which together represent the same picture as the original. The $\text{assemble}$ method combines many small images which make up a single picture into a single image that contains the picture. The $\text{fragment}$ method works recursively, so an image that has been produced by a previous $\text{fragment}$ call can be further fragmented. $\text{assemble}$ always produces a full-sized image from fragments, however small.

Use of the $\text{size}$, $\text{fullSize}$, and $\text{startPos}$ members varies within each subclass. Typically the $\text{size}$ variable holds the number of pixels that an object is storing. If an object is not produced by $\text{fragment()}$, then $(\text{size} == \text{fullSize})$. If the object is produced by a $\text{fragment()}$ call, $\text{size}$ may be less than or equal to $\text{fullSize}$. An object’s $\text{fullSize}$ may be bigger or smaller than $\text{width} \times \text{height}$. It would be bigger, for example, in $\text{DCTImage}$, where the amount of allocated storage must be rounded up to be a multiple of the block size. It would be smaller, for example, for an object that contains run-length coded video.

The $\text{frameId}$ variable is used during $\text{assemble}$. Fragments with the same $\text{frameId}$’s are assembled into the same image. So, it is important that different frames from the same source have different $\text{frameIds}$.

The comparison operators $==, !=, <, >, etc.$ compare two objects’ $\text{frameId}$’s. They can be used to resequence images or to sort image fragments.

The copy constructor and $\text{clone}$ methods have an optional integer argument. If a non-zero argument is provided, then all parameter values of the copied object are copied to the created object, but none of the image data is copied. If no argument or a zero argument is provided, then the image data is copied as well. Classes derived from $\text{BaseImage}$ should maintain this policy.

The $\text{GrayImage}$ class, derived from $\text{BaseImage}$, is used to represent gray-scale images. The $\text{DCTImage}$ class is used to represent images or image fragments that have been encoded using the discrete-cosine transform. The $\text{MVImage}$ class is a bit more specialized. It stores a frame’s worth of motion vectors.
In designing new primitives that use data structures it is very important to understand how to instantiate a data structure, when it should be created using new (in the heap), when an instance of a data structure may be deleted and when not. The incorrect use of the data structure mechanism can lead to memory leaks or to unexpected behavior.

15.1 Initializing Data Structures

The DsHandler class holds a list with all defined data structures that are loaded at one given time. To properly use these data structures, DsHandler class provides static methods to access the class objects or to create value objects.

Methods for handling data structure class objects:

- `const TypeClass& DsHandler::findClass (const String& pName);`
- `const TypeClass* DsHandler::findClassPointer (const String& pName);`
- `bool DsHandler::isDataStructure (const String& pName);`

`findClass` searches and returns a const reference to a data structure by its name. When the data structure is not found it throws `NotFoundDataTypeException`. The second method, `findClassPointer`, returns a const pointer to a specified data structure and NULL when is not found. With `isDataStructure` you can test if a given data structure exists (is loaded) or not. Do not try to change the references returned by this methods.

Methods to create data structure values:

- `Type& DsHandler::newValue (const Kernel::String& pName);`
- `Type& DsHandler::newInteger ();`
- `Type& DsHandler::newFloat ();`
- `Type& DsHandler::newIntVector (unsigned int pLength, int pDefault=0);`
- `Type& DsHandler::newFloatVector (unsigned int pLength, double pDefault=0);`
- `Type& DsHandler::newValueFromString (const Kernel::String& pStr);`
All these methods throw `NotFoundDataTypeException` when the specified data structure is not found.

In primitives design use `TypeRef` class in case you have a non const Type object and `ConstTypeRef` when you have a const Type object. The life time of this object determines the creation and releasing of a data structure value.

Example:

```go
{  
    TypeRef tData=DsHandler::newValue("Root.Protocol.TCPProtocol");  
    .... // do something with it  
}  
// at the exit of the method the object is deleted and  
the TypeRef destructor is called. This frees the data  
structure value.
```

You can use directly only `TypeRef` class constructor to instantiate a data structure (without using `DsHandler`).

```cpp
TypeRef tData("Root.Protocol.TCPProtocol");  
// creates an objects that holds a reference to a  
// TCPProtocol data structure  
```

The old methods are still available:

```cpp
const Type* DsHandler::findStructure (const char* pName);  
    Type* DsHandler::makeNewStructure (const char* pName);  
    Type* DsHandler::makeNewInt();  
    Type* DsHandler::makeNewFloat();  
    Type* DsHandler::makeNewFromString (const char* pString)  
        throw ( DataTypeSyntaxException* );
```

Please take care if you use `findStructure` method. It is still available for compatibility but now it generates memory leak. It is therefore recommended to replace this method with `findClass` method.

**NOTE:** You can call these methods directly on the class, no object instance is needed.

## Data Structure Values

During the simulation a certain number of data structure values is generated. MLDesigner 2.3 introduces a mechanism that saves you from the overhead of creating, duplicating or releasing data structure values. MLDesigner uses a pool mechanism to reuse created values that are no longer needed. Every data structure has a reference counter that is increased when an object reference to it, and is decrease every time this object is no longer referencing to the value. When reference counter reaches zero means the value is free and is put in a pool of free values. A clone method looks first if there are any free data structures in the pool to be reused, if not a new data structure
is created. Use a reference class (TypeRef or derivated) every time you want to work with data structures. Every time a method returns Type& or Type* you can assign it to a TypeRef or a derivated class. When it returns a const reference, you have to use the ConstTypeRef class. As a general rule you have to know that non const references returned by methods are in fact copies of the original data structure inside and every time a method returns a const reference, this is a reference to the real object inside. This is the reason why expressions like tVector[i] = tElement don’t work.

15.2 Using Data Structures

Two types of operations can be executed on data structures:

- Generic type operations - operations that can be executed on every data structure regardless of its type.
- Type specific operations - operations defined only for specific data structures

15.2.1 Generic Type Operations

Methods provided by the data structure class are listed in table 15.1.

15.2.2 Type Specific Interfaces

15.2.2.1 Base Types Interface

You can perform the following operations on base types:

- read its value (using the cast operator).
- set the value by calling setValue(Type); or by changing the value of the reference returned by cast operators.

Example:

```cpp
IntegerR tInteger; //create a new integer
tInteger.setValue(123); //set the value 123 to this integer.
tInteger = 321;
int IntValue = tInteger; //read the value (call the cast to //int operator)
```

This example creates a special reference class to an integer. When the reference life time ends, the value is set as free and becomes available for further use.

15.2.2.2 Vector Data Structure Interface

Numeric vectors provide two kinds of type specific operations. On the one hand there are methods to manage the vectors, such as setting the length and the default value and on the other hand there
are methods to use vectors as values i.e., to set and get the vector elements.

You can manage the vector’s length with these methods:

```cpp
void setLength (int pLength);
```
sets the length of a vector for the first time before initialization.

```cpp
int getLength () const;
```
returns the vector’s length.

```cpp
void changeLength (int pLength);
```
with this method you can change the size of an already initialized vector during a simulation.

Other methods for vector management are:

```cpp
void setDefault (T pDefault);
T getDefault ();
```
methods with whom you can set a default value for vector elements. This make sense on an uninitialized vector. When you create a new vector, all its elements are set with this default value.

To operate over vector elements use the following methods:

```cpp
Type& operator [] (int index) const;
Type& getElement (unsigned int index);
int getIntElement (unsigned int index);
double getFloatElement (unsigned int index);
void setElement (unsigned int index, T pValue);
```

With this methods it is possible to access vector elements. Return methods are also defined as const methods that return a const reference. In case of a generic vector, operator [] returns a copy of the value inside the vector, therefore an expression like `tVector[i] = tElement` does not work and generates memory leak. All this methods throw `IndexOutOfRangeException` in case the specified index does not exist.

**Example 1:**

```cpp
// create a vector of ten integers, all elements set to one.
TypeRef tVector = DsHandler::newIntVector(10,1);

// set the first element to 99
TIntVector.setElement(0,99);

// read the value of the first element.
int tValue = tIntVector.getIntElement(0);
```
To call operator [] on a vector it is necessary to have an object of \texttt{IntVector} or \texttt{FloatVector}, due to the operator [] that is not defined in the base class \texttt{Type}. For this it is necessary to use \texttt{static_cast} operator.

**Example 2:**

```cpp
// create a generic vector
TypeRef tVector = DsHandler::newValue("Root.Vector");

// set the default value to a defined data structure
tVector.setDefault(DsHandler::findClass("Root.NetworkProtocol.TCPProtocol"));

// change the vector length to 25.
// The vector is initialized with default elements.
tVector.changeLength(25);
```

### 15.2.2.3 Enumeration Data Structure Operations

Enumeration class (\texttt{EnumType}) is designed to satisfy the requirements of C++ style enumerations. In this way an Enumeration object can be used to describe an enumeration, i.e. its elements, as well as holding its value, which must be in the range described by its elements.

Following methods are provided for elements management:

- `void addElement (const Element* e);`
  - Add a new element in the enumeration list. This element should have the index set. Take care that the Element object must be created with new.

- `void addNewValue (const String& pValue);`
  - Add a new value in the enumeration list this function set the element index to the last element’s index incremented by one.

- `int deleteElement (const String& pValue);`
- `int deleteIndex (int pIndex);`
  - All these methods can be used to remove an element from an enumeration. The difference between them is the way the element is searched for: by its value or by its index.

- `const void setValue (const String& pValue);`
- `const void setValue (int pValue);`
- `const void setValue (Element* pValue);`
  - You can set a value of an enumeration using one of these three methods. In case the specified element is not found in the enumeration this method will throw an Exception, so it is better to place this methods in a try...catch block.

\texttt{EnumType} provides various methods to get the value inside. You can get the value as an element, as a string, or you can get the index of the value.
const Element* getValueElement () const;

You can use this method to get the value element of an enumeration. This is a pointer to the value member, so don’t try to change or to delete this pointer.

String value () const;
operator const char* ();
const char* operator () (void);

This methods and operators return the actual value of an enumeration as a string.

int indexOfMyValue () const;

You can use this method to get the actual value as index.

With the following methods you can get information on the possible elements of an enumeration.

String getValueAtIndex (int pIndex);

Returns the value of the element with the specified index. The value is returned as a string.

Element* getElementAtIndex (int pIndex);

Use this method to get a pointer to the specified element. Changes on this pointer affects the element inside the enumeration.

int getIndexForValue (const String& pValue);

This method returns the index of an element specified by its value.

15.2.2.4 Operations for Composite Data Structures

Composite data structures use a handler class called DataStructMember for member management. With composite data structures you can use a reference to such a handler object or use methods directly on the composite object. DataStructure class provides methods to set or read member values.

void setField (const String& name, int value);
void setField (const String& name, float value);
void setField (const String& name, const String& value);
void setField (const String& name, const Type& value);

You can set a value for a field using these methods, where name is the name of the member.

void setField (unsigned int index, int value);
void setField (unsigned int index, float value);
void setField (unsigned int index, const String& value);
void setField (unsigned int index, const Type& value);
15.2 Using Data Structures

If you know the index you can call these methods with index and value making the simulation much faster.

If the member with the given name or index is not found a `DataTypeException` is thrown.

Example:

```c++
// creates a new data structure
TypeRef tData("Root.NetworkProtocol.TCPProtocol");
// sets the field "Name" to value XYZ
    tData.setField("Name","XYZ");
// set the "SourceID" field to index "1"
// using the string representation
    tData.setField(1,"{192,168,2,0}");
// create a new IPAddress
TypeRef tDestination("Root.Address.IPAddress");
// sets "Byte1" to 255
    tDestination.setField("Byte1",255);
// outputs the data structure to the "Output" port
    Output.put(arrivalTime) << tData;
```

For reading member values use one of the following methods:

```c++
Type& getField (const String& name);
Type& getField (unsigned int pIndex);
const Type& getField (const String& name) const;
const Type& getField (unsigned int pIndex) const;
```

When the method is non const it returns a copy of the data inside, otherwise it returns a reference to the data inside. When the field is not found, the methods throw a `DataTypeException`.

```c++
int getIndexForMember (const String& pName) const;
```

Returns the index for a specific member. Use this method in `setup` primitive method to save the index (see example).

Example (this is part of a primitive code):

```c++
protected
{
    int mIndex; // primitive member to save the index
}

setup
{
    const TypeClass& tTCP = DsHandler::findClass("Root.NetworkProtocol.TCPProtocol");
    mIndex = tTCP.getIndexForMember("SourcePort");
```
In the new mechanism the field handler, DataStructMember, belongs only to the data structure descriptors, the class part of data structures. It does not hold any longer a data structure value therefore old methods like:

```cpp
Type& data ();
Type* getData ();
const Type& data () const;
const Type* getData () const;
void writeData (const Type* data);
```

are no longer available. If you used them, you have to replace them with calls directly on the DataStructure object. Return methods like `getData()` and `data()` can be replaced with `getField(...)` methods from DataStructure class, and `writeData(...)` with `setField(...)`. You can iterate over members of a data structure using the iterator class

```cpp
DSMemberIter (DataStructureClass& ds)
```

and over the values inside a composite data structure with

```cpp
DSFieldIter (DataStructure& ds)
```

`operator++` returns the current member and moves the cursor to the next one.

If you use the old iterator `DSMemberIter(DataStructure&)` you’ll have to replace it with one of these two iterators.

### 15.3 When to Clone/Release Data Structures.

MLDesigner 2.3 introduces a new data structure mechanism and with it a new type of classes that handle the copying and releasing of data structure value objects. The base class is called `TypeRef` and from this are some type specific classes derivated and they have distinctive R as last letter. To avoid any problems that might appear (like memory leak or unexpected behavior) is better to use only this kind of classes when programming using data structures. This mechanism is based on the object’s life time. Every time a reference object is created it increase the reference
15.4 When is a data structure released?

counter of the value object. When the object is destroyed the destructor is called and with that the reference counter is decreased. When this becomes zero the value data structure is put in a pool of free elements. As a result when you use objects of this class types you have to take care that the life time of this will end after exiting the method that defines it. Do not initialize this classes dynamic unless you want to handle the object life time manually.

Next paragraph contains a description of the cloning and releasing mechanism in case you want to make an idea about it. It is important to know exactly when a data structure must be cloned and whether a cloned value data structure must be released, by calling the die() method, or deleted. This is critical in order to avoid memory leak.

Data structure clones are created when:

- A data structure is returned by the method DsHandler::newValue(...). Call this method every time you want to instance a new data structure for direct usage as a value in primitives.
- Methods that return data structure values, return a copy of the object inside when the method is defined non const, and a const reference to the inside value when the method is defined const.
- When you call methods that take as parameter a const reference to a data structure the value is cloned inside the method.

Example:

```cpp
// data structure is instantiated and sent on the "Output" port
TypeRef tData("Root.Address.IPAddress");
Output.put(arrivalTime) << tData;
```

Exception to this rule are old methods, now obsolete:

```cpp
getTypeForMember(...) - in DataStructure class;
```

Const references returned by some methods of data structures are references to the values inside the object.

Example:

```cpp
// the fastest way to read a memory when no changes have occurred.
const Memory tMemory;
const ConstTypeRef tData = tMemory;
```

15.4 When is a data structure released?

- You can release a data structure by calling the die() method. This method places the data structure in a pool of free data structure of the same type.
- DataStruct particles release any data structure they reference once it is no longer needed. If you get a data structure from an input port or you create one and place it on an output port, you don’t have to worry about whether the data structure is released.
• Every clone data structure value must be released using the die() method. The exception is when you place it on an output port or when you use a TypeRef class.

Example 1:

```go
{  
    TypeRef tData = MyMemory.readData(); // clone returned  
    // or  
    TypeRef tData = MyMemory;  
    ....  
} // life time of tData ends - data structure is no longer needed and it is released.
```

Example 2:

```go
TypeRef tData = MyMemory.readData(); // clone returned  
....  
Output.put(arrivalTime) << tData;  
// data structure is placed on the output port.
```

Errors in using clone() and die() methods in a simulation can lead to unexpected behavior or memory leak, so try to analyze and use the data structure mechanism correctly.

15.5 Compatibility Problems

In designing the new data structure mechanism we tried to keep unchanged as much as possible the actual interface. Unfortunately this was not fully possible and some methods had to be changed or removed in the new interface. In case you run into compatibility problems - compile errors in old primitives - you can easily fix it following the indications in the next paragraph.

15.6 Known problems

const Type* DsHandler::findStructure(...) generates memory leak.

In the new mechanism, DsHandler contains only descriptors of data structures, that are Type-Class objects or are derived from this class. In this case this method makes no more sense, and has been replaced with findClass(...) or findClassPointer(...). The old method now creates a value object (a clone) when the specified data structure is found and returns it as const pointer. This generates memory leak since this pointer cannot be changed any more.

Data structure access methods from DataStructMember were removed.

The new DataStructMember does not hold any references to value objects. It is used to handle the description of composite data structures and exists only in the class objects. In case you use these methods:
you will have to replace them to calls directly on the DataStructure object.

Example:

// before:
Type* tTCP = DsHandler::makeNewStructure("Root.Protocol.
TCPProtocol");
DataStructMember* tMember = tTCP->getMember("SourcePort");
Type* tData = tMember->getData();
// do something with data
tMember->writeData(tData);
tData->die(); // data structure released, no longer needed.
tTCP->die(); // data structure released, no longer needed.

// replaced with:
TypeRef tTCP("Root.Protocol.TCPProtocol");
TypeRef tData = tTCP.getField("SourcePort");
// do something with data
tTCP.setField("SourcePort",tData);

DataStructMember iterators are modified.

Since a DataStructure object doesn’t hold references to DataStructMember objects any longer, it is not possible to use a DSMemberIter iterator over it. Depending on what you want to do you can use DSMemberIter (DataStructureClass&) or DSFieldIter (DataStructure&). If you need to access DataStructMember objects, then use DSMemberIter over a DataStructClass object that returns DataStructMember objects. If you need to access the data under a member in a composite data structure value, then use DSFieldIter over a DataStructure object and you will get all the fields in this composite.

Example:

// you used a DSMemberIter in this way:
Type* tTCP = DsHandler::makeNewStructure("Root.Protocol.TCPProtocol");
DSMemberIter next(*tTCP);
while (DataStructMember* tMember = next++)
{
    Type* tData = tMember->getData();
    // do something with data
    tData->die(); // data structure released, no longer needed.
}
tTCP->die(); // data structure released, no longer needed.
// replace with:
    TypeRef tTCP("Root.Protocol.TCPProtocol");
    DSFieldIter next(tTCP);
    while (Type* tData = next++)
    {
        // do something with data
        tData->die();
    }

**Operator [] in Vector data structure might generate memory leak.**

In order to have a general mechanism for cloning and releasing value data structures the data structure returned by this operator changed from a reference to the element to a copy of the element. Example:

// if you used an expression like:
    tVector[i] = tElement; // where tVector is a Vector
// replace with:
    tVector.setElement(i,tElement);
## 15.6 Known problems

<table>
<thead>
<tr>
<th>primitive</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getName()</td>
<td>returns the name of a data structure</td>
</tr>
<tr>
<td>String getFullName()</td>
<td>returns the full name of a data structure</td>
</tr>
<tr>
<td>String getUniqueName()</td>
<td>returns the unique name of a data structure</td>
</tr>
<tr>
<td>bool isA(const Type&amp; pType) const;</td>
<td>Tests if the type has the same name as the parameter. If you want to compare two types, call one of this methods</td>
</tr>
<tr>
<td>bool isParentOf(const Type* pChild) const;</td>
<td>Tests if pChild is a data structure derived from this data structure</td>
</tr>
<tr>
<td>bool isChildOf(const Type* pParent) const;</td>
<td>Tests if pParent is a parent for this data structure</td>
</tr>
<tr>
<td>String toString() const;</td>
<td>Returns the string representation of a data structure</td>
</tr>
<tr>
<td>void createFromString(const char <em>pString) throw (DataTypeSyntaxException</em>);</td>
<td>This method sets all the data structure’s values conforming to the string representation. When you call this function you should be careful to place it in a try catch block. The exception is thrown when a syntax error occur in the parameter string.</td>
</tr>
<tr>
<td>const Type&amp; operator = (const Type&amp;);</td>
<td>Sets all values to the values provided by the parameter. If the parameter is not of the same type, a DataTypeException is thrown.</td>
</tr>
<tr>
<td>bool operator == (const Type&amp;);</td>
<td>Tests if two data structures are equal. First the type is tested, and, if they are the same type, values are tested.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>primitive</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 15.1: Data structure Class Methods
Chapter 16

Using Tcl/Tk in Primitives

Tcl (Tool Command Language) pronounced “tickle” was created in 1988 by John Ousterhout, while he was a professor at UC Berkeley. Since that time its use has spread through a wide variety of software industries for mission-critical integration tasks.

Tk is a graphical user interface toolkit that makes it possible to create powerful GUIs incredibly quickly. Both Tcl and Tk have been integrated into MLDesigner. Parts of the graphical user interface and all of the textual interpreter ptcl are designed using them. Several of the primitives in the standard primitive library also use Tcl/Tk. This chapter explains how to use the most basic of these primitives, TclScript, as well as how to design such primitives from scratch. It is possible to define very sophisticated, totally customized user interfaces using this mechanism.

In this chapter, it is assumed you are familiar with the Tcl language. Up-to-date documentation and software releases are available on the Tcl Developer Xchange web page at www.tcl.tk. There is also a newsgroup called comp.lang.tcl.

The principal use of Tcl/Tk in MLDesigner is to customize the user interface.

16.1 Writing Tcl/Tk Scripts for the TclScript Primitive

Several of the domains in MLDesigner have a primitive called TclScript. This primitive provides the quickest and easiest path to a customized user interface.

The TclScript primitive has an unspecified number of inputs and outputs as indicated by the double arrows at its input and output ports. The TclScript primitive has one settable parameter tcl_file. This parameter defines a string that contains the full path name of a file containing a Tcl script. The Tcl script file specifies initialization commands, for example to open new windows on the screen, and may optionally define a procedure to be invoked by the primitive every time it runs. We begin with two examples that illustrate most of the key techniques needed to use this primitive.

Example 1: This example demonstrates the use of the TclScript primitive for creating customized GUI elements using Tcl files. The system can be found in MLD Examples/Tutorials/TclSystem. We must create a system containing the TclScript primitive and the TkShowValues found in MLD Libraries/SDF Domain/SDF Demo/Tcl/Tk (see fig. 16.2).

This is done as follows:
Figure 16.1: Examples of *TclScript* icons

Figure 16.2: System model of *TclScript* demo
16.1 Writing Tcl/Tk Scripts for the TclScript Primitive

- Click the New Model icon and create a new Library called TclScript.
- Click the New Model icon and select System from the Type of model menu.
- Give the system a name such as MyTclSystem. Select the newly created library from the Select Library dialog.
- Open the library MLD Libraries/SDF Domain/TclTk and click and drag the primitive TclScript into the Model Editor Window MyTclSystem. From the Select Special Primitive dialog choose TclScript.input=0.output=1 and click OK.
- Click and drag the TkShowValues primitive into the Model Editor Window.
- Connect the ports of the two model instances. (see fig. 16.2).

You must now create a (.tcl) file containing the following script:

```tcl
set s $ptkControlPanel.middle.button_$starID
if {! [winfo exists $s]} {
    button $s -text "PUSH ME"
    pack append $ptkControlPanel.middle $s {top}
    bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
    bind $s <ButtonRelease-1> "setOutputs_$starID 0.0"
    setOutputs_$starID 0.0
}
unset s
```

Save this file in your TclScript library with the name TclScript.tcl.

**NOTE:** Since in Ptolemy vocabulary a primitive is called a star, the variable starID always means the identifier of the primitive.

You must now alter the reference to the Tcl script as follows:

- select the TclScript primitive in the design area;
- in the Property Editor, set the value of parameter tcl file to $MLD_USER/TclScript/-TclScript.tcl (see fig. 16.3).

Change the RunLength value to -1. This makes the simulation run endlessly or until you decide to terminate it. You can now run the simulation. Click the Switch to Simulation Mode icon and click Go. This script creates a push button in the tclRunControl panel. The primitive outputs the value 1.0 when the button labeled Push Me is clicked and held and 0.0 when it is released (see fig. 16.4).

### 16.1.1 Create a New TclScript Special Primitive

If you want to create a new special primitive because one with the correct amount of input or output ports does not exist, you need to first create a new primitive derived from TclScript and then create a special primitive with the correct amount of ports.

**NOTE:** The Derived primitive has its own empty go method which overwrites the go method of the parent. You must either delete the go entry in the derived primitives source code or write new code for the method.
16 Using Tcl/Tk in Primitives

Figure 16.3: Full path where TclScript.tcl is saved

Figure 16.4: Additional controls in tclRunControl panel
16.1 Writing Tcl/Tk Scripts for the TclScript Primitive

16.1.2 The Tcl Script Explained

The Tcl script is explained here:

```tcl
set s $ptkControlPanel.middle.button_$starID
```

This defines a Tcl variable `s` whose value is the name of the window to be used for the button. The first part of the name, `$ptkControlPanel`, is a global variable giving the name of the control panel window itself. This global variable has been set by MLDesigner and can be used by any Tcl script. The second part, `.middle`, specifies that the button should appear in the sub-window named `middle` of the control panel. The control panel, by default, has empty sub-windows named `high`, `middle`, and `low`. The last part, `.button_$starID`, gives a unique name to the button itself. The Tcl variable `starID` has been set by the `TclScript` primitive to a name that is guaranteed to be unique for each instance of the primitive. Using a unique name for the button permits multiple instances of the primitive in a model to create separate buttons in the control window without conflict.

```tcl
if {! [winfo exists $s]} {
    ...
}
```

This conditionally checks to see whether or not the button already exists. If, for example, the system is being run a second time, then there is no need to create the button a second time. In fact, an attempt to do so will generate an error message. If the button does not already exist, it is created by the following lines:

```tcl
button $s -text "PUSH ME"
pack append $ptkControlPanel.middle $s {top}
```

The first of these defines the button, and the second packs it into the control panel, refer to the Tk documentation. The following Tcl statement binds a particular command to a mouse action, thus defining the response when the button is pushed.

```tcl
bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
```

When button number 1 of the mouse is pressed, the Tcl interpreter invokes a procedure named `setOutputs_$starID` with a single argument, 1.0 (passed as a string). This procedure has been defined by the `TclScript` primitive. It sets the value(s) of the outputs of the primitive. In this case, there is only one output, so there is only one argument. The next statement defines the action when the button is released:

```tcl
bind $s <ButtonRelease-1> "setOutputs_$starID 0.0"
```

The next statement initializes the output of the primitive to value 0.0.

```tcl
setOutputs_$starID 0.0
```
The last command unsets the variable \( s \), since it is no longer needed.

As illustrated in the previous example, a number of procedures and global variables will have been defined for use by the Tcl script by the time it is sourced. These enable the script to modify the control panel, define unique window names, and set initial output values for the primitive. Much of the complexity in the above example is due to the need to use unique names for each primitive instance that sources this script. In the above example, the Tcl procedure for setting the output values has a name unique to this primitive. Moreover, the name of the button in the control panel has to be unique to handle the case when more than one \texttt{TclScript} primitive sources the same Tcl script. These unique names are constructed using a unique string defined by the primitive prior to sourcing the script. That string is made available to the Tcl script in the form of a global Tcl variable \texttt{starID}. The procedure used by the Tcl script to set output values is called \texttt{setOutputs}_{\texttt{starID}}. This procedure takes as many arguments as there are output ports. The argument list should contain a floating-point value for each output of the primitive.

In the above example, Tcl code is executed when the Tcl script is sourced. This occurs during the setup phase of the execution of the primitive. After the setup phase, no Tcl code will be executed unless the user pushes the "PUSH ME" button. The command

\[
\text{bind } \$s \text{ <ButtonPress-1> } \"\text{setOutputs}_{\text{starID}} 1.0\"
\]

defines a Tcl command to be executed asynchronously. Notice that the command is enclosed in quotation marks, not braces. Tcl aficionados will recognize that this is necessary to ensure that the \texttt{starID} variable is evaluated when the command binding occurs (when the script is sourced), rather than when the command is executed. There is no guarantee that the variable will be set when the command is executed.

In the above example, no Tcl code is executed when the primitive fires. The following example shows how to define Tcl code to be executed each time the primitive fires, and also how to read the inputs of the primitive from Tcl.

Example 2: Consider the following design in the SDF domain

Suppose the Tcl script for the \texttt{TclScript} primitive reads:

```
proc goTcl_{starID} {starID}
{
```
16.1 Writing Tcl/Tk Scripts for the TclScript Primitive

```tcl
set inputVals [grabInputs_$starID]
set xin [lindex $inputVals 0]
set yin [lindex $inputVals 1]
setOutputs_$starID [expr $xin+$yin]
}
```

Unlike the previous example, this script does not define any code that runs when the script is sourced, during the setup phase of execution of the primitive. Instead, it simply defines a procedure with a name unique to the instance of the primitive. This procedure reads two input values, adds them, and writes the result to the output. Although this would be a very costly way to accomplish addition in MLDesigner, this example nonetheless illustrates an important point. If a Tcl script sourced by a TclScript primitive defines a procedure called goTcl_$starID, then that procedure will be invoked every time the primitive fires. The single argument passed to the procedure when it is called is the starID. In this example, the procedure uses grabInputs_$starID, defined by the TclScript primitive, to read the inputs. The current input values are returned by this procedure as a list, so the Tcl command lindex is used to index into the list. The final line adds the two inputs and sends the result to the output.

As shown in the previous example, if the Tcl script defines the optional Tcl procedure goTcl_$starID, then that procedure will be invoked every time the primitive fires. It takes one argument (the starID) and returns nothing. This procedure, therefore, allows for synchronous communication between the MLDesigner simulation and the Tcl code (it is synchronized to the firing of the primitive). If no goTcl_$starID procedure is defined, then communication is asynchronous (Tcl commands are invoked at arbitrary times, as specified when the script is read). For asynchronous operation, typically X events are bound to Tcl/Tk commands that read or write data to the primitive.

The inputs to the primitive can be of any type. The print() method of the particle is used to construct a string passed to Tcl. Although it is not illustrated in the above examples, asynchronous reads of the primitive inputs are also allowed.

Below is a summary of the Tcl procedures used when executing a TclScript primitive:

- **grabInputs_$starID**
  - A procedure that returns the current values of the inputs of the primitive corresponding to the given starID. This procedure is defined by the TclScript primitive if and only if the instance of the primitive has at least one input port.

- **setOutputs_$starID**
  - A procedure that takes one argument for each output of the TclScript primitive. The value becomes the new output value for the primitive. This procedure is defined by the TclScript primitive if and only if the instance of the primitive has at least one output port.

- **goTcl_$starID**
  - If this procedure is defined in the Tcl script associated with an instance of the TclScript primitive, then it will be invoked every time the primitive fires.
wrapupTcl_{\$starID}

If this procedure is defined in the Tcl script associated with an instance of the TclScript primitive, then it will be invoked every time the wrapup method of the primitive is invoked. In other words, it will be invoked when a simulation stops.

destructorTcl_{\$starID}

If this procedure is defined in the Tcl script associated with an instance of the TclScript primitive, then it will be invoked when the destructor for the primitive is invoked. This can be used to destroy windows or to unset variables that are no longer needed.

In addition to the $starID global variable, the TclScript primitive makes other information available to the Tcl script. The mechanism used is to define an array with a name equal to the value of the $starID variable. Tcl arrays are indexed by strings. Thus, not only is $starID a global variable, but so is $\$starID$. The value of the former is a unique string, while the value of the latter is an array. One of the entries in this array gives the number of inputs that are connected to the primitive. The value of the expression \[\text{set \$\{\$starID\}(numInputs)}\] is an integer giving the number of inputs. The Tcl command set, when given only one argument, returns the value of the variable whose name is given by that argument. The array entries are summarized below.

$\$starID

This evaluates to a string that is different for every instance of the TclScript primitive. The $starID global variable is set by the TclScript primitive.

[set \$\{\$starID\}(numInputs)]

This evaluates to the number of inputs that are connected to the primitive.

[set \$\{\$starID\}(numOutputs)]

This evaluates to the number of outputs that are connected to the primitive.

[set \$\{\$starID\}(tcl_file)]

This evaluates to the name of the file containing the Tcl script associated with the primitive.

[set \$\{\$starID\}(fullName)]

This evaluates to the full name of the primitive (which is of the form system.module.module.primitive).

### 16.2 Tcl Utilities Available to the Programmer

A number of Tcl global variables and procedures that will be useful to the Tcl programmer have been incorporated into MLDesigner. Any of these can be used in any Tcl script associated with an instance of the TclScript primitive. For instance, in example 1 on page 16-1, the global variable ptkControlPanel specifies the control panel that is used to run the system. Below is a list of the useful global variables that have been set by the MLDesigner graphical interface when the Tcl script is sourced or when the goTcl_{\$starID} procedure is invoked.
$ptkControlPanel
A string giving the name of the control panel window associated with a given run.
This variable is set by MLDesigner.

$ptkControlPanel.high
The uppermost panel in the control panel that is intended for user-defined entries.

$ptkControlPanel.middle
The middle panel in the control panel that is intended for user-defined entries.

$ptkControlPanel.low
The lowest panel in the control panel that is intended for user-defined entries.

In addition to these global variables, a number of procedures have been supplied. Using these procedures can ensure a consistent look-and-feel across a variety of MLDesigner applications. The complete set of procedures can be found in $MLD/lib/tcl. Only the most useful commands are listed here. Note also that the entire set of commands defined in the Tcl-based textual interpreter for MLDesigner, P Tcl, are also available. So for example, the command curuniverse will return the name of the current system (see ch. 9).

NOTE: In Ptolemy vocabulary a system is called universe.

ptkExpandEnvVar
Procedure to expand a string that begins with an environment variable reference. For example, ptkExpandEnvVar $MLD/src will return something like /opt/mld/src (depending on the installation of MLDesigner).

ptkImportantMessage
Procedure to pop up a message window and grab the focus. The process is suspended until the message is dismissed.
Arguments:
  win window name to use for the message
  text text to display in the pop-up window

ptkMakeButton
Procedure to make a pushbutton in a window. A callback procedure must be defined by the programmer. It will be called whenever the user pushes the button, and takes no arguments.
Arguments:
  win name of window to contain the button
  name name to use for the button itself
  desc description to be put into the display
  callback name of callback procedure to register changes

ptkMakeEntry
Procedure to make a text entry box in a window. A callback procedure must be defined by the programmer. It will be called whenever the user changes the value in the entry box and types <Return>. Its single argument will be the new value of the entry.

Arguments:
- `win`: name of window to contain the entry box
- `name`: name to use for the entry box itself
- `desc`: description to be put into the display
- `default`: the initial value of the entry
- `callback`: name of callback procedure to register changes

**ptkMakeMeter**

Procedure to make a bar-type meter in a window.

Arguments:
- `win`: name of window to contain the entry box
- `name`: name to use for the entry box itself
- `desc`: description to be put into the display
- `low`: the value of the low end of the scale
- `high`: the value of the high end of the scale

**ptkSetMeter**

Procedure to set the value of a bar-type meter created with ptkMakeMeter.

Arguments:
- `win`: name of window to contain the entry box
- `name`: name to use for the entry box itself
- `value`: the new value to display in the meter

**ptkMakeScale**

Procedure to make a sliding scale. All scales in the control panel range from 0 to 100. A callback procedure must be defined by the programmer. It will be called whenever the user moves the control on the scale. Its single argument will be the new position of the control, between 0 and 100.

Arguments:
- `win`: name of window to contain the scale
- `name`: name to use for the scale itself
- `desc`: description to be put into the display
- `position`: initial integer position between 0 and 100
- `callback`: name of callback procedure to register changes

**NOTE:** A widget is created with name `$win.$name.value` that should be used by the programmer to display the current value of the slider. Thus, the callback procedure should contain a command like: `$win.$name.value configure -text $new_value`
16.2 Tcl Utilities Available to the Programmer

to display the new value after the slider has been moved. This is not performed automatically because the fixed range from 0 to 100 may be correct from the user’s perspective. So, for example, if the programmer divides the scale value by 100 before displaying it, it will appear to the user as if the scale ranges from 0.0 to 1.0. It is also possible to control the position of the slider from Tcl (overriding the user actions) using a command like $win.$name.scale set $position, where position is an integer-valued variable in the range of 0 to 100.

Example 3: The following Tcl script can be used with the TclScript primitive in the system configuration given in example 1 on page 16-1

```tcl
ptkMakeMeter $ptkControlPanel.high meter_$starID "meter tracking scale" 0 100
proc scale_update_$starID {new_value} {
    ptkSetMeter $ptkControlPanel.high meter_$starID $new_value
    $ptkControlPanel.high.scale_$starID.value configure -text "$new_value"
}
ptkMakeScale $ptkControlPanel.high scale_$starID "my scale" 50 scale_update_$starID
ptkMakeButton $ptkControlPanel.middle button_$starID "my button" button_update
proc button_update {} {ptkImportantMessage .msg "Hello"}
ptkMakeEntry $ptkControlPanel.low entry_$starID "my entry" 10 entry_update_$starID
proc entry_update_$starID {new_value} {setOutputs_$starID $new_value}
```

It will create the rather raw control panel shown in fig. 16.6. The commands are explained individually below.

![Simulated control panel](image)

Figure 16.6: Simulation control window for the TclScript demo

The first two lines of the script
ptkMakeMeter $ptkControlPanel.high meter_$starID "meter tracking scale" 0 100

call the method ptkMakeMeter to create a meter display with the label ”meter tracking scale” in the upper part of the control panel with range from 0 to 100. The next script lines

proc scale_update_$starID {new_value} {
  "ptkSetMeter $ptkControlPanel.high meter_$starID \$new_value
  $ptkControlPanel.high.scale_$starID.value \configure -text \$new_value"
}

define the callback function to be used for the slider (scale) shown below the meter. The callback function sets the meter and updates the numeric display to the left of the slider. Notice that the body of the procedure is enclosed in quotation marks rather than the usual braces. This ensures that the variables ptkControlPanel and starID will be evaluated at the time the procedure is defined, rather than at the time it is invoked. To make sure that new_value is not evaluated until the procedure is invoked, a preceding backslash is being used, as in \$new_value. The ptkControlPanel and starID values could have been alternatively passed as arguments.

ptkMakeScale $ptkControlPanel.high scale_$starID "my scale" 50 scale_update_$starID

This creates the slider itself, and sets its initial value to 50, half of full scale.

ptkMakeButton $ptkControlPanel.middle button_$starID "my button" button_update

This creates a button labeled ”my button”.

proc button_update {} {ptkImportantMessage .msg "Hello"}

This defines the callback function connected with the button. This callback function opens a new window with the message ”Hello”, and grabs the focus. the programmer must dismiss the new window before continuing.

proc button_update {} {ptkImportantMessage .msg "Hello"
  ptkMakeEntry $ptkControlPanel.low entry_$starID "my entry" 10 entry_update_$starID
}

This creates the entry box with initial value ”10”.

proc entry_update_$starID {new_value} {
  "setOutputs_$starID \$new_value"
}

This defines the callback function associated with the entry box. Again notice that the procedure body is enclosed in quotation marks.
16.3 Creating Primitives Derived from *TclScript*

A large number of useful primitives can be derived from the *TclScript* primitive. The *TkShowValues* primitive used in example 1 on page 16-1 is such a primitive. This primitive takes inputs of any type and displays their value in a window that is optionally located in the control panel. It has three settable parameters:

- **label**
  a string parameter giving a label to identify the display

- **put_in_control_panel**
  a boolean parameter that specifies whether the display should be put in the control panel or in its own window

- **wait_between_outputs**
  a boolean parameter that specifies whether the execution of the system should pause each time a new value is displayed. If it does, then a mouse click in the display restarts the system.

Conspicuously absent is the *tcl_file* parameter of the *TclScript* primitive from which this is derived. The file is hard-wired into the definition of the primitive by the following C++ statement included in the begin method.

```
tcl_file="$MLD/MLD_Libraries/SDF/TclTk/tkShowValues.tcl";
```

The parameter is hidden from the user of the primitive by the following statement included in the constructor.

```
tcl_file.clearAttributes(A_SETTABLE);
```

Thus, the user only sees the parameters that are defined in the derived primitive. This is a key part of customizing the primitive.

A second issue is that of communicating the new parameter values to the Tcl script. For example, the Tcl script will need to know the value of the *label* parameter in order to create the label for the display. The *TclScript* primitive automatically makes all the parameters of any derived primitive available as array entries in the global array whose name is given by the global variable *starID*. To read the value of the *label* parameter in the Tcl script, use the expression `[set $\{starID\}(label)]`. The confusing syntax is required to ensure that Tcl uses the value of *starID* as the name of the array. The string *label* is just the index into the array. The *set* command in Tcl, when given only one argument, returns the value of the variable whose name is given by the argument.

Some programmers may prefer an alternative way to refer to parameters that is slightly more readable. The Tcl statement

```
upvar #0 $starID params
```

allows subsequent statement to refer to parameters simply as `$param(param_name)`. The *upvar* command with argument #0 declares the local variable *params* equivalent to the global variable whose name is given by the value of *starID*.

Many more examples can be found in `$MLD/MLD_Libraries/SDF/TclTk/primitives`. 
16.4 Writing Tcl Primitives for DE Domain

In the discrete-event (DE) domain, primitives are fired in chronological order according to the time stamps of the new data that has arrived at their input ports. The Tcl interface class TclStarIfc, which was originally written with the SDF domain in mind, works well for some types of DE primitives. Specifically, any primitive with an input in the DE domain can use this class effectively. Consequently, a basic Tcl/Tk primitive, TclScript, has been written for the DE domain. The TclScript primitive can have any number of input or output ports.
Chapter 17

Domain Related Issues

17.1 SDF Domain

Synchronous Data Flow (SDF) is a statically scheduled data flow domain in MLDesigner. "Statistically scheduled” means that the firing order of the primitives is determined once, during the start-up phase. The firing order will be strictly periodic. The SDF domain in MLDesigner is one of the most mature and common, with a large library of primitives and demo programs. It is a simulation domain, but the model of computation is the same as that used in most of the code generation domains. A number of different schedulers, including parallelizing schedulers, have been developed for this model.

It is assumed that you are familiar with the SDF model of computation. Refer to the ch. 18 for further informations. We also assume you are familiar with writing primitives for the SDF domain (see ch. 13.) Since most of the examples given in that chapter are from the SDF domain, there is only a little more information to add here.

Setting SDF Porthole Parameters

All primitives in the SDF domain must follow the basic SDF principle. The number of particles consumed or produced on any porthole does not change within the simulation runs. These numbers are given for each porthole as part of the primitive definition. Most primitives consume just one particle on each input and produce just one particle on each output. In these cases, no special action is required, since the porthole SDF parameters will be set to one by default. However, if the numbers differ from one, the primitive definition must reflect this. For example, the \textit{FFTCx} primitive has a \texttt{size} parameter that specifies how many input samples are given. The value of that parameter specifies the number of samples required at the input in order for the primitive to fire. The following line in the setup method of the primitive is used to make this information available to the scheduler.

\begin{verbatim}
        input.setSDFParams (int(size), int(size)-1);
\end{verbatim}

The name of the input porthole is \texttt{input}. The first argument to \texttt{setSDFParams} specifies how many samples are consumed by the primitive when it fires. It is the same as the number of samples required in order to enable the primitive. The second argument to \texttt{setSDFParams} specifies how many past samples (before the most recent one) will be accessed by the primitive when it fires.
If the number of particles produced or consumed is independent from any parameters, then it may be declared right along with the declaration of the `input`, in the `.pl` file. For example,

```plaintext
input
{
    name { signalIn }
    type { complex }
    num { 2 }
    desc { Complex input that consumes 2 input particles. }
}
```

This declares an input that consumes two successive complex particles.

## 17.2 DDF Domain

A Dynamic Data Flow (DDF) primitive, as distinct from an SDF primitive, has at least one port-hole, either an input or an output, that receives or sends a variable number of particles. Such port-holes are called *dynamic*. Consequently, for a DDF primitive, how many particles to read or write is determined at run time, in the `go` method. Consider an example, the `LastOfN` primitive:

```plaintext
defprimitive
{
    name {LastOfN}
    domain {DDF}
    desc
    {
        Outputs the last token of N input tokens,
        where N is the value of the control input.
    }
    input
    {
        name {input}
        type {anytype}
        num {0}
    }
    input
    {
        name {control}
        type {int}
    }
    output
    {
        name {output}
        type {anytype}
    }
    private
```
The `LastOfN` primitive discards the first $N - 1$ particles from the `input` porthole and routes the last one to the `output` porthole. The value $N$ is read from the `control` input. Since the control data varies, the number of particles to read from the `input` porthole is variable, as expected for a DDF primitive. The programmer can specify that the `input` porthole is dynamic by setting the `num` field of the `input` declaration to 0 using the preprocessor format:

```
num {0}
```

The firing rule of the primitive is controlled by the `waitFor` method of the `DDFStar` class (actually, it is defined in the base class, `DynDFStar`). The `waitFor` method takes a porthole as an argument, and an optional integer as a second argument. It indicates that the primitive should fire when the amount of data specified by the integer (default is 1) is available on the specified port. In the above example, the `setup` method specifies that the primitive should first wait for a `control` input. When a `control` input arrives, the `go` method reads the control value, and uses `waitFor` to specify that the primitive should fire next when the specified number of values
have arrived at input. The private member readyToGo is used to keep track of which input the primitive is waiting for. The line

```java
for (int i = num; i > 0; i--) input.receiveData();
```

causes the appropriate number of inputs (given by num) to be consumed.

**DownCounter**

This is an example of the DDF primitive with a dynamic output porthole: the DownCounter primitive.

```java
defprimitive
{
    name { DownCounter }
    domain { DDF }
    author { Soonhoi Ha }
    copyright { Copyright (c) 1990-1996 The Regents of the University of California. All rights reserved. See the file $MLD/copyright for copyright notice, limitation of liability, and disclaimer of warranty provisions. }
    desc
    {
        Given an integer input with value N, produce a sequence of output integers with values (N-1), (N-2), ... 1, 0.
    }
    input
    {
        name { input }
        type { int }
    }
    output
    {
        name { output }
        type { int }
        num {0}
    }
    location { DDF library }
    go
    {
        // get input token from Geodesic
```
input.receiveData();

// generates output
// the output only has a buffer of length one, so
// we store one data token in the buffer and send it
int in = int(input%0);
for (int i = in - 1; i >= 0; i--)
{
  output%0 << i;
  output.sendData();
}
}
}

The *DownCounter* primitive has a dynamic output porthole that will generate the down-counter sequence of integer data starting from the value read through the input porthole. The code in the go method is self-explanatory.

It is possible, although a bit uncommon, for a primitive to alternate between SDF-like and DDF-like behavior. To assert that its next firing should be under SDF rules, the primitive calls `clearWaitPort()`.

The following example shows a primitive that uses the same input for control and data. An integer input specifies the number of particles that will be consumed on the next firing. After these particles have been consumed, the primitive reverts to SDF behavior to collect the next control input.

In the following example, `readyToGo` and `num` are private integers.

```c
setup
{
  clearWaitPort();
  readyToGo = FALSE;
}

\[\text{go}\]
{
  int i;
  if (!readyToGo)
  {
    // get input token from Geodesic
    input.receiveData();
    num = int(input%0);
    waitFor(input, num);
    readyToGo = TRUE;
  }
  else
  {
    for (i = 0; i < num; i++)
    {
      input.receiveData();
      output%0 << int(input%0);
    }
  }
}
```
Because of the `clearWaitPort()` in the `setup` method, the primitive begins as an SDF primitive. It consumes one data, stores its value in `num`, and issues a `waitFor` command. This changes its behavior to DDF and specifies the number of input tokens that are required. On the next firing, it will read `num` input tokens and copy them to the `output`, and then it will revert to SDF behavior.

### 17.3 BDF Domain

Boolean-controlled Data Flow (BDF) primitives are written in almost exactly the same way as SDF primitives. When the `go` method of the primitive is executed, it is guaranteed that all required input data are present, and after execution, any particles generated by the primitive are correctly sent off to their destinations. The only addition the primitive writer has to know is how to specify that a port’s condition depends on other ports. This is accomplished with a method of the class `BDFPortHole` called `setBDFParams`.

The `setBDFParams` method takes four arguments. The first argument is the number of particles transferred by the port when the port is enabled. Note that unconditional ports are always enabled. The second argument is either a pointer or a reference to another `BDFPortHole`, which is called the associated port. The function has two overloaded forms, which is why the argument may be specified either as a pointer or as a reference. The third argument is a flag specifying the relation between the porthole the method is called on and the associated port:

- **BDF_NONE**
  - This flag indicates no relation at all.
- **BDF_TRUE**
  - This flag indicates that data are transferred by the port only when the conditional port has a `TRUE` particle.
- **BDF_FALSE**
  - This flag indicates that data are transferred by the port only when the conditional port has a `FALSE` particle.
- **BDFSAME**
  - This flag indicates that the stream transferred by the associated port is the same as the stream transferred by this port. This relation is specified for the BDF Fork actor and aids the operation of the clustering algorithm.
- **BDF_COMPLEMENT**
  - This flag indicates that the stream transferred by the associated port is the logical complement of the stream transferred by this port. This relation is specified for the BDF Not actor and aids the operation of the clustering algorithm.

The fourth argument for `setBDFParams` is the maximum delay, that is, the largest value that the primitive may specify as an argument to the `%` operator on that porthole. The default value is zero. This argument serves the same purpose as the second argument to `setSDFParams`.

The `setSDFParams` function may be used on BDF portholes. It does not alter the associated port or the relation type, but does alter the other two parameters of `setBDFParams`.

By default,
BDF portholes transfer one token, unconditionally. Calls to `setBDFParams` may be placed in the `setup` method of a primitive, or alternatively in the constructor if the call does not depend on any parameters of the primitive. Considering for example the `Switch` primitive. This primitive’s functionality is as follows: on each execution, it reads a particle from its `control` input port. If the value is `TRUE`, it reads a particle from its `trueInput` port; otherwise it reads a particle from its `falseInput` port. In any case, the particle is copied to the `output` port. Using the Ptolemy language, the `setup` method could be written:

```plaintext
setup {
    trueInput.setBDFParams(1, control, BDF_TRUE, 0);
    falseInput.setBDFParams(1, control, BDF_FALSE, 0);
}
```

and the `go` method could be written:

```plaintext
go {
    if (int(control%0))
        output%0 = trueInput%0;
    else
        output%0 = falseInput%0;
}
```

### 17.4 DE Domain

The discrete event (DE) domain in MLDesigner provides a general environment for time-oriented simulations of systems such as queuing networks, communication networks, and high-level computer architectures. In this domain, each `Particle` represents an event that corresponds to a change of the system state. The DE schedulers process events in chronological order. Since the time interval between events is generally not fixed, each particle has an associated time-stamp. Time stamps are generated by the block producing the particle, using the time stamps of the input particles and the latency of the block.

A knowledge of the DE model of computation is assumed in this section. Refer to the ch. 22 for additional information. Moreover, it is assumed the reader is familiar with ch. 13. In this section, additional information required to write primitives for the DE domain are given.

### 17.4.1 Programming Primitives in the DE Domain

A DE primitive can be viewed as an event-processor; it receives events from the outside, processes them, and generates output events after some latency. In a DE primitive, the management of the time stamps of the particles (events) is as important as the input/output mapping of particle values. Generating output values and time stamps are separable tasks. For greatest modularity, some DE primitives, so-called *delay primitives*, are dedicated to time management. Examples of such primitives are `Delay` and `Server`. These primitives, when fired, produce output events that typically
have larger time stamps than the input events. They usually do not manipulate the value of the particles in any significant way. The other primitives, so-called functional primitives, avoid time management, usually by generating output events with the same time stamp as the input events. They, however, do manipulate the value of the particles.

### Time Stamps

For managing time stamps, the DEStar class has two DE-specific members: arrivalTime and completionTime, summarized in table 17.1. Before firing a primitive, a DE scheduler sets the value of the arrivalTime member equally to the time stamp of the event triggering the current firing. When the primitive fired, before returning, it typically sets the value of the completionTime member to the value of the time stamp of the latest event produced by the primitive. The schedulers do not use the completionTime member, however, so it can actually be used in any way the primitive writer wishes. DEStar also contains a field delayType and a method setMode that are used to signal the properties of the primitive, as described below.

<table>
<thead>
<tr>
<th>Class DEStar</th>
<th>#include &quot;DEStar.h&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>double completionTime</td>
<td>store the completion time of the current execution, which in turn is equal to the next free time</td>
</tr>
<tr>
<td>double arrivalTime</td>
<td>is set by the scheduler to the time stamp of the event triggering the current firing</td>
</tr>
<tr>
<td>int delayType</td>
<td>flag to indicate whether it is a delay type primitive or not</td>
</tr>
<tr>
<td>void setMode(FiringMode m)</td>
<td>set the firing mode to PHASE or SIMPLE</td>
</tr>
</tbody>
</table>

Table 17.1: Summary of methods of class DEStar

#### 17.4.1.1 Delay primitives

Delay primitives manipulate time stamps. Two types of examples of delay primitives are pure delays, and servers. A pure-delay primitive generates an output with the same value as the input sample, but with a time stamp that is greater than that of the input sample. The difference between the input sample time stamp and the output time stamp is a fixed, user-defined value. Consider for example the DE primitive Delay:

```plaintext
defprimitive
{
  name  {Delay}
  domain {DE}
  desc  { Delays its input by a fixed amount }
  input
```
Inside the `go` method description, the `completionTime` is calculated by adding the delay to the arrival time of the current event. The last two lines will be explained in more detail below. Another type of delay primitive is a server. In a server primitive, the input event waits until a simulated resource becomes free to attend to it. An example is the DE primitive `Server`:

```java
defprimitive
{
    name {Server}
    domain {DE}
    desc
    {
        This primitive emulates a server. If an input event arrives when it is not busy, it delays it by the service time (a constant parameter). If it arrives when it is busy, it delays it by more than the service time. It must become free, and then serve the input.
    }
    input
```
This primitive uses the completionTime member to store the time at which it becomes free after processing an input. On a given firing, if the arrivalTime is greater than the completionTime, meaning that the input event has arrived when the server was free, the server then delays the input by the serviceTime only. Otherwise, the time stamp of the output event is calculated as the serviceTime plus the time at which the server becomes free, the completionTime. Both pure delays and servers are delay primitives. Hence their constructor sets the delayType member, summarized in table 17.1. This information is used by the scheduler. 

The technical meaning of the delayType flag needs some explanation. Such a primitive guarantees that it will never produce any output event with zero delay. All its output events will have time stamps larger than the time of the firing in which they are emitted. Primitives that can produce zero-delay events should leave delayType set to its default value of FALSE.

Actually, primitives often cheat a little bit on this rule. As just shown, the standard DE primitive Delay sets delayType even if the user sets the primitive’s delay parameter to zero. This causes the primitive to be treated as though it had a positive delay for the purpose of assigning firing priorities, which is normally what is wanted. Both pure delays and servers are delay primitives.
Hence their constructor sets the `delayType` member, table 17.1. This information is used by the scheduler, and is particularly important when determining which event (of several simultaneous) to process first.

### 17.4.1.2 Functional Primitives

In the DE model of computation, a primitive is runnable (ready for execution), if any input port-hole has a new event, and that event has the smallest time stamp of any pending event in the system. When the primitive fires, it may need to know which input or inputs contain the events that triggered the firing. An input port-hole containing a new particle has the `dataNew` flag set by the scheduler. The primitive can check the `dataNew` flag for each input. A functional primitive will typically read the value of the new input particles, compute the value of new output particles, and produce new output particles with time stamps identical to those of the new inputs. To see how this is done, consider the DE primitive `Switch`:

```plaintext
defprimitive
{
  name       { Switch }
  domain     { DE }
  author     { Tommy Baumann }
  copyright  { Copyright (c) 1990-1997 The Regents of the University of California. All rights reserved. See the file $MLD/copyright for copyright notice, limitation of liability, and disclaimer of warranty provisions. }

desc
{
  The function of this primitive is to pass an Input DS to one of the two output ports. The value of the "control" input determines which output is to be enabled. If the value of the "control" input is zero, the input DS is placed on the "falseOut" output port. The input DS is placed on the "trueOut" output port for non-zero values of the "control" input.
}

input
{
  name     { input }
  type     { anytype }
  desc     { This is the DS to be placed on one of the two output ports. }
}

output
```
The *Switch* primitive has two input ports *input* and *control*. When an event arrives at the *input* port, it routes the event to either the *trueOut* or the *falseOut* output port depending on the value of the last received *control* input. In the *go* method, the programmer has to check whether a new input event has arrived. If not, then the firing was triggered by a *control* input event, and there is nothing to do. If the *input* has new data, then its particle is read using *get* method, as summarized in table 17.2. In addition, the most recent value from the *control* input
is read. This value is used to determine which output should receive the data input.

<table>
<thead>
<tr>
<th>Class <strong>InDEPort</strong></th>
<th>Method Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Particle</strong> &amp; operator %</td>
<td>get a particle from the porthole without resetting <strong>dataNew</strong></td>
<td></td>
</tr>
<tr>
<td><strong>void before</strong> (GenericPort &amp; p)</td>
<td>simultaneous inputs here should be processed before those at <strong>p</strong></td>
<td></td>
</tr>
<tr>
<td><strong>int dataNew</strong></td>
<td>flag indicating whether the porthole has new data</td>
<td></td>
</tr>
<tr>
<td><strong>Particle</strong> &amp; get ()</td>
<td>get a particle from the porthole and reset <strong>dataNew</strong></td>
<td></td>
</tr>
<tr>
<td><strong>void getSimulEvent ()</strong></td>
<td>fetch a simultaneous event from the global event queue</td>
<td></td>
</tr>
<tr>
<td><strong>int numSimulEvents ()</strong></td>
<td>return the number of pending simultaneous events at this input</td>
<td></td>
</tr>
<tr>
<td><strong>void triggers ()</strong></td>
<td>indicate that the input does not trigger any immediate output events</td>
<td></td>
</tr>
<tr>
<td><strong>void triggers (GenericPort &amp; p)</strong></td>
<td>indicate that the input triggers an immediate output on port <strong>p</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 17.2: Summary of methods of class **InDEPort**

<table>
<thead>
<tr>
<th>Class <strong>OutDEPort</strong></th>
<th>Method Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Particle</strong> &amp; operator %</td>
<td>get the most recent particle from the porthole</td>
<td></td>
</tr>
<tr>
<td><strong>Particle</strong> &amp; put (double <strong>time</strong></td>
<td>get a new writable particle with the given time stamp</td>
<td></td>
</tr>
<tr>
<td><strong>void sendData ()</strong></td>
<td>flush output porthole data (generated by <strong>put</strong>) to the global event queue</td>
<td></td>
</tr>
</tbody>
</table>

Table 17.3: Summary of methods of class **OutDEPort**

There are three ways to access a particle from an input or output port. First, the programmer can use the % operator followed by an integer, which is equivalent to the same operator in the SDF domain. For example, **control % 0** returns the most recent particle from the **control** porthole. The second method, **get**, is specific to class **InDEPort**. It resets the **dataNew** member of the port as well as returning the most recent particle from an input port. If you need to reset the
dataNew member of an input port after reading the newly arrived event (the more common case) you should use the get method instead of %0 operator. Alternatively, you can reset the dataNew flag explicitly using a statement like:

```plaintext
input.dataNew = FALSE;
```

The put method is specific to OutDEPort. It sets the timeStamp member of the port to the value given by its argument, and returns a reference to the most recent particle from an output port. Considering the line in the above example:

```plaintext
trueOut.put(arrivalTime) = tP;
```

This says that the particle tP is copied to the trueOut output port with

timeStamp = arrivalTime. The programmer can send more than one output event to the same port by calling the put method repeatedly. A new particle is returned each time.

### 17.4.1.3 Simultaneous Events

An input port may have a sequence of simultaneous events pending (events with identical time stamps). Normally, the primitive will be fired repeatedly until all these events have been consumed. Optionally, a DE primitive may process simultaneous events during a single firing. The getSimulEvent method can be used as in the following example, taken from an up-down counter primitive:

```plaintext
go {
    ...
    while (countUp.dataNew) {
        count++;
        countUp.getSimulEvent();
    }
    ...
}
```

Here, countUp is an input porthole. The getSimulEvent method examines the global event queue to see if any more events are available for the porthole with the current time stamp. If so, it fetches the next one and sets the dataNew flag to TRUE. If none remain, it sets the dataNew flag to FALSE. In this example, the actual values of the input events are uninteresting, but the primitive could use get() within the loop if it did need the event values.

Sometimes, a primitive simply needs to know how many simultaneous events are pending on a given porthole. Without fetching any event, we can get the number of simultaneous events by calling the numSimulEvents method. This returns the number of simultaneous events still waiting in the global event queue. The one already in the porthole isn’t counted.

If the primitive has multiple input ports, the programmer should carefully consider the desired behavior of simultaneous inputs on different ports, and choose the order of processing of events.
17.4 DE Domain

accordingly. For example, it might be appropriate to absorb all the events available for a control
porthole before examining any events for a data porthole.

17.4.1.4 Source Primitives

The DE primitives discussed so far fire in response to input events. In order to build signal gen-
erators, or source primitives, or primitives with outputs but no inputs, we need another class of
DE primitive, called a self-scheduling primitive. A self-scheduling primitive fools the scheduler
by generating its own input events. These feedback events trigger the primitive firings. An event
generator is a special case of a delay primitive, its role is mainly to control the time spacing of
source events. The values of the source events can be determined by a functional block attached
to the output of the event generator (e.g. Const, Ramp, etc).

<table>
<thead>
<tr>
<th>Class DERepeatStar</th>
<th>#include &quot;DERepeatStar.h&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>int canGetFired ()</td>
<td>return 1 if the primitive is enabled for firing</td>
</tr>
<tr>
<td>void refireAtTime (double t)</td>
<td>schedule the primitive to fire again at time t</td>
</tr>
<tr>
<td>void begin ()</td>
<td>schedule the primitive to fire at time zero</td>
</tr>
</tbody>
</table>

Table 17.4: Summary of methods of class DERepeatStar

A self-scheduling primitive is derived from class DERepeatStar, which in turn is derived
from class DEStar. The DERepeatStar class has two special methods to facilitate the
self-scheduling function: refireAtTime and canGetFired. These are summarized in
table 17.4. The DE primitive Poisson illustrates these:

defprimitive
{  
  name {Poisson}  
  domain {DE}  
  derivedfrom { RepeatStar }  
  desc  
  {  
    Generates events according to a Poisson process.  
    The first event comes out at time zero.  
  }  
  output  
  {  
    name {output}  
    type {float}  
  }  
  defparameter
The Poisson primitive generates a Poisson process. The inter-arrival time of events is exponentially distributed with parameter meanTime. Refer to sec. 13.6.6 on page 13-53 for information about the random number generation. The method refireAtTime launches an event onto a feedback arc that is invisible to the programmer. The feedback event triggers the self-scheduling primitive some time later.
Note that the feedback event for the next execution is generated in the current execution. To initiate this process, an event is placed on the feedback arc by the `DERepeatStar::begin` method, before the scheduler runs.

The `DERepeatStar` class can also be used for other purposes besides event generation. For example, a sampler primitive might be written to fire itself at regular intervals using the `refireAtTime` method.

Another uncommon named method, `canGetFired` is seldom used in the primitive definitions. The method checks for the existence of a new feedback event, and returns `TRUE` if it is there, or `FALSE` otherwise.

The internal feedback arc consists of an input and an output porthole that are automatically created and connected together, with a delay marker added to prevent the scheduler from complaining about a delay-free loop. This effectively assumes that refire requests will always be for time stamps greater than the current time.

Sometimes the programmer of a primitive derived from `DERepeatStar` needs to be explicitly aware of these portholes. In particular, they should be taken into account when considering whether a primitive is delay-type. Setting `delayType` in a `DERepeatStar` derivative asserts that not only do none of the primitive’s visible input portholes trigger output events with zero delay, but refire events do not either. Frequently this is a false statement. It’s usually safer to write `triggers` directives that indicate that specific input portholes cannot trigger zero-delay outputs. Since the feedback portholes have a delay marker, it is never necessary to mention the feedback output porthole in `triggers` directives, even for an input porthole that gives rise to `refireAtTime` requests, the scheduler is uninterested in zero-delay paths to the feedback output.

The event passed across the feedback arc is an ordinary `FLOAT` particle, normally having value zero. Sometimes it can be useful to store extra information in the feedback event. The `refireAtTime` method accepts an optional second parameter that gives the numeric value to place in the feedback event. Fetching the value currently requires direct access to the feedback input port, for example:

```cpp
if (feedbackIn->dataNew)
{
    double feedbackValue = double(feedbackIn->get());
    ...
}
```

A future version of `DERepeatStar` might provide some syntactic changes to hide the details of this operation.

### 17.4.1.5 Init and WrapUp Primitives

The two primitives `Init` and `WrapUp` changed in the the DE domain with version 2.3. Thereafter systems using these primitives display a warning in the log console when the simulation is executed. The warning asks that you look at the online documentation for details about the changes. It is important to know that these primitives are not automatically replaced by the new primitives. Problems arise if a system uses MLDesigner modules containing these primitives and an instance of the primitive is also used in the system.
The primitive instance in the module has been replaced with the new \texttt{WrapUp} or \texttt{Init} primitive but the primitive instance is not automatically updated. We recommend you replace the old instance of the primitive with the new.

The new \texttt{Init} and \texttt{WrapUp}:

1. allow negative values for relative order;
2. use descending order firing (from relative order variables, so 1 fires before 0).

The old version of \texttt{Init} and \texttt{WrapUp}, (now called \texttt{Init\_MLD2\_3} and \texttt{WrapUp\_MLD2\_3} respectively, are stored in the \textit{Compatibility} library):

1. do not allow negative values for relative order.
2. use ascending order firing (0 fires before 1)

\textbf{NOTE:} Do not mix old and new \texttt{Init} and \texttt{WrapUp} primitives in a system as this will lead to conflicts.

### 17.4.2 Programming Examples

This section presents different examples of programming in the discrete-event domain. We will give several examples of DE primitives that work with matrices.

#### Identity Matrix Primitive

This section develops a primitive in the DE domain that will create an identity matrix. Instead of creating a source primitive which must schedule itself, it is being done with a primitive that fires whenever it receives an new input value. For example, a clock or some other source can be attached to the primitive to set its firing pattern.

```python
defprimitive
{
    name   { Identity_M }
    domain { DE }
    desc   { Output a floating-point identity matrix }
    author { Brian L. Evans }
    input
    {
        name   { input }
        type   { anytype }
    }
    output
    {
        name   { output }
        type   { FLOAT\_MATRIX\_ENV }
    }
    defparameter
    {
        name   { rowsCols }
```
This is a functional primitive because the time stamp on the input particle is not altered. The output is a matrix message. The matrix is a square matrix. In order for the matrix to remain defined after the `go` method finishes, the matrix result cannot be allocated from local memory. Instead, it must be allocated from global dynamic memory via the `new` operator. In the syntax for the `new` operator, the `int` cast in `int(rowsCols)` extracts the value from `rowsCols` which is an instance of the `State` class. The dynamic memory allocated for the matrix will be automatically deleted by the `Message` class. Then, the matrix is reset to be an identity matrix. Finally, the matrix is sent to the output port with the same time stamp as that of the input data. Note that the syntax to output data in the DE domain differs from the syntax of the SDF due to the time stamp. In the SDF domain, the output code would be

```
output%0 << result
```

### Matrix Transpose

In the next example, the matrix transpose will be implemented.

```
name { input }  
  type { FLOAT_MATRIX_ENV }  
}  
output  
{  
  name { output }  
  type { FLOAT_MATRIX_ENV }  
}  
ccinclude { "Matrix.h" }  
go  
{  
// Functional Primitive: pass timestamp without change  
  completionTime = arrivalTime;  

// Extract the matrix on the input port  
Envelope Xpkt;  
input.get().getMessage(Xpkt);  
const FloatMatrix& Xmatrix =  
  *(const FloatMatrix *)Xpkt.myData();  

// Create a copy of the input matrix  
FloatMatrix& xtrans = *(new FloatMatrix(Xmatrix));  

// Transpose the matrix  
xtrans.transpose();  

// Send the matrix result to the output port  
output.put(completionTime) << xtrans;  
}  

The key difference between creating an identity matrix and taking a matrix transpose in the DE domain is the conversion of the input data to a matrix. The input data comes in the form of an envelope which is essentially an instance of a class embedded in a message particle. To extract the contents of the message, the message is first extracted from the input envelope. Then, the data field is taken from the message and cast it to a FloatMatrix. Just as in the previous example, the dynamic memory is needed to be allocated to hold the value of the matrix to be output. In this case, the programmer do not have to code the transpose operation since it is already built into the matrix classes.
Part III

Domains
Chapter 18

SDF Domain

18.1 Introduction

Synchronous data flow (SDF) is a data-driven, statically scheduled domain in MLDesigner. It is a direct implementation of the techniques given in [LM87a] and [LM87b]. "Data-driven" means that the availability of Particles at the inputs of a primitive enables it. Primitives without any inputs are always enabled (including disconnected Xgraphs.) "Statically scheduled" means that the firing order of the primitives is determined once during the start-up phase. The firing order will be periodic. The SDF domain is one of the most mature in MLDesigner, having a large library of primitives and demo programs. It is a simulation domain, but the model of computation is the same as that used in most of the code generation domains. A number of different schedulers, including parallel schedulers, have been developed for this model of computation.

18.2 Basic Data Flow Terminology

SDF is a special case of the data flow model introduced by Dennis [Den75]. In the terminology of the data flow literature, primitives are called actors. An invocation of the `go()` method of a primitive is called a firing. Particles are called tokens. In a digital signal processing system, a sequence of tokens might represent a sequence of samples of a speech signal or a sequence of frames in a video sequence.

When an actor fires, it consumes a number of tokens from its input arcs, and produces a number of output tokens. In synchronous data flow, these numbers remain constant throughout the execution of the system. It is for this reason that this model of computation is suitable for synchronous signal processing systems, but not for asynchronous systems. The fact that the firing pattern is determined statically is both a strength and a weakness of this domain. It means that long runs can be very efficient, a fact that is heavily exploited in the code generation domains. But it also means that data-dependent flow of control is not allowed. This would require dynamically changing firing patterns. The Dynamic Data Flow (DDF) and Boolean Data Flow (BDF) domains were developed to support this.
18.3 Balancing production and consumption of tokens

Each porthole of each SDF primitive has an attribute that specifies the number of particles consumed (for input ports) or the number of particles produced (for output ports). When you connect two portholes with an arc, the number of particles produced on the arc by the source primitive may not be the same as the number of particles consumed from that arc by the destination primitive. To maintain a balanced system, the scheduler must fire the source and destination primitives with different frequency.

Consider a simple connection between three primitives, as shown in fig. 18.1. The symbols adjacent to the portholes, such as $N$, represent the number of particles consumed or produced by that porthole when the primitive fires. For many signal processing primitives, these numbers are simply one, indicating that only a single token is consumed or produced when the primitive fires. But there are three basic circumstances in which these numbers differ from one:

- Vector processing in the SDF domain can be accomplished by consuming and producing multiple tokens on a single firing. For example, a primitive that computes a fast Fourier transform (FFT) will typically consume and produce $2M$ samples when it fires, where $M$ is an integer. Examples of vector processing primitives that work this way are FFTCx, Burg, and LevDur found in the MLD Libraries/SDF/DSP library. This behavior is quite different from the matrix primitives, which operate on particles where each individual particle represents a matrix.
- In multirate signal processing systems, a primitive may consume $M$ samples and produce $N$, thus achieving a sampling rate conversion of $N/M$. For example, the FIR and FIRCx primitives optionally perform such a sampling rate conversion, and with an appropriate choice of filter coefficients, can interpolate between samples. Other primitives that perform sample rate conversion include UpSample, DownSample, and Chop.
- Multiple signals can be merged using primitives such as Commutator or a single signal can be split into sub-signals at a lower sample rate using the Distributor primitive.

![Figure 18.1: A simple connection of SDF primitives, used to illustrate the use of balance equations in constructing a schedule.](image)

To be able to handle these circumstances, the scheduler first associates a simple balance equation
with each connection in the graph. For the graph in fig. 18.1, the balance equations are

\[ r_A N_{A1} = r_C N_{C1} \]
\[ r_A N_{A2} = r_B N_{B1} \]
\[ r_B N_{B2} = r_C N_{C2} \]

This is a set of three simultaneous equations in three unknowns. The unknowns, \( r_A, r_B \) and \( r_C \) are the repetitions of each actor that are required to maintain balance on each arc. The first task of the scheduler is to find the smallest non-zero integer solution for these repetitions. It is proven in [LM87a] that such a solution exists and is unique for every SDF graph that is consistent, as defined below.

### 18.4 Iterations in SDF

When running an SDF system under the graphical user interface, you will have the opportunity to specify the **Run Length**. Since the SDF domain has no notion of time, this is not given in units of time. Instead, it is given in units of SDF iterations. At each SDF iteration, each primitive is fired the minimum number of times to satisfy the balance equations.

Create a system with the modeling domain set as SDF containing a `Const.level` primitive from the Sources Library, a `FFTCx` from the Spectrals Library and a `XMgraph` from the Sinks Library (see fig. 18.2). Select the primitive `FFTCx` and set the parameter `size` to 128. It will consume 128 samples and produce 128 samples. The `Const.level` primitive produces exactly one sample on each output, and the `XMgraph` primitive consumes one sample from each input. In summary,

\[ N_{A1} = N_{A2} = N_{C1} = N_{C2} = 1 \]
\[ N_{B1} = N_{B2} = 128 \]

The balance equations become

\[ r_A = r_C \]
\[ r_A = 128 \cdot r_B \]
\[ 128 \cdot r_B = r_C \]

The smallest integer solution is

\[ r_A = r_C = 128 \]
\[ r_B = 1 \]

Hence, each iteration of the system includes one firing of the `FFTCx` primitive and 128 firings each of primitives \( A \) and \( B \).

### 18.5 Inconsistency

It is not always possible to solve the balance equations. Suppose that in fig. 18.1 we have

\[ N_{A1} = N_{A2} = N_{C1} = N_{C2} = N_{B1} = 1 \]
$N_{B2} = 2$

In this case, the balance equations have no non-zero solution. The problem with this system is that there is no sequence of firings that can be repeated indefinitely with bounded memory. If we fire $A$, $B$, $C$ in sequence, a single token will be left over on the arc between $B$ and $C$. If we repeat this sequence, two tokens will be left over. Such a system is said to be inconsistent, and is flagged as an error. The SDF scheduler will refuse to run it. If you must run such a system, change the domain of your graph to the DDF domain.

### 18.6 Delays

Delays are indicated in MLDesigner by small green diamonds that are placed on an arc. Delays are created using the **Add Delay** tool button on the editor toolbar. Please refer to sec. 3.10.2. The delay has a single parameter, the number of samples of delay to be introduced. In the SDF domain, a delay with parameter equal to one is simply an initial particle on an arc. This initial particle may enable a primitive, assuming that the destination primitive for the delay arc requires one particle in order to fire. To avoid deadlock, all feedback loops must have delays. The SDF scheduler will throw an error if it finds a loop with no delays. For most particle types, the initial value of a delay will be zero. For particles which hold matrices, the initial value is an empty envelope, which must be checked for by primitives which work on matrix inputs. Initializable delays allow the user to give values to the initial particles placed in the buffer.

### 18.7 Targets

#### 18.7.1 Default SDF target

The default SDF target has a simple set of options:

- **logFile** (STRING) The name of a file into which the scheduler will write the final schedule. The initial default is the empty string.
- **loopScheduler** (STRING) Default = DEF A String specifying whether to attempt to compact the schedule for forming looping structure (see below). Choices are DEF, CLUST and
ACYLOOP. The case does not matter: DEF, def, Def are all the same. For backward compatibility, "0" or "NO", and "1" or "YES" are also recognized, with "0" or "NO" being DEF, and "1" or "YES" being CLUST.

schedulePeriod (FLOAT) Default = 0.0
A floating-point number defining the time taken by one iteration through the schedule. This is not needed for pure SDF systems, but if SDF systems are mixed with timed domains, such as DE, then this will determine the amount of simulated time taken by one iteration.

The SDF scheduler determines the order of execution of primitives in a system at start time. It performs most of its computation during its setup() phase. If the loopScheduler target parameter is DEF, then we get a scheduler that exactly implements the method described in [LM87a] for sequential schedules. If there are sample rate changes in a program graph, some parts of the graph are executed multiple times. This scheduler does not attempt to generate loops; it simply generates a linear list of blocks to be executed. For example, if primitive A is executed 100 times, the generated schedule includes 100 instances of A. A loop scheduler will include in its "looped" schedule (where possible) only one instance of A and indicate the repetition count of A, as in (100 A). For simulation, a long unstructured list might be tolerable, but not in code generation. (The SDF schedulers are also used in the code generation for a single processor target).

Neglecting the overhead due to each loop, an optimally compact looped schedule is one that contains only one instance of each actor, and we refer to such schedules as single appearance schedules. For example, the looped schedule (3A)(2B), corresponding to the firing sequence AAABB, is a single appearance schedule, whereas the schedule AB(2A)B is not.

By setting the loopScheduler target parameter to CLUST, we select a scheduler developed by Joe Buck. Before applying the non-looping scheduling algorithm, this algorithm collects actors into a hierarchy of clusters. This clustering algorithm consists of alternating a merging step and a looping step until no further changes can be made. In the merging step, blocks connected together are merged into a cluster if there is no sample rate change between them and the merge will not introduce deadlock. In the looping step, a cluster is looped until it is possible to merge it with the neighbor blocks or clusters. Since this looping algorithm is conservative, some complicated looping possibilities are not always discovered. Hence, even if a graph has a single appearance schedule, this heuristic may not find it.

Setting the loopScheduler target parameter to ACYLOOP results in another loop scheduler being selected, this one developed by Praveen Murthy and Shuvra Bhattacharyya. This scheduler only tackles acyclic SDF graphs, and if it finds that the system is not acyclic, it automatically resets the loopScheduler target parameter to CLUST. This scheduler is optimized for program as well as buffer memory. Basically, for a given SDF graph, there could be many different single appearance schedules. These are all optimally compact in terms of schedule length (or program memory in inline code generation). However, they will, in general, require differing amounts of buffering memory; the difference in the buffer memory requirement of an arbitrary single appearance schedule versus a single appearance schedule optimized for buffer memory usage can be dramatic. Again, in simulation this does not make that much difference (unless really large
SDF graphs with large rate changes are being simulated, but in code generation it is very helpful. Note that acyclic SDF graphs always have single appearance schedules; hence, this scheduler will always give single appearance schedules. If the logFile target parameter is set, then a summary of internal scheduling steps will be written to that file. Essentially, two different heuristics are used by the ACYLOOP scheduler, called APGAN and RPMC, and the better one of the two is selected. The generated file will contain the schedule generated by each algorithm, the resulting buffer memory requirement, and a lower bound on the buffer memory requirement (called BMLB) over all possible single appearance schedules.

Note that the ACYLOOP scheduler modifies the system during its computations; hence, scripted runs that depend on the system remaining in the original state, cannot be used with this scheduler. Since the system reverts to its original state after a run sequence, the ACYLOOP scheduler will work fine in normal usage.

### 18.7.2 The loop-SDF target

An exact looping algorithm, available in an alternative target called the loop-SDF target, was developed by adding post-processing steps to the CLUST loop scheduling algorithm. For lack of a better name, this technique is called “SJS scheduling”, for the first initials of the designers (Shuvra Bhattacharyya, Joe Buck, and Soonhoi Ha). In the post-processing, the attempt was to decompose the graph into a hierarchy of acyclic graphs [BBHL93], for which a compact looped schedule can easily be constructed. Cyclic subgraphs that cannot be decomposed by this method, called tightly interdependent subgraphs, are expanded to acyclic precedence graphs in which looping structures are extracted by the techniques developed in [BL93a] and extensions to these techniques developed by Soonhoi Ha. This scheduling option is selected when the loopTarget is chosen instead of the default SDF target. The target options are:

- **logFile**
- **schedulePeriod**

They have the same interpretation as for the default target, but in the loop-SDF target, **schedulePeriod** has an initial default of 10000.0.

When there are sample rate changes in the program graph, the default SDF scheduler may be much slower than the loop schedulers, and in code generation, the resulting schedules may lead to unacceptably large code size. Buck’s scheduler provides a fast way to get compact looped schedules for many program graphs, although there are no guarantees of optimality. The somewhat slower SJS scheduler is guaranteed to find a single appearance schedule whenever one exists [BBHL95]. Furthermore, a schedule generated by the SJS scheduler contains only one instance of each actor that is not contained in a tightly interdependent subgraph. However, neither the SJS scheduler nor Buck’s scheduler will attempt to optimize for buffer memory usage; this need is met by the ACYLOOP scheduler chosen through the default-SDF target as described above, for acyclic graphs. Algorithms for generating single appearance schedules optimized for buffer memory systematically for graphs that may contain cycles have not yet been implemented.

The looped result can be seen by setting the logFile target parameter. That file will contain all the intermediate procedures of looping and the final scheduling result. The loop scheduling
18.8 An overview of SDF Primitives

Algorithms are usually used in code generation domains, not in the simulation SDF domain. Refer to the Code Generation domain documentation for a detailed discussion to the section on “Schedulers” on page 13-6.

18.7.3 SDF to PTCL target

This target is substantially incomplete, only a rough outline is given below. The SDF-to-PTCL target uses CGMultiInOut primitives to generate abstract ptcl graphs which capture the SDF semantics of a simulation SDF system. These abstract graphs can then be used to test SDF schedulers.

The ptcl output filename will use the system name as a prefix, and append .pt to the name (e.g., the ptcl output for the butterfly demo would be in butterfly.pt). Currently the directory that will contain the ptcl output is hardwired to ~/MLD/ptcl/. You may need to create this directory by hand.

The most interesting aspect about the target is that it collects statistics on the execution time of each primitive. This is valuable for seeing the relative runtime of the various primitives which can be used in code generation. It collects statistics by running the scheduled system, accumulating elapsed CPU time totals for each primitive. This new target does not call the wrapup methods of the primitives, so you will not see XGraph outputs.

18.8 An overview of SDF Primitives

The SDF library is divided into sub-libraries. These are:

- **Arithmetic**: Contains basic adders, subtracters, multipliers, and amplifiers, for all the standard scalar data types (floating point, complex, fixed-point, and integer).

- **Atm**

- **Coding**

- **Comm**: Contains primitives that are specific to digital communications functions, such as pulse shapers, speech coders, and QAM encoders.

- **Comparison**

- **Compatibility**

- **Contrib**

- **Control**: Contains primitives that manipulate the flow of tokens, such as commutators and distributors, downsamplers and upsamplers, and forks.
Conversion
Contains primitives that explicitly accomplish type conversion.

Counters

DSHandling

DSP
Contains various signal processing functions such as fixed and adaptive filters of various types.

Delay

Dfm
Design Flow Management library containing primitives that use strings and files as data types.

Filter

Image
Contains primitives for image and video signal processing.

logic
Contains primitives that perform Boolean and comparison operations, such as and, or, and greater than.

Matlab

Matrix
Contains matrix operators such as matrix addition and multiplication. More complex primitives that use matrix operations internally can be found in other libraries, such as the singular value decomposition and Kalman filters in the Signal Processing library.

MemoryAccess

Misc

Modulation

Neural
Contains neural network primitives.

Nonlinear
Contains primitives that compute transcendental functions, such as logarithm, cosine, sine, and exponential functions, as well as quantizer and table lookup primitives.

NumberGenerators

Radar
18.9 Source primitives

Source primitives are primitives with no inputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. In the data flow model of computation, they are always enabled, and hence can be fired at any time. In the synchronous data flow model, the frequency with which they are fired, relative to other primitives in the system, is determined by the solution to the balance equations. The source primitives are summarized below in no particular order.

18.9.1 Floating Point Sources

**Const**
Output a constant signal with value given by the level parameter (default 0.0).

**Impulse**
Generate a single impulse or an impulse train. Each impulse has an amplitude level (default 1.0). If period (default 0) is equal to 0, then only a single impulse is generated; otherwise, period specifies the period of the impulse train.

**IIDGaussian**
Generate an identically independently distributed white Gaussian pseudo-random process with mean (default 0) and variance (default 1).

**IIDUniform**
Generate an identically independently distributed uniformly distributed pseudo-random process. Output is uniformly distributed between lower (default 0) and upper (default 1).
Ramp
Generate a ramp signal, starting at value (default 0.0) and incrementing by step size step (default 1.0) on each firing.

RanConst
Generate a random number with a uniform, exponential, or normal distribution, as determined by the distribution parameter.

ReadFile
Read ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.

ReadVar
Output the value of a double-precision floating point variable from a shared memory. Use the writeVar primitive to write values into the shared memory. WARNING: This primitive may produce unpredictable results, since the results will depend on the precedences in the block diagram in which it appears as well as the scheduler used.

Rect
Generate a rectangular pulse of height (default 1.0) and width (default 8). If period is greater than zero, then the pulse is repeated with the given period.

WaveForm
Output a waveform as specified by the array parameter value (default \([1 - 1]\)). You can get periodic signals with any period, and can halt a simulation at the end of the given waveform. The following table summarizes the capabilities:

<table>
<thead>
<tr>
<th>haltAtEnd</th>
<th>periodic</th>
<th>period</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>YES</td>
<td>0</td>
<td>The period is the length of the waveform</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>(N &gt; 0)</td>
<td>The period is (N)</td>
</tr>
<tr>
<td>NO</td>
<td>NO</td>
<td>anything</td>
<td>Output the waveform once, then zeros</td>
</tr>
<tr>
<td>YES</td>
<td>anything</td>
<td>anything</td>
<td>Stop after outputting the waveform once</td>
</tr>
</tbody>
</table>

Table 18.1: parameters for WaveForm primitive

The first line of the table gives the default settings. This primitive may be used to read a file by simply setting value to something of the form < filename, preferably specifying a complete path.

Window
Generate standard window functions or periodic repetitions of standard window functions. The possible functions are: Rectangle, Bartlett, Hanning, Hamming, Blackman, Kaiser and SteepBlackman. One period of samples is produced at each firing.

TclScript
Invoke a Tcl script that can optionally define a procedure that is invoked every time the primitive fires. That procedure can read the prim-
18.9 Source primitives

itive’s inputs and update the value of the outputs.

TkSlider
Output a value determined by an interactive on-screen scale slider.

TkButtons
This primitive outputs the value 0.0 on all outputs unless the corresponding button is pushed. When the button is pushed, the output takes the value given by the parameter value. If synchronous is YES, then outputs are produced only when some button is pushed. I.e., the primitive waits for a button to be pushed before its go() method returns. If allow_simultaneous_events is YES, then the buttons pushed are registered only when the button labeled ”PUSH TO PRODUCE OUTPUTS” is pushed. Note that if synchronous is NO, this primitive is nondeterminate.

18.9.2 Fixed-point sources

ConstFix
Constant source for fixed-point values.

RampFix
Ramp for fixed-point values.

RectFix
Generate a fixed-point rectangular pulse of height (default 1.0) and width (default 8). If period is greater than zero, then the pulse is repeated with the given period. The precision of height can be specified in bits.

18.9.3 Complex sources

ConstCx
Constant source for complex values.

WaveFormCx
Output a complex waveform as specified by the array parameter value (default ”(1,0) (-1,0)”). Note that ”(a,b)” means a+jb. The parameters work the same way as in the WaveForm primitive.

expgen
Generate a complex exponential with the given frequency (relative to the samplerate parameter).

RectCx
Generate a rectangular pulse of height (default 1.0) and width (default 8). If period is greater than zero, then the pulse is repeated with the given period.

bits
Produce ”0” with probability probOfZero, else produce ”1”.

RampInt
Ramp for integer values.

PCMReadInt
Read a binary µ-law encoded PCM file. Return one sample on each firing. The file format that is read is the same as the one written by the Play primitive. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.

ConstInt
Constant source for integer values.
18.9.4 Matrix Sources

The Matrix and Identity primitives each have four different icons for the different matrix data types.

**Matrix**

Produce a matrix with floating-point entries. The entries are read from the array parameter FloatMatrixContents in rasterized order: i.e., for an M x N matrix, the first row is filled from left to right using the first N values from the array.

**Matlab**

Evaluate a Matlab function if inputs are given or evaluate a Matlab command if no inputs are given. Any Matlab script can be evaluated, provided that the current machine has a license to run Matlab.

**MatlabCx**

Complex version of the above primitive.

**Identity**

Output a floating-point identity matrix.

18.10 Sink primitives

The primitives in this library have no output ports. They display signals in various ways, write output datasets to files or simply discard the input particles (see Black Hole).

18.10.1 Batch Plotting Facilities

The first six primitives in this library are all based on the pxgraph program. This program has many options, summarized in ch. 8.2. The differences between primitives often amount to little more than the choice of default options. Some, however, preprocess the signal in useful ways before passing it to the pxgraph program. The first allows only one input signal, the second allows any number (notice the double arrow on the input port).

**BlackHole**

Discards the input particles. This is exactly the same as terminating a output port via the context menu.

**DmpNFloat**

The Float output dataset is written to a file. The default path is $MLD_USER/DmpNFloat.out$.

**DmpNInt**

The Integer output dataset is written to a file. The default path is $MLD_USER/DmpNFloat.out$.

**Play**

Play an input stream on the SparcStation speaker. The "gain" parameter (default 1.0) multiplies the input stream before it is mu-law compressed. The "gain" should chosen to scale the input values in the range -32000.0 to 32000.0. The scaled input values are compressed from 16-bit linear amplitude format to sign magnitude mu-law 8-bit format and written to a file. When the wrapup method is called, a file of 8-bit mu-law will be played at the fixed sampling rate of 8000 samples/second by the "ptplay" program, which must be in your path.

**PlayAIFF**

Play an Audio Interchange File Format (AIFF) input stream on the workstation speaker.
18.11 Arithmetic primitives

PlayAIF2
Play a stereo AIFF input stream on the workstation speaker.

Printer
Print out one sample from each input port per line. A "fileName" can be specified via the appropriate dialog; the options stdout and cout which specify the standard output stream, and stderr and cerr which specify the standard error stream, are also supported.

Scatter
This primitive writes a complex input to a file and an XYgraph. The dataset is stored in a data file which is specified using the parameter fileName. The graph is displayed as a scatter diagram. In the data file the values of the signal elements are arranged as follows:

Re(1), Im(1), Re(2), Im(2), ..., Re(k), Im(k)

where k is the number of complex elements processed.

Xgraph
Generate a generic single-signal plot with the pxgraph program.

XMgraph
Generate a generic multi-signal plot.

XYgraph
Generate an X-Y plot with the pxgraph program. The X data is on xInput and the Y data is on input.

Xscope
Generate a multi-trace plot with the pxgraph program. Successive traces are overlaid on one another.

Xhistogram
Generate a histogram with the pxgraph program. The parameter binWidth determines the bin width.

Waterfall
Plot a series of traces in the style of a waterfall plot. This is a type of three-dimensional plot used to show the evolution of signals or spectra. Optionally, each plot can be made opaque, so that lines that would appear behind the plot are eliminated.

WriteVar
Generate a generic single-signal plot with the pxgraph program.

18.11 Arithmetic primitives

In principle, it should be possible to overload the basic arithmetic operators so that, for example, a single Add primitive could handle any data type. The decision, however, was in favor of more explicit typing, in which there is an Add primitive for each particle type supported in the kernel. As before, when there is no data type suffix in the name of the primitive, the data type supported is double-precision floating point.

Each primitive type has equivalent primitives for floating-point, complex, fixed-point, and integer arithmetic, respectively. The basic primitive type functions are:

Add  Output the sum of the inputs.
Sub  Output the "pos" input minus all "neg" inputs.
Mpy  Output the product of the inputs.
Gain This is an amplifier; the output is the input multiplied by the gain (default 1.0).
The floating-point and complex-valued scalar data types also have the following primitive:

**Average**

Average some number of input samples or blocks of input samples. Blocks of successive input samples are treated as vectors.

The floating-point type has one additional arithmetic primitive:

**Integrator**

This is an integrator with leakage, limits, and reset. With the default parameters, input samples are simply accumulated, and the running sum is the output. To prevent any resetting in the middle of a run, connect a `Const` source with value 0 to the "reset" input. Otherwise, whenever a non-zero is received on this input, the accumulated sum is reset to the current input (i.e. no feedback).

Limits are controlled by the `top` and `bottom` parameters. If `top≤bottom`, no limiting is performed (this is the default). Otherwise, the output is kept between `bottom` and `top`. If `saturate=YES`, saturation is performed. If `saturate=NO`, a wrap-around is performed (this is the default). Limiting is performed before output. Leakage is controlled by the `feedbackGain` parameter (default 1.0). The output is the data input plus `feedbackGain·state`, where `state` is the previous output.

The integer type has the following primitive:

**DivByInt**

This is an amplifier. The integer "output" is the integer "input" divided by the integer divisor (default 1). Truncated integer division is used.

### 18.12 Nonlinear primitives

The nonlinear library in the SDF domain includes transcendental functions, quantizers, table lookup primitives, and miscellaneous nonlinear functions.

#### 18.12.1 Quantizers

**AdaptLinQuant**

Quantize the input to one of $2^{bits}$ possible output levels. The high and low output levels are anti-symmetrically arranged around zero and their magnitudes are determined by $(2^{bits} - 1) \cdot \frac{inStep}{2}$. The steps between levels are uniformly spaced at the step size given by the "inStep" input value. The linear quantizer can be made adaptive by feeding back past information such as quantization level, quantization value, and step size into the current step size.

**LinQuantIdx**

Quantize the input to the number of levels given by the `levels` parameter. The quantization levels are uniformly spaced between `low` and `high` inclusive. Rounding down is performed, so that output level will equal high only if the input level equals or exceeds high. If the input is below `low`, then the quantized output will equal low.
The quantized value is output to the "amplitude" port, while the index of the quantization level is output to the "stepNumber" port.

**Quant**
Quantize the input value to one of \( N + 1 \) possible output levels using \( N \) thresholds. For an input less than or equal to the \( n \)-th threshold, but larger than all previous thresholds, the output will be the \( n \)-th level. If the input is greater than all thresholds, the output is the \( N + 1 \)-th level. If level is specified, there must be one more level than thresholds; the default value for level is 0, 1, 2, \ldots \, N. This primitive is much slower than LinQuantIdx, so if possible, that one should be used instead.

**QuantIdx**
Quantize the input value to one of \( N + 1 \) possible output levels using \( N \) thresholds, and output both the quantized result and the quantization level. See the Quant primitive for more information.

**Quantizer**
This primitive quantizes the input value to the nearest output value in the given codebook. The nearest value is found by a full search of the codebook, so the primitive will be significantly slower than either Quant or LinQuantIdx. The absolute value of the difference is used as a distance measure.

### 18.12.2 Math Functions

**Abs**
Compute the absolute value of its input.

**cexp**
Compute the complex exponential function of its complex input. See also expjx.

**conj**
Compute the conjugate of its complex input.

**Cos**
Compute the cosine of its input, assumed to be an angle in radians.

**Dirichlet**
Compute the normalized Dirichlet kernel (also called the aliased sinc function):

\[
d_N(x) = \frac{\sin(Nx/2)}{N \cdot \sin(x/2)}
\]

The value of the normalized Dirichlet kernel at \( x = 0 \) is always 1, and the normalized Dirichlet kernel oscillates between -1 and +1. The normalized Dirichlet kernel is periodic in \( x \) with a period of either \( 2\pi \) when \( N \) is odd or \( 4\pi \) when \( N \) is even.

**Exp**
Compute the real exponential function of its real input.

**expjx**
Compute the complex exponential function of its real input. See also cexp.

**Floor**
Output the greatest integer less than or equal to its input.

**Log**
Output the natural logarithm of its input.

**Limit**
The output of this primitive is the value of the input limited to the range between bottom and top inclusive.

**MaxMin**
Finds maximum or minimum, value or magnitude, of a fixed number of
data values on its input. If you want to use this primitive to operate over multiple data streams, then precede this primitive with a Commutator and set the parameter $N$ accordingly.

**Modulo**
The output is equal to the remainder after dividing the input by the modulo parameter.

**ModuloInt**
The output is equal to the integer remainder after dividing the integer input by the integer modulo parameter.

**OrderTwoInt**
Takes two inputs and outputs the greater and lesser of the two integers.

**Reciprocal**
Output the reciprocal of its input, with an optional magnitude limit (magLimit). If magLimit is greater than zero, and the input value is zero, then the output will equal magLimit.

**Sgn**
Compute the signum of its input. The output is $\pm 1$. Note that 0.0 maps into 1.

**Sin**
Computes the sine of its input, assumed to be an angle in radians.

**Sinc**
Computes the sinc of its input given in radians. The sinc function is defined as $\sin(x)/x$, with value 1.0 when $x = 0$.

**Sqrt**
Computes the square root of its input.

### 18.12.3 Other Nonlinear Functions

**DB**
Convert input to a decibels (dB) scale. Zero and negative values are assigned the value min (default -100). The inputIsPower parameter should be set to YES if the input signal is a power measurement (vs. an amplitude measurement).

**PcwzLinear**
This primitive implements a piecewise linear mapping from the list of (x,y) pairs, which specify the breakpoints in the function. The sequence of x values must be increasing. The function implemented by the primitive can be represented by drawing straight lines between the (x,y) pairs, in sequence. The default mapping is the tent map, in which inputs between -1.0 and 0.0 are linearly mapped into the range -1.0 to 1.0. Inputs between 0.0 and 1.0 are mapped into the same range, but with the opposite slope, 1.0 to -1.0. If the input is outside the range specified in the “x” values of the breakpoints, then the appropriate extreme value will be used for the output. Thus, for the default map, if the input is -2.0, the output will be -1.0. If the input is +2.0, the output will again be -1.0.

**powerEst**
Estimate the power in decibels (dB) by filtering the square of the input using a first-order filter with the time constant given as a number of sample periods.

**powerEstCx**
Like powerEst, but for complex inputs.

**powerEstLin**
Same as powerEst, but the output is on a linear scale instead of...
decibels (dB).

**Table**

This primitive implements a real-valued lookup table indexed by an integer-valued input. The input must lie between 0 and \( N - 1 \), inclusive, where \( N \) is the size of the table. The values parameter specifies the table. Its first element is indexed by a zero-valued input. An error occurs if the input value is out-of-bounds.

**TableCx**

Table lookup for complex values.

**TableInt**

Table lookup for integer values.

**TclScript**

Invoke a Tcl script that can optionally define a procedure that is invoked every time the primitive fires. That procedure can read the primitive’s inputs and update the value of the outputs.

### 18.13 Logic primitives

The logic library contains a number of base primitives. To select the specific type of primitive function you require, drag and drop the primitive into a system or module to open the select special primitive dialog shown in fig. 18.3.

**Test**

Compare two inputs. The test condition can be any of \{EQ, NE, GT, GE\} or \{==, !=, >, >=\}, resulting in equals, not equals, greater than, or greater than or equals. The four icons represent these possibilities. If crossingsOnly is TRUE, then the output is non-zero only when the outcome of the test changes from \( \text{TRUE} \) to \( \text{FALSE} \) or \( \text{FALSE} \) to \( \text{TRUE} \). In this case, the first output is always \( \text{TRUE} \).

**Multiple**

Output a 1 if the top input is a multiple of the bottom input.

**Logic**

This primitive applies a logical operation to any number of inputs. The inputs are integers interpreted as Booleans, where zero is a \( \text{FALSE} \) and nonzero is a \( \text{TRUE} \). The logical operations supported are \{NOT, AND, NAND, OR, NOR, XOR, XNOR\}, with any number of inputs.
18 SDF Domain

18.14 Control primitives

Control primitives manipulate the flow of tokens. All of these primitives are polymorphic; they operate on any data type. The control primitives can be divided into three categories. They are:

18.14.1 Single-Rate Operations

Fork
Copy input particles to each output. Note that a fork is automatically inserted in a schematic when a single output is sent to more than one input. However, when a delay is needed on one of the connections, then an explicit fork primitive must be used.

Reverse
On each execution, read a block of $N$ samples (default 64) and write them out backwards.

Transpose
Transpose a rasterized matrix (one that is read as a sequence of particles, row by row, and written in the same form). The number of particles produced and consumed equals the product of $\text{samplesInaRow}$ and $\text{numberOfRows}$.

Trainer
Pass the value of the train input to the output for the first train-Length samples, then pass the decision input to the output. This primitive is designed for use with adaptive equalizers that require a training sequence at start-up, but it can be used whenever one sequence is used during a start-up phase, and another sequence after that.

18.14.2 Multirate Operations

Commutator
Synchronously combine $N$ input streams (where $N$ is the number of inputs) into one output stream. The primitive consumes $B$ input particles from each input (where $B$ is the block size), and produces $N \times B$ particles on the output. The first $B$ particles on the output come from the first input, the next $B$ particles from the next input, etc.

DownSample
Decimate by a given factor (default 2). The phase tells which sample of the last factor samples to output. If phase is 0 (by default), the most recent sample is the output, while if phase is the $-1$, the oldest sample is the output. Note that phase has the opposite sense of the phase parameter in the UpSample primitive, but the same sense as the phase parameter in the FIR primitive.

Distributor
Synchronously split one input stream into $N$ output streams, where $N$ is the number of outputs. The primitive consumes $N \times B$ input particles, where $B$ is the blockSize parameter, and sends the first $B$ particles to the first output, the next $B$ particles to the next output, etc.

Repeat
Repeat each input sample a specified number of times.

UpSample
Upsample by a given factor (default 2), giving inserted samples the
value fill (default 0.0). The phase parameter (default 0) tells where to put the sample in an output block. A phase of 0 says to output the input sample first, followed by the inserted samples. The maximum phase is equal to factor $-1$. Although the fill parameter is a floating-point number, if the input is of some other type, such as complex, then the fill particle will be obtained by casting fill to the appropriate type.

18.14.3 Other Operations

**Chop**  
On each execution, this primitive reads a block of nread particles and writes them to the output with the given offset. The number of particles written is given by nwrite. The output block contains all or part of the input block, depending on offset and nwrite. The offset specifies where in the output block the first (oldest) particle in the input block will lie. If offset is positive, then the first offset output particles will be either particles consumed on previous firings (if use_past_inputs parameter is YES), or zero (otherwise). If offset is negative, then the first offset input particles will be discarded.

**ChopVarOffset**  
This primitive has the same functionality as the Chop primitive except the offset parameter is determined at run time by a control input.

**DeMux**  
Demultiplex one input onto any number of output streams. The primitive consumes $B$ particles from the input, where $B$ is the blockSize. These $B$ particles are copied to exactly one output, determined by the ”control” input. The other outputs get a zero of the appropriate type. Integers from 0 through $N - 1$ are accepted at the ”control” input, where $N$ is the number of outputs. If ”control” is outside this range, all outputs get zeros.

**Mux**  
Multiplex any number of inputs onto one output stream. $B$ particles are consumed on each input, where $B$ is the blockSize. But only one of these blocks of particles is copied to the output. The one copied is determined by the ”control” input. Integers from 0 through $N - 1$ are accepted at the ”control” input, where $N$ is the number of inputs. If ”control” is outside this range, an error is signaled.

18.15 Conversion primitives

This library shows a collection of primitives for format conversions of various types. The first two rows contain primitives with functions that are fundamentally different from the automatic type conversion performed by MLDesigner.
18 SDF Domain

18.15.1 Complex data type formats

CxToRect  Convert a complex input to real and imaginary parts.

RectToCx  Convert real and imaginary inputs to a complex output.

RectToPolar  Convert real and imaginary inputs into magnitude and phase form. The
           phase output is in the range \(-\pi\) to \(\pi\).

PolarToRect  Convert magnitude and phase to rectangular form.

18.15.2 Other data type formats

PCMBitCoder  Encode voice samples for a 64 kbps bit stream using CCITT Recom-
               mendation G.711. The input is one 8 kHz sample of voice data and the
               output is the eight-bit codeword (the low-order 8 bits of an integer)
               representing the quantized samples.

\(\mu\)Law  This primitive encodes its input into an 8 bit representation using the
           nonlinear companding \(\mu\)-law. It is similar to PCMBitCoder, but it
           does the conversion in a single primitive, rather than a module.

PCMBitDecoder  Decode 8-bit PCM codewords that were encoded using
                PCMBitCoder.

BitsToInt  The integer input sequence is interpreted as a bit stream in which any
           non-zero value is a ”1” bit. This primitive consumes \(n\)Bits succes-
           sive bits from the input, packs them into an integer, and outputs the
           resulting integer. The first received bit becomes the most significant
           bit of the output. If \(n\)Bits is larger than the integer word size, then
           the first bits received will be lost. If \(n\)Bits is smaller than the word
           size minus one, then the output integer will always be non-negative.

IntToBits  Read the least significant \(n\)Bits bits from an integer input, and output
           the bits as integers serially on the output, most significant bit first.

BusToNum  This primitive accepts a number of input bit streams, where this num-
           ber should not exceed the word size of an integer. Each bit stream
           has integer particles with values 0, 3, or anything else. These are inter-
           preted as binary 0, tri-state, or 1, respectively. When the primitive
           fires, it reads one input bit from each input. If any of the input bits
           is tri-stated, the output will be the previous output (or the initial value
           of the previous parameter if the firing is the first one). Otherwise,
           the bits are assembled into an integer word, assuming two’s comple-
           ment encoding, and sign extended. The resulting signed integer is sent
           to the output. This primitive is particularly useful for interfacing to
digital logic simulation domains.

NumToBus  This primitive accepts an integer and outputs the low-order bits that
           make up the integer on a number of outputs, one bit per output. The
           number of outputs should not exceed the word size of an integer. This
           primitive is particularly useful for interfacing to digital logic simula-
Automatic type conversion has limitations. If a given output port has more than one destination, then all destinations must have the same type input. This is true even if an explicit fork primitive is used. Explicit type conversions are needed to get around this limitation. For this reason, the library also contains a set of type conversions that behave exactly the same way the automatic type conversions behave.

18.15.3 Matrix Conversion Primitives

The following type conversions construct a new matrix of the destination type by converting each element of the old matrix as it is copied to the new one. For FixMatrix types, the precision is specified as a parameter of the conversion primitive. The actual conversions are implemented using the cast conversion in the underlying class, except for the conversions to the FixMatrix type which are more complex because they involve possible changes in precision and require a rounding option. The primitives provided are:

- **IntToFix_M** Convert an integer input matrix to a fixed-point output matrix.
- **IntToFloat_M** Convert an integer input matrix to a floating-point output matrix.
- **IntToCx_M** Convert an integer input matrix to a complex output matrix.
- **FixToInt_M** Convert a fixed-point input matrix to an integer output matrix.
- **FixToFloat_M** Convert a fixed-point input matrix to a floating-point output matrix.
- **FixToCx_M** Convert a fixed-point input matrix to a complex output matrix.
- **FloatToInt_M** Convert a floating-point input matrix to an integer output matrix.
- **FloatToFix_M** Convert a floating-point input matrix to a fixed-point output matrix.
- **FloatToCx_M** Convert a floating-point input matrix to a complex output matrix.
18.16 Matrix primitives

The primitives in the matrix library operate on particles that represent matrices with floating-point, fixed-point, complex, or integer entries. Most of the work is done in the underlying matrix classes, FloatMatrix, ComplexMatrix, FixMatrix, and IntMatrix. These classes are treated as ordinary particles. In MLDesigner, matrix types are indicated with thick stems, where the color of the terminal stem corresponds to the data type of the matrix elements.

The Matrix conversion primitives are in the conversion library, see "Matrix Conversion Primitives" on page 5-22 for more information.

18.16.1 Matrix-Vector Conversion

Acceptor MxCom_M

Accept input matrices and create a matrix output. Each input matrix represents a decomposed sub-matrix of output matrix in row by row. Note that for one output image, we will need a total \((\text{numRows}/\text{numRowsSubMx}) \times (\text{numCols}/\text{numColsSubMx})\) input matrices.

Decomposer MxDecom_M

Decompose a portion of input matrix into a sequence of sub-matrices. The desired portion of input matrix is specified by the parameters startRow, startCol, numRows and numCols. Then output each submatrix with dimension numRowsSubMx \(\times\) numColsSubMx in row by row. Note that for one input matrix, there will be a total of \((\text{numRows}/\text{numRowsSubMx}) \times (\text{numCols}/\text{numColsSubMx})\) output matrices.

The following conversions perform more interesting functions. They also come in four versions, one for each data type, and again we only list the floating-point version.

Producer Pack_M

Produce a matrix with floating-point entries constructed from floating-point input particles. The inputs are put in the matrix in rasterized order, e.g. for a \(M \times N\) matrix, the first row is filled from left to right using the first N input particles.

Generator Toeplitz_M

Generate a floating-point data matrix \(X\), with dimensions \((\text{numRows}, \text{numCols})\), from a stream of \(\text{numRows} + \text{numCols} - 1\)
input particles organized as shown below:

\[
X = \begin{bmatrix}
x(M-1) & x(M-2) & \cdots & x(0) \\
x(M) & x(M-1) & \cdots & x(1) \\
\vdots & \vdots & \ddots & \vdots \\
x(N-1) & x(N-2) & \cdots & x(N-M)
\end{bmatrix}
\]

Here numRows = \(N - M + 1\) and numCols = \(M\). This Toeplitz matrix is the form of the matrix that is required by the \(\text{SVD}_M\) primitive, among others.

\text{UnPk}_M \quad \text{Read a floating-point matrix and output its elements, row by row, as a stream of floating-point particles.}

### 18.16.2 Matrix operations

The following blocks are functions defined only for the ComplexMatrix data type.

- \text{Conjugate}_M \quad \text{Conjugate a matrix.}
- \text{Hermitian}_M \quad \text{Perform a Hermitian transpose (conjugate transpose) on the input matrix.}

The following blocks also appear in the signal processing library.

- \text{SmithForm} \quad \text{Decompose an integer matrix } S \text{ into one of its Smith forms } S = UDV, \text{ where } U, D, \text{ and } V \text{ are simpler integer matrices. The Smith form decomposition for integer matrices is analogous to singular value decomposition for floating-point matrices.}
- \text{SVD}_M \quad \text{Compute the singular-value decomposition of a Toeplitz data matrix } A \text{ by decomposing } A = UWV', \text{ where } U \text{ and } V \text{ are orthogonal matrices, and } V' \text{ represents the transpose of } V. W \text{ is a diagonal matrix composed of the singular values of } A, \text{ and the columns of } U \text{ and } V \text{ are the left and right singular vectors of } A.

The following are usual matrix operations. They are arranged row by row, with one row for each data type (floating point, complex, fixed point, and integer). We list below only the floating point data types.

- \text{Add}_M \quad \text{Add two floating-point matrices.}
- \text{Gain}_M \quad \text{Multiply a floating-point matrix by a static scalar gain value.}
- \text{Inverse}_M \quad \text{Invert a square floating-point matrix.}
- \text{Mpy}_M \quad \text{Multiply two floating-point matrices } A \text{ and } B \text{ to produce matrix } C. \text{ Matrix } A \text{ has dimensions (numRows, } x) \text{. Matrix } B \text{ has dimensions } (x, \text{numCols}). \text{ Matrix } C \text{ has dimensions } (\text{numRows}, \text{numCols}). \text{ The user need only specify numRows and numCols. An error will be}
generated if the number of columns in $A$ does not match the number of rows in $B$.

- **Sub$_M$**: Subtract floating-point matrix $B$ from $A$.
- **Transpose$_M$**: Transpose a floating-point matrix read as a single particle.
- **SubMx$_M$**: Find a submatrix of the input matrix.
- **MpyScalar$_M$**: Multiply a floating-point matrix by a scalar gain value given in parameter.

### 18.16.3 Miscellaneous

- **Table$_M$**: (3 primitives for floating-point, complex and integer) This primitive implements a lookup table indexed by an integer-valued input. The output is a matrix. The input must lie between 0 and $N - 1$, inclusive, where $N$ is the number of matrices in the table. The `floatTable` parameter specifies the entries of matrices in the table. Note that the entries of each matrix in the table should be given in row major ordering. The first matrix in the table is indexed by a zero-valued input. An error occurs if the input value is out of bounds.
- **SampleMean**: Find the average amplitude of the components of the input matrix.
- **AvgSqrErr**: Find the average squared error between two input sequences of matrices.
- **Abs$_M$**: Return the absolute value of each entry of the floating-point matrix.

### 18.17 Matlab primitives

The Matlab primitives provide an interface between MLDesigner and Matlab, a numeric computation and visualization environment from The Math Works, Inc. Each Matlab primitive can contain a single Matlab function, command, statement, or several statements. MLDesigner handles the conversion of inputs into Matlab format and the results from Matlab into MLDesigner format. For the Matlab primitives to work, Matlab version 4.1 or later must be installed.

**NOTE:** Matlab is not distributed with MLDesigner.

If a Matlab primitive is run and Matlab is not installed, then MLDesigner will report an error. All Matlab primitives send their commands to the same Matlab process. Xavier Warzee of Thomson-CSF provided a method of running Matlab on a remote machine and obtaining the results from within MLDesigner. If a simulation needs to start Matlab, then the `PTMATLAB_REMOTE_HOST` environment variable is checked. If this variable is set, then its value is assumed to be the name of the remote machine to run Matlab on. The remote Matlab process is started up with the Unix `rsh` command. Once the remote process is running, if the `MATLAB_SCRIPT_DIR` environment variable is set, then its value is passed to the remote Matlab process as part of the command

```
path(path.'MATLAB_SCRIPT_DIR')
```
where MATLAB_SCRIPT_DIR is the value of that variable on the local machine. Internally, Matlab distinguishes between real matrices and complex matrices. As a consequence, there are two types of Matlab primitives: one outputs floating-point matrices and one outputs complex-valued matrices. These primitives can take any number of inputs provided that the inputs have the same data type (floating point or complex). The two types of Matlab primitives are:

- Matlab_M: Evaluate a Matlab expression and output the result as floating-point matrices.
- MatlabCx_M: Evaluate a Matlab expression and output the result as complex-valued matrices.

The implementation of Matlab primitives is built on Matlab’s engine interface. The interface is managed by a base primitive, SDFMatlab. The base primitive does not have any inputs or outputs. It provides methods for starting and killing a Matlab process, evaluating Matlab commands, managing Matlab figures, changing directories in Matlab, and passing MLDesigner matrices in and out of Matlab. Currently, the base primitive does support real- and complex-valued matrices, but not Matlab’s other two matrix data types, sparse and string matrices.

Figures generated by a Matlab primitive are managed according to the value of the primitive’s DeleteOldFigures parameter. If TRUE or YES, then the Matlab primitive will close any plots, graphics, etc., that it has generated when the Matlab primitive is destroyed (e.g., when the run panel in the graphical interface is closed). Otherwise, the figures remain until MLDesigner exits. It is better to set this parameter to NO so that the plots will not disappear when the standalone program finishes.

There are several ways in which Matlab commands can be specified in the Matlab primitives. The Matlab primitives Matlab_M and MatlabCx_M have a parameter MatlabFunction. If only a Matlab function name is given for this parameter, then the function is applied to the inputs in the order they are numbered and the output(s) of the function is (are) sent to the primitive’s outputs. For example, specifying eig means to perform the eigen decomposition of the input. The function will be called to produce one or two outputs, according to how many output ports there are. If there is a mismatch in the number of inputs and/or outputs between the MLDesigner primitive and the Matlab function, MLDesigner will report the error generated by Matlab.

The user may also specify how the inputs are to be passed to a Matlab function or how the outputs are taken from the Matlab function. For example, consider a two-input, two-output Matlab primitive to perform a generalized eigen decomposition. The command

\[
\text{[output\#2, output\#1] = eig( input\#2, input\#1 )}
\]

says to perform the generalized eigen decomposition on the two input matrices, place the generalized eigenvectors on output\#2, and the eigenvalues (as a diagonal matrix) on output\#1. Before this command is sent to Matlab, the pound characters ‘#’ are replaced with underscore ‘_’ characters because the pound character is illegal in a Matlab variable name.

The Matlab primitives also allow a sequence of commands to be evaluated. Continuing with the previous example, we can plot the eigenvalues on a graph after taking the generalized eigen
decomposition:

```matlab
[output#2, output#1] = eig( input#2, input#1 );
plot( output#1 )
```

When entering such a collection of commands in MLDesigner, both commands would appear on the same line without a newline after the semicolon. In this way, very complicated Matlab commands can be built up. We can make the plot of eigenvalues always appear in the same plot without interfering with other plots generated by other Matlab primitives:

```matlab
[output#2, output#1] = eig( input#2, input#1 );
if ( exist('myEigFig') == 0 ) myEigFig = figure; end;
figure(myEigFig);
plot( output#1 );
```

### 18.18 Signal processing (DSP) primitives

This library contains various primitives which perform signal processing functions. These include fixed and adaptive filters of various types.

#### 18.18.1 Filters

**Biquad**
A two-pole, two-zero Infinite Impulse Response filter (a biquad). The default is a Butterworth filter with a cutoff at 0.1 times the sample frequency. The transfer function is

\[
H(z) = \frac{n_0 + n_1z^{-1} + n_2z^{-2}}{1 - d_1z^{-1} + d_2z^{-2}}
\]

**Convolve**
Convolve two causal finite sequences of floating point numbers. The truncationDepth parameter specifies the number of terms used in the convolution sum. Set truncationDepth larger than the number of output samples of interest.

**ConvolveCx**
Convolve two causal finite sequences of complex numbers. The truncationDepth parameter specifies the number of terms used in the convolution sum. Set truncationDepth larger than the number of output samples of interest.

**FIR**
A Finite Impulse Response (FIR) filter. Coefficients are specified by the taps parameter. The default coefficients give an 8th order, linear-phase, low-pass filter. To read coefficients from a file, replace the default coefficients with `fileName`, preferably specifying a complete path. Rational sampling rate changes, implemented by polyphase multirate filters, is also supported.

**FIRCx**
A complex FIR filter. Coefficients are specified by the taps parameter. The default coefficients give an 8th order, linear phase, low-pass filter. To read coefficients from a file, use the syntax: `fileName`,...
preferably specifying a complete path. Real and imaginary parts should be paired with parentheses, e.g. (1.0, 0.0). Polyphase multirate filtering is also supported.

**RaisedCosine**

An FIR filter with a magnitude frequency response that is shaped like the standard raised cosine or square-root raised cosine used in digital communications. By default, the primitive upsamples by a factor of 16, so 16 outputs will be produced for each input unless the interpolation parameter is changed.

**FIRFix**

An FIR filter with fixed-point capabilities. The fixed-point coefficients are specified by the taps parameter. The default coefficients give an 8th order, linear phase low-pass filter. To read coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path. Polyphase multirate filtering is also supported.

**Kalman**

Output the state vector estimates of a Kalman filter using a one-step prediction algorithm.

**GAL**

A Gradient Adaptive Lattice filter.

**Goertzel**

Second-order recursive computation of the k-th coefficient of an N-point DFT using Goertzel’s algorithm.

**GGAL**

Ganged Gradient Adaptive Lattice filters.

**Hilbert**

Output the (approximate) Hilbert transform of the input signal. This primitive approximates the Hilbert transform by using an FIR filter, and is derived from the FIR primitive.

**IIR**

An Infinite Impulse Response (IIR) filter implemented in direct form II. The transfer function is of the form

\[ H(z) = G \frac{N(1/z)}{D(1/z)} \]

where \( N() \) and \( D() \) are polynomials. The parameter gain specifies \( G \), and the floating-point arrays numerator and denominator specify \( N() \) and \( D() \), respectively. Both arrays start with the constant terms of the polynomial and decrease in powers of \( z \) (increase in powers of \( 1/z \)). Note that the constant term of \( D \) is not omitted, as is common in other programs that assume it is always normalized to unity.

**IIRFix**

This is a fixed-point version of the IIR primitive. The coefficient precision, input precision, accumulation precision, and output precision can all be separately specified.

**Lattice**

An FIR lattice filter. The default reflection coefficients form the optimal predictor for a particular 4th-order AR random process. To read other reflection coefficients from a file, replace the default coefficients with `< fileName`, preferably specifying a complete path.

**phaseShift**

This module applies a phase shift to a signal according to the "shift" input. If the "shift" input value is time varying, then its slope deter-
mines the instantaneous frequency shift.

**RLattice**
A recursive (IIR) lattice filter. The default coefficients implement the synthesis filter for a particular 4th-order AR random process. To read reflection coefficients from a file, replace the default coefficients with `<fileName`, preferably specifying a complete path.

### 18.18.2 Adaptive Filters

**LMS**
An adaptive filter using the Least-Mean Square (LMS) adaptation algorithm. The initial coefficients are given by the `taps` parameter. The default initial coefficients give an 8th order, linear phase low-pass filter. To read default coefficients from a file, replace the default coefficients with `<fileName`, preferably specifying a complete path. This primitive, which is derived from FIR, supports decimation, but not interpolation.

**LMSCx**
Complex version of the LMS primitive.

**LMSCxTkPlot**
This primitive is just like the LMSCx primitive, but with an animated Tk display of the taps, plus associated controls.

**LMSLeak**
An LMS adaptive filter in which the step size is input (to the ”step” input) every iteration. In addition, the `mu` parameter specifies a leakage factor in the updates of the filter coefficients.

**LMSPlot**
This primitive is just like the LMS primitive, except that, in addition to the functions of LMS, it makes a plot of the tap coefficients. It can produce two types of plots: a plot of the final tap values or a plot that traces the time evolution of each tap value. The time evolution is obtained if the value of the parameter `trace` is YES.

**LMSTkPlot**
the primitive is just like the LMS primitive, but with an animated Tk display of the taps, plus associated controls.

**LMSOscDet**
This filter tries to lock onto the strongest sinusoidal component in the input signal, and outputs the current estimate of the cosine of the frequency of the strongest component and the error signal. It is a three-tap LMS filter whose first and third coefficients are fixed at one. The second coefficient is adapted. It is a normalized version of the Direct Adaptive Frequency Estimation Technique.

**LMSPlotCx**
Complex version of LMSPlot. Separate plots are generated for the magnitude and phase of the filter coefficients.

### 18.18.3 Block Filters

The next group of primitives perform ”block filtering”, which means that on each firing, they read a set of input particles all at once, process them, and produce a set of output particles. The number of particles in a set is specified by the blockSize parameter.
18.18 Signal processing (DSP) primitives

**BlockAllPole**
This primitive implements an all pole filter with the denominator coefficients of the transfer function externally supplied. For each set of coefficients, a block of input samples is processed, all in one firing. The transfer function is

\[ H(z) = \frac{1}{1 - D(z)} \]

where the coefficients of \( D(z) \) are externally supplied.

**BlockFIR**
This primitive implements an FIR filter with coefficients that are periodically updated from the outside. For each set of coefficients, a block of input samples is processed, all in one firing.

**BlockLattice**
A block forward lattice filter. It is identical to the Lattice primitive except that the reflection coefficients are updated each time the primitive fires by reading the "coefs" input. The order parameter indicates how many coefficient should be read. The blockSize parameter specifies how many data samples should be processed for each set of coefficients.

**BlockRLattice**
A block recursive (IIR) lattice filter. It is identical to the RLattice primitive, except that the reflection coefficients are updated each time the primitive fires by reading the "coefs" input. The order and blockSize parameters have the same interpretation as in the Block-Lattice primitive.

**blockPredictor**
A block predictor module used in speech processing.

**blockVocoder**
A block vocoder module.

### 18.18.4 Vector Quantization
Quantization is the heart of converting analog signals to digital signals. Traditional techniques are based on scalar coding which quantizes symbols, such as pixels in images, one by one. On the other hand, vector quantization can perform better by operating the quantization on groups of symbols instead of individual symbols.

**GLA**
Use the Generalized Lloyd Algorithm (GLA) to yield a code-book from input training vectors. Note that each input matrix will be viewed as a row vector in row by row. Each row of output matrix represents a codeword of the codebook.

**MRVQCoder**
Mean removed vector quantization coder.

**SGVQCodebk**
Jointly optimized codebook design for shape-gain vector quantization. Note that each input matrix will be viewed as a row vector in row by row. Each row of first output matrix represents a codeword of the shape codebook. Each element of the second output matrix represents a codeword of the gain codebook.

**SGVQCoder**
Shape-gain vector quantization encoder. Note that each input matrix will be viewed as a row vector in row by row.
VQCoder  
Full search vector quantization encoder. It consists in finding the index of the nearest neighbor in the given codebook corresponding to the input matrix. Note that each input matrix will first be viewed as a row vector in row by row, in order to find the nearest neighbor codeword in the codebook.

18.19 Spectral analysis

This group of primitives are concerned with various signal analysis algorithms.

autocorrelation  
Estimate a power spectrum using the autocorrelation method, a method that uses the Levinson-Durbin algorithm to compute linear predictor coefficients, and then uses these coefficients to construct an approximate maximum entropy power spectrum estimate.

blockFFT  
An overlap and add implementation of the FFT.

burg  
Estimate a power spectrum using Burg’s method, a method that computes linear predictor coefficients, and then uses them to construct a maximum entropy power spectrum estimate.

Burg  
This primitive uses Burg’s algorithm to estimate the linear predictor coefficients of an input random process. These coefficients are produced both in autoregressive form (on the “lp” output) and in lattice filter form (on the “refl” output). The “errPower” output is the power of the prediction error as a function of the predictor order. This primitive is used in the burg module.

DB  
Convert input to a decibel (dB) scale. Zero and negative values are assigned the value \( \min \) (default -100). The inputIsPower parameter should be set to YES if the input signal is a power measurement (vs. an amplitude measurement).

DTFT  
Compute the discrete-time Fourier transform (DTFT) at frequency points specified on the “omega” input.

FFTCx  
Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter order (default 8) is the log base 2 of the transform size. The parameter size (default 256) is the number of samples read (\( \leq 2^{\text{order}} \)). The parameter direction (default 1) is 1 for the forward, -1 for the inverse FFT.

GoertzelPower  
Second-order recursive computation of the power of the kth coefficient of an \( N \)-point DFT using Goertzel’s algorithm. This form is used in touch-tone decoding.

LevDur  
This primitive uses the Levinson-Durbin algorithm to compute the linear predictor coefficients of a random process, given its autocorrelation function as an input. These coefficients are produced both in autoregressive form (on the “lp” output) and in lattice filter form (on the
"refl" output). The "errPower" output is the power of the prediction error as a function of the predictor order.

**MUSIC**

This primitive is used to estimate the frequencies of some specified number of sinusoids in a signal. The output is the eigen spectrum of a signal, such that the locations of the peaks of the eigen spectrum correspond to the frequencies of the sinusoids in the signal. The input is the right singular vectors in the form generated by the SVD primitive. The MUSIC algorithm (multiple signal characterization) is used.

**periodogram**

Estimate a power spectrum using the periodogram method. This consists in computing the magnitude squared of the DFT of a set of observations of the signal. The FFT algorithm is used.

**SmithForm**

Decompose an integer matrix S into one of its Smith forms \( S = UDV \), where \( U \), \( D \), and \( V \) are simpler integer matrices. The Smith form decomposition for integer matrices is analogous to singular value decomposition for floating-point matrices.

**SVD**

Compute the singular-value decomposition of a Toeplitz data matrix \( A \) by decomposing \( A = UWV' \), where \( U \) and \( V \) are orthogonal matrices, and \( V' \) represents the transpose of \( V \). \( W \) is a diagonal matrix composed of the singular values of \( A \), and the columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

**Unwrap**

Unwraps a phase plot, removing discontinuities of magnitude 2. This primitive assumes that the phase never changes by more than in one sample period. It also assumes that the input is in the range \([-\pi, \pi]\).

**Window**

Generate standard window functions or periodic repetitions of standard window functions. The possible functions are Rectangle, Bartlett, Hanning, Hamming, Blackman, Steep-Blackman and Kaiser. One period of samples is produced on each firing. This primitive is also found in the signal sources library.

### 18.19.1 Miscellaneous signal processing blocks

**Autocor**

Estimate an autocorrelation function by averaging input samples. Both biased and unbiased estimates are supported.

**PattMatch**

This primitive accepts a template and a search window. The template is slid over the window one sample at a time, and cross correlations are calculated at each step. The cross-correlations are output on the "values" output. The "index" output is the value of the time-shift which gives the largest cross correlation. This index refers to a position on the search window beginning with 0 corresponding to the earliest arrived sample of the search window that is part of the best match with the template.
18 SDF Domain

18.20 Communication primitives

The limited set of communication primitives that have been developed are summarized below. Many of these are modules, and should be viewed as examples of systems that a user can create.

18.20.1 Sources and Pulse Shapers

- **bits**
  Produce "0" with probability probOfZero, else produce "1".

- **cosine.pal**
  Produce a cosine waveform whose energy is normalized with respect to Amplitude. It is used in simulations for binary frequency shift keying (BFSK) demonstrations. This module differs from the cosine primitive which computes the cosine of the input signal (see "Non-linear primitives" on page 5-13 for more information on the cosine primitive).

- **Hilbert**
  Output the approximate Hilbert transform of the input signal. This primitive approximates the Hilbert transform by using an FIR filter, and is derived from the FIR primitive. The Hilbert primitive is also in the signal processing library.

- **RaisedCosine**
  An FIR filter with a magnitude frequency response shaped like the standard raised cosine or square-root raised cosine used in digital communication. By default, the primitive upsamples by a factor of 16, so 16 outputs will be produced for each input unless the interpolation parameter is changed.

- **RaisedCosineCx**
  This module uses the RaisedCosine primitive to implement an FIR filter for complex inputs with a raised cosine or square-root raised cosine transfer function.

18.20.2 Transmitter Functions

- **NR2Zero**
  Binary to Nonreturn-to-Zero Signaling Converter

- **QAM4**
  Encode an input bit stream into a 4-QAM (or 4-PSK) complex symbol sequence.

- **QAM16**
  Encode an input bit stream into a 16-QAM complex symbol sequence.

- **Scrambler**
  Scramble the input bit sequence using a feedback shift register. The taps of the feedback shift register are given by the polynomial parameter, which should be a positive integer. The n-th bit of this integer indicates whether the n-th tap of the delay line is fed back. The low-order bit is called the 0-th bit, and should always be set. The next low-order bit indicates whether the output of the first delay should be fed back, etc. The default polynomial is an octal number defining the V.22bis scrambler.

- **Spread**
  Frame synchronized direct-sequence spreader.

- **xmit2fsk**
  Binary frequency shift keying (BFSK) transmitter.
18.20 Communication primitives

xmit2pam Simple 2-level pulse amplitude modulation (PAM) transmitter.
xmit4pam Simple 4-level pulse amplitude modulation (PAM) transmitter.
xmit2psk Binary 2-level phase shift keying (BPSK) Modulator.
xmitspread Direct-sequence spreader (i.e., spread-spectrum transmitter).

18.20.3 Receiver functions

DeScrambler Descramble the input bit sequence using a feedback shift register. The taps of the feedback shift register are given by the polynomial parameter. This is a self-synchronizing descrambler that will exactly reverse the operation of the Scrambler primitive if the polynomials are the same. The low-order bit of the polynomial should always be set.

DeSpreader Frame synchronized direct-sequence despreader.

hilbertSplit This module implements a phase splitter, in which the real-valued input signal is converted to an (approximate) analytic signal. The signal is filtered by the Hilbert block to generate the imaginary part of the output, while the real part is obtained by creating a matching delay.

qam4Slicer This module implements a slicer (decision device) for a 4-QAM (or equivalently, 4-PSK) signal. The output decision is a complex number with +1 or -1 for each of the real or imaginary parts.

qam16Slicer This module implements a slicer (decision device) for a 16-QAM complex signal. The output decision is a complex number with +1, -1, +3, or -3 for each of the real or imaginary parts.

qam16Decode A 16-QAM decoder similar to the CCITT V22.bis standard. The quadrant is differentially de-encoded.

phaseShift Shifts the phase of the input signal on the input by the shift value on the shift input. The phase shifting is implemented by filtering the input signal with a complex FIR filter to convert it into an analytic signal and the complex result is modulated by a complex exponential. If the shift value is time varying, then its slope determines the instantaneous frequency shift.

rec2fsk Binary frequency shift keying (BFSK) Receiver.

rec2pam Simple 2-level pulse amplitude modulation (PAM) receiver.

rec4pam Simple 4-level pulse amplitude modulation (PAM) receiver.

rec2psk Binary pulse shift keying (BPSK) Demodulator.

recspread Direct sequence receiver.

18.20.4 Channel Models

AWGNchannel Model an additive Gaussian white noise channel with optional linear distortion.
**basebandEquivChannel**  
Baseband equivalent channel.

**freqPhase**  
Impose frequency offset and/or phase jitter on a signal in order to model channels, such as telephone channels, that suffer these impairments.

**noiseChannel**  
A simple channel model with additive Gaussian white noise.

**nonLinearDistortion**  
Generate second and third harmonic distortion by squaring and cubing the signal, and adding the results in controlled proportion to the original signal.

**telephoneChannel**  
Simulate impairments commonly found on a telephone channel, including additive Gaussian noise, linear and nonlinear distortion, frequency offset, and phase jitter.

## 18.21 Telecomm

This library contains primitives and modules used for processing signals in telecommunication networks.

### 18.21.1 Conversion, Signal Sources, and Signal Tests

**MuLaw**  
Transform the input using a logarithmic mapping if the `compress` parameter is true. In telephony, applying the $\mu$-law to eight-bit sampled data is called companding, and it is used to quantize the dynamic range of speech more accurately. The transformation is defined in terms of the non-negative integer parameter $mu$:

$$output = \frac{\log(1 + mu \cdot |input|)}{\log(1 + mu)}$$

**DTMFGenerator**  
Generate a dual-tone modulated-frequency (DTMF) signal by adding a low frequency and a high frequency sinusoid together. DTMF tones only consist of first harmonics. The default parameters generate a ”1” on a touch tone telephone.

**PostTest**  
Return whether or not a valid dual-tone modulated-frequency has been correctly detected based on the last three detection results.

**ToneStrength**  
Decision circuit for dual-tone modulated-frequency (DTMF) decoding. It returns true if $A_{max}$ is greater than or equal to $A_i$ for $i = \{1, 2, 3, 4\}$ such that $i$ does not equal index.

### 18.21.2 Touch tone Decoders

**DTMFDecoder**  
Dual-tone modulated-frequency (DTMF) decoder based on post-processing of a bank of Goertzel discrete Fourier transform filters. This module decodes touch tones generated by a telephone.
DTMFDecoderBank  Implement one of the banks for detecting dual-tone frequency-modulated (DTMF) touch tones. Touch tones are generated by adding a low frequency and a high frequency sinusoid together. The module is used to detect either the low or high frequency component, depending on the parameter settings. This algorithm examines the magnitude of the expected frequency components and their second harmonics. DTMF tones do not have second harmonics, so if they are present, then the input is likely speech and not touch tones. The valid output is true if the input is probably a touch tone. The default parameters are used to detect the low frequency tones.

GoertzelDetector  Detect the energy of the first and second harmonic using a pair of Goertzel filters.

lmsDTMFDecoderBank  Dual-tone modulated frequency detection based on the post-processing of the output of two LMS algorithms in cascade. These two algorithms are used to detect the two strongest frequencies present in the signal.

lmsDualTone  Detect the location of the two strongest harmonic components in the input signal for every input sample using the normalize direct frequency estimation technique, which is based on the LMS algorithm. This module is used in touch tone detection.

lmsDTMFDecoder  Least-mean squares dual-tone modulated-frequency decoder. Dual-tone modulated frequency detection based on the post-processing of the output of two LMS algorithms in cascade. These two algorithms are used to detect the two strongest frequencies present in the signal.

18.21.3 Channel Models

For more complete descriptions, see the channel models for the communications primitives given on page 5-36.

AWGN  Simulate a channel with additive Gaussian noise.
basebandEquivChannel  Baseband equivalent channel.
freqPhase  Impose frequency offset and/or phase jitter on a signal in order to model channels, such as telephone channels, that suffer these impairments.
noiseChannel  A simple channel model with additive Gaussian white noise.
nonLinearDistortion  Generate second and third harmonic distortion by squaring and cubing the signal, and adding the results in controlled proportion to the original signal.
TelephoneChannel  Telephone channel simulator with Gaussian noise and nonlinear distortion.
18.21.4 PCM and ADPCM

ADPCMCoder  Implement adaptive differential pulse code modulation using an LMS primitive. Both the quantized and unquantized prediction-error signals are available as outputs.

ADPCMDecoder  Decode the quantized prediction error signal produced by the ADPCMCoder module.

ADPCMFromBits  Convert a bit stream encoded with the ADPCMTobits module back to floating-point values. The 4 low-order bits of the input integer are changed to 1 of 16 floating-point values scaled by range.

ADPCMTobits  Convert the quantized prediction error of the ADPCMCoder module into a bit stream. The quantized prediction error has 16 possible levels, so this module produces 4 bits in each output sample.

PCMBitCoder  64kps PCM encoder (CCITT Recommendation G.711).

PCMBitDecoder  64kps PCM encoder (CCITT Recommendation G.711).

18.22 Spatial Array Processing

The spatial array processing primitives given here support a single demonstration named RadarChainProcessing developed by Karim Khiar from Thomson CSF. The radar simulation, though five-dimensional, is implemented using SDF, which is a one-dimensional data flow model.

18.22.1 Data Models

RadarAntenna  Generate a specified number of Doppler filter outputs. This module consists of a cascade of a network of antennas, a bank of matched filters, a bank of windows, and a Doppler filter. The bank of matched filters convolves the antenna outputs with a filter matched to a complex pulse train.

RadarTargets  Model the observed data as the addition of the receive signal plus sensor noise. The received signal consists of a summation of the emissions of all of the targets.

GenTarget  Model the reception of signals by one sensor. A complex pulse train is delayed and then multiplied by a complex exponential.

RectCx  Generate a rectangular pulse of width width (default 240). If period is greater than zero, then the pulse is repeated with the given period.

18.22.2 Sensor and Antenna Models

SubAntenna  Models a sub-antenna. It multiplies the input by a complex exponential.
18.23 Image Processing Primitives

sensor
Compute the excitation of a plane wave arriving at a sensor at the given position with the arrival angle specified as an input. Position (0,0) is assumed to receive phase zero for any angle of arrival.

ThermalNoise
Generate thermal noise as a complex noise process whose real and imaginary components are identically independently distributed Gaussian random processes.

Psi
Model sub-antenna excitation.

SpheToCart
Compute the inner product of two vectors, one given by a magnitude and two angles in spherical components, the other given by three Cartesian components.

18.22.3 Doppler Effects

PulseComp
This module generates any number of targets and performs pulse compression. It uses the original chirp to perform the pulse compression. This output represents the output of the radar processing along the range bin axis. The y-axis represents the target magnitude on a linear, logarithmic scale.

OneDoppler
Generate one Doppler output. This module performs an antenna to pulse multi-projection transformation followed by a decimator.

18.22.4 Beamforming Methods

steering
Multiply a sensor signal by a window sample and apply a steering correction.

18.23 Image Processing Primitives

The image processing primitives were originally written by Paul Haskell. The image processing infrastructure was rewritten by Bilung Lee to use matrices as the underlying image representation.

18.23.1 Displaying images

DisplayImage
Accept a black-and-white input gray image represented by a float matrix and generate output in PGM (portable gray-map) format. Send the output to a user-specified command (by default, xv is used). The user can set the root filename of the displayed image (which will probably be printed in the image display window title bar) and can choose whether or not the image file is saved or deleted. The image frame number is appended to the root filename in order to form the complete filename of the displayed image.

DisplayRGB
This is similar to DisplayImage, but accepts three color images (Red, Green, and Blue) from three input float matrix and generates
a .ppm (portable pixmap) format color image file. The image file is displayed using a user-specified command (by default, the internal viewer is used).

DisplayVideo

Accept a stream of black-and-white images from input float matrix, save the images to files, and display the resulting files as a moving video sequence. This primitive requires that programs from the Utah Raster Toolkit (URT) be in your path. Although this toolkit is not included with MLDesigner, it is available for free. The user can set the root filename of the displayed images (which probably will be printed in the display window title bar) with the ImageName parameter. If no filename is set, a default will be chosen.

The Save parameter can be set to YES or NO to choose whether the created image files should be saved or deleted. Each images frame number is appended to the root filename in order to form the images complete filename.

The ByFields parameter can be set to either YES or NO to choose whether the input images should be treated as interlaced fields that make up a frame or as entire frames. If the inputs are fields, then the first field should contain frame lines 1, 3, 5, etc. and the second field should contain lines 0, 2, 4, 6, etc.

videodpy

Display an image sequence in an X window. This is simply the SDFDisplayVideo primitive encapsulated in a module so that it can be easily used in other domains.

18.23.2 Reading images

ReadImage

Read a sequence of PGM-format images from different files and send them out in a float matrix. If present, the character # in the fileName parameter is replaced with the frame number to be read next. For example, if the frameId parameter is set to 2 and if the fileName parameter is dir.#/pic# then the file that is read and output is dir.2/pic2.

ReadRGB

Read a PPM-format image from a file and send it out in three different images; a Red, Green, and Blue image. Each image is represented in a float matrix. The same mechanism for reading successive frames as in ReadImage is supported.

videosrc

Read in an image from a specified file. This is simply the SDFReadImage primitive encapsulated in a module so that it can be easily used in other domains.

SunVideo

Reads frames from the SunVideo card and outputs them as 3 matrices: one for Y, U and V components. This primitive is implemented for future uses and is not yet fully implemented in the Linux version of the MLDesigner, and therefore has no demos.
18.23.3 Color conversions

**RGBToYUV**
Read three float matrices that describe a color image in RGB format and output three float matrices that describe an image in YUV format. No downsampling is done on the U and V signals.

**YUVToRGB**
Read three float matrices that describe a color image in YUV format and output three float matrices that describe an image in RGB format.

18.23.4 Image and video coding

**DCTImage**
Take a float matrix input particle, compute the discrete cosine transform (DCT), and output a float matrix.

**DCTImageInv**
Take a float matrix input, compute the inverse discrete cosine transform (DCT), and output a float matrix.

**DCTImgCde**
Take a float matrix which represents a DCT image, insert "start of block" markers, run-length encode it, and output the modified image. For the run-length encoding, all values with absolute value less than the \textit{Thresh} parameter are set to 0.0, to help improve compression. Run lengths are coded with a "start of run" symbol and then an (integer) run-length.

The \textit{HiPri} parameter determines the number of DCT coefficients per block are sent to "hiport", the high-priority output. The remainder of the coefficients are sent to "loport", the low-priority output.

**InvDCTImgCde**
Read two coded float matrices (one high priority and one low priority), invert the run-length encoding, and output the resulting float matrix. Protection is built in to avoid crashing even if some of the coded input data is affected by loss.

**DPCMImage**
Implement differential pulse code modulation of an image. If the "past" input is not a float matrix or has size 0, pass the "input" directly to the "output". Otherwise, subtract the "past" from the "input" (with leakage factor \textit{alpha}) and send the result to "output".

**DPCMImageInv**
This primitive inverts differential pulse code modulation of an image. If the "past" input is not a float matrix or has size 0, pass the "diff" directly to the "output". Otherwise, add the "past" to the "diff" (with leakage factor \textit{alpha}) and send the result to "output".

**MotionCmp**
If the "past" input is not a float matrix (e.g. dummyMessage), copy the "input" image unchanged to the "diffOut" output and send a null field (zero size matrix) of motion vectors to "mvHorzOut" and "mvVertOut" outputs. This should usually happen only on the first firing of the primitive.

For all other inputs, perform motion compensation and write the difference frames and motion vector frames to the corresponding outputs. This primitive can be used as a base class to implement slightly different motion compensation algorithms. For example, synchronization
techniques can be added or reduced-search motion compensation can be performed.

**MotionCmpInv**

For NULL inputs (zero size matrices) on "mvHorzIn" and/or "mvVertIn", copy the "diffIn" input unchanged to "output" and discard the "pastIn" input. (A NULL input usually indicates the first frame of a sequence.) For non-NULL "mvHorzIn" and "mvVertIn" inputs, perform inverse motion compensation and write the result to "output".

**RunLenImg**

Accept a float matrix and run-length encode it. All values closer than Thresh to meanVal are set to meanVal to help improve compression. Run lengths are coded with a start symbol of meanVal and then a run-length between 1 and 255. Runs longer than 255 must be coded in separate pieces.

**RunLenImgInv**

Accept a float matrix and inverse run-length encode it.

**ZigZagImage**

Zig-zag scan a float matrix and output the result. This is useful before quantization.

**ZigZagImageInv**

Inverse zig-zag scan a float matrix.

**codef**

This module encodes a sequence of images using motion compensation, a discrete-cosine transform, quantization, and run-length encoding. The outputs are split into high priority and low priority, where corruption of the low priority data will impact the image less.

**codei**

This module inverts the encoding of the codef block, and outputs a reconstructed image sequence.

**videofwd**

This module is obsolete and will probably disappear in the next release.

**videoinv**

This module is obsolete and will probably disappear in the next release.

### 18.23.5 Miscellaneous image blocks

**AddMotionVecs**

Over each block in the input image, superimpose an arrow indicating the size and direction of the corresponding motion vector.

**Contrast**

Enhance the contrast in the input image by histogram modification. Input image should be in an integer matrix. The possible contrast type are Uniform (default) and Hyperbolic.

**Dither**

Do digital halftoning (dither) of input image for monochrome printing. Input image should be in a float matrix. The possible dither methods are Err-Diffusion (default), Clustered, Dispersed, and Own. If you specify Own, then you can use your own dither mask.

**EdgeDetect**

Detect edges in the input image. Input image should be in a float matrix. The possible detectors are Sobel (default), Roberts, Prewitt, and Frei-Chen.
18.24 Neural Networks

The neural network primitives demonstrate logic functions using classical artificial neurons and McCulloch-Pitts neuron. These primitives were written by Biao Lu (The University of Texas at Austin), Brian L. Evans (The University of Texas at Austin).

**MedianImage**

Accept an input gray image represented by a float matrix, median-filter the image, and send the result to the output. Filter widths of 1, 3, 5 work well. Any length longer than 5 will take a long time to run. Median filtering is useful for removing impulse-type noise from images. It also smooths out textures, so it is a useful pre-processing step before edge detection. It removes inter-field flicker quite well when displaying single frames from a moving sequence.

**RankImage**

Accept an input gray image represented by a float matrix, rank filter the image, and send the result to the output. A common example of a rank filter is the median filter, e.g. `MedianImage`, which is derived from this primitive. Pixels at the image boundaries are copied and not rank filtered.

**MPNeuron**

This is a McCulloch-Pitts neuron. The activation of this neuron is binary. That is, at any time step, the neuron either fires, or does not fire.

**Neuron**

This neuron will output the sum of the weighted inputs, as a floating value.

**ConstThreshold**

Output a constant signal with value given by the "level" parameter (default 0.0)

**Binary**

Binary threshold of the input.

**Sigmoid**

Compute the Sigmoid function, defined as $\frac{1}{1 + \exp(-r \cdot \text{input})}$ where $r$ is the learning rate.

**MPandBinary**

The fact that the McCulloch-Pitts neuron is a digital device makes this neuron well-suited to the representation of a two-valued logic, such as AND, OR, and NAND.

**MPxorBinary**

This example shows that a network of McCulloch-Pitts neurons has the power of the finite state automaton known as a Turing machine.

**xorBinary**

XOR function can be implemented by a three-layer neural network which consists of an input layer, a hidden layer and an output layer. A binary activation function is used.

**xorSigmoid**

XOR function can be implemented by a three-layer neural network which consists of an input layer, a hidden layer and an output layer. A sigmoid activation function is used.
18 SDF Domain

18.25 Tcl primitives

Most of the primitives that interface to Tcl appear in libraries that reflect their function. For instance, all the primitives beginning with Tk in the sinks library are actually Tcl primitives derived from TclScript. This is the most generic Tcl primitive with no function on its own. It must have a Tcl script associated with it to make it useful. Please refer to ch. 16 to learn about writing such scripts.

18.25.1 Interactive Graphics Facilities

These primitives are multiple configurations of only six primitives. These primitives all use the Tk toolkit associated with the Tcl language to create interactive, animated displays on the screen.

- **TkPlot**
  Plot "Y" input(s) vs. time with dynamic updating. Two styles are currently supported: **dot** causes individual points to be plotted, whereas **connect** causes connected lines to be plotted. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.

- **TkXYPlot**
  Plot "Y" input(s) vs. "X" input(s) with dynamic updating. Two styles are currently supported: **dot** causes points to be plotted, whereas **connect** causes connected lines to be plotted. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.

- **TkShowValues**
  Display the values of the inputs in textual form. The **print** method of the input particles is used, so any data type can be handled, although the space allocated on the screen may need to be adjusted.

- **TkBarGraph**
  Dynamically display the value of any number of input signals in bar-chart form. The first 12 input signals will be assigned distinct colors. After that, the colors are repeated. The colors can be controlled using X resources.

- **TkMeter**
  Dynamically display the value of any number of input signals on a set of bar meters.

- **TkShowBooleans**
  Display input Booleans using color to highlight their value.

Programmable Interactive Sinks:

- **TclScript**
  Invoke a Tcl script that can optionally define a procedure that is invoked every time the primitive fires. That procedure can read the primitive’s inputs and update the value of the outputs.

- **MatlabCx.M**
  Evaluate a Matlab function if inputs are given or evaluate a Matlab command if no inputs are given.
18.26 Overview of SDF Demos

Sound:

Play

Play an input stream on the workstation speaker. The gain parameter (default 1.0) multiplies the input stream before it is \(\mu\)-law compressed and written. The inputs should be in the range of -32000.0 to 32000.0. The file is played at a fixed sampling rate of 8000 samples per second. When the wrapup method is called, a file of 8-bit \(\mu\)-law samples is handed to a program named ptplay which plays the file.

Textual Display:

Printer

Print out one sample from each input port per line. The fileName parameter specifies the file to be written; the special names stdout and cout which specify the standard output stream, as well as stderr and cerr which specify the standard error stream, are also supported.

TkText

Display the values of the inputs in a separate window, keeping a specified number of past values in view. The print method of the input particles is used, so any data type can be handled.

Other:

WriteVar

Write the value of the input to a double-precision floating-point variable in shared memory. Use the ReadVar primitive to read values from the shared memory.

NOTE: This primitive may produce unpredictable results, since the results will depend on the precedences in the block diagram in which it appears, as well as the scheduler (target) used.

18.26 Overview of SDF Demos

A rather large number of SDF demonstrations have been developed. These can serve as valuable illustrations of the possibilities. Almost every primitive is illustrated in the demos. Because of the large number, the demos are organized into a set of libraries. Certain demos may appear in more than one library.

18.27 Basic demos

These demos illustrate the use of certain primitives without necessarily performing functions that are sophisticated.

butterfly

Use sines and cosines to compute a curve known as the butterfly curve, invented by T. Fay. The curve is plotted in polar form.

chaoticNoise

Chaotic Markov map example with a nonlinear feedback loop.

comparison

Compare two sinusoidal signals using the Test primitive.
complexExponential Generate and plot a complex exponential.
delayTest Illustrates the use of initializable delays.
lmsFreqDetect Illustrate the use of the LMS algorithm to estimate the dominant sinusoidal frequency in the input signal.
freqPhaseOffset Impose frequency jitter and phase offset on a sinusoid using the freqPhase SDF block.
gaussian Generate a Gaussian white noise signal, and plot its histogram and estimated autocorrelation.
ingegrator Demonstrate the features of the integrator primitive, such as limiting, leakage, and resetting.
Modulo Demonstrate modulus computation for float and integer data types.
muxDeMux Demonstrate the Mux and DeMux primitives, which perform multiplexing and demultiplexing. Contrast with the scramble demo below.
quantize Demonstrate the use of the Quantizer primitive.
scramble This system rearranges the order of samples of signal using the Commutator and Distributor primitives. Note that because these are multirate primitives, one iteration involves more than one sample. Contrast with the muxDeMux demo above.
sinMod Modulate a sinusoid by multiplying by another sinusoid.
tbus Illustrate the bus facility in MLDesigner, in which multiple signals are combined onto a single graphical connection.

18.28 Multirate demos

These demos illustrate synchronous data flow principles as applied to multirate signal processing problems.

analytic Use a FIRCx primitive filter to reduce the sample rate of a sinusoid by a factor of 8/5, and at the same time produce a complex approximately analytic signal (one that has no negative frequency components).
broken Give an example of an inconsistent SDF system. It fails to run, generating an error message instead.
downSample Convert from the digital audio tape sampling rate (48 kHz) to the compact disc sampling rate (44.1 kHz). The conversion is performed in multiple stages for better performance.
filterBank Implement an eight-level perfect reconstruction one-dimensional filter bank based on the biorthogonal wavelet decomposition.
filterBank-NonUniform Implement a simple split of the frequency domain into two non-uniform frequency bands.
Use an FIR filter to upsample by a factor of 8 and linearly interpolate between samples.

Upsample a sinusoidal signal by a ratio of $5/2$ using a polyphase low-pass interpolating FIR filter.

Convert from the compact disc sampling rate (44.1 kHz) to the digital audio tape sampling rate (48 kHz). The conversion is performed in multiple stages for better performance.

This library contains some examples of digital communication systems and channel simulators.

A 16-QAM signal is sent through a baseband equivalent channel that simulates the following impairments: frequency offset, phase jitter and white Gaussian noise.

Dual-Tone Modulated Frequency Demo. Generate touch tones and decode the based on the Goertzel Algorithm.

Plot an eye diagram for a binary antipodal signal with a raised-cosine pulse shape and user controlled noise.

Dual-Tone Modulated Frequency Demo. Generate touch tones and decode them based on the LMS Algorithm.

Illustrate the effect on speech of a zero-substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability. Note that this demo requires audio capability and will probably only work on Sun workstations.

Illustrate the effect on speech of a previous cell substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability. Note that this demo requires audio capability.

Baseband model of a 16-QAM modem.

Generate a pseudo-random sequence of zeros and ones using a maximal-length shift register and test its randomness by estimating its autocorrelation.

Generate raised cosine and square-root raised cosine pulses and demonstrate matched filtering with the square-root raised cosine pulse.

Bit Error determination through simulation at various noise levels.

Simple 2-level PAM communication system (matched filtering at the receiver).

Simple 4-level PAM communication system (no filtering at the receiver).
18.29.1 Older communications demos

qam

Produce a 16-point quadrature amplitude modulated (QAM) signal and displays the eye diagram for the in-phase part, the constellation, and the modulated transmitted signal.

QAM4withDFE

This is a model of a digital communication system that uses quadrature amplitude modulation (QAM) and a fractionally spaced decision feedback equalizer.

codeDecode

Encode and decode a 16-QAM signal using differential encoding for the quadrant and Gray coding for the point within the quadrant.

plldemo

Simulate a fourth-power optical phase-locked loop with laser phase noise and additive Gaussian white noise operating on a complex baseband envelope model of the signal.

telephoneChannelTest

Assuming a sampling rate of 8 kHz, a sinusoid at 500 Hz is transmitted through a simulation of a telephone channel with additive Gaussian noise, nonlinear distortion, and phase jitter.

18.30 Digital signal processing demos

A fairly large number of signal processing applications are represented in this library.

adaptFilter

An LMS adaptive filter converges so that its transfer function matches that of a fixed FIR filter.

allPole

Two realizations of an all-pole filter are shown to be equivalent. One uses an FIR filter in a feedback path, the other uses the BlockAllPole primitive.

animatedLMS

An LMS adaptive filter is configured as in the adaptFilter demo, but this time the filter taps are displayed as they adapt.

animatedLMSCx

A complex LMS adaptive filter is configured as in the adaptFilter demo, but in addition, user-controlled noise is added to the feedback loop using an on-screen slider to control the amount of noise. The filter taps are displayed as they adapt.

cep

Given the coefficients of any polynomial, this demo uses the cepstrum to find a minimum-phase polynomial. Thus, given the coefficients of the denominator polynomial of an unstable filter, this demo will compute the coefficients of a stable denominator polynomial that has the same magnitude frequency response.

chaos

This is a simple demonstration of chaos, in which the phase-space plot of the famous Henon map is given.

convolve

Convolve two rectangular pulses in order to demonstrate the Convolve primitive.

dft

Compute a discrete Fourier transform of a finite signal using the FFT
primitive. The magnitude and phase (unwrapped) are plotted.

doppler A sine wave is subjected to four successive amounts of doppler shift. The doppler shift is accomplished by the phaseShift module, which forms an analytic signal (using a Hilbert transform) that modulates a complex exponential.

dtft Demonstrate the DTFT primitive, showing how it is different from the FFTCx primitive. Specifically, the range, number, and spacing of frequency samples is arbitrary.

freqsample This system designs FIR filters using the frequency sampling method. Samples of the frequency response are converted into FIR filter coefficients.

iirDemo Two equivalent implementations of IIR filtering.

lattice Demonstrate the use of lattice filters to synthesize an auto-regressive (AR) random process.

latticeDesign Use of Levinson-Durbin algorithm to design a lattice filter with a specified transfer function.

levinsonDurbin Use the Levinson-Durbin algorithm to estimate the parameters of an AR process.

linearPrediction Perform linear prediction on a test signal consisting of three sinusoids in colored, Gaussian noise. Two mechanisms (Burg’s algorithm and an LMS adaptive filter) for linear prediction are compared.

overlapAddFFT Convolution is implemented in the frequency domain using overlap and add.

phasedArray Simulate a plane wave approaching a phased array with four sensors. The plane wave approaches from angles starting from head on and slowly rotating 360 degrees. The response of the antenna is plotted as a function of direction of arrival in polar form.

powerSpectrum Compare three methods for estimating a power spectrum of a signal with three sinusoids plus colored noise. The three methods are the periodogram method, the autocorrelation method, and Burg’s method.

timeVarSpec A time-varying spectrum is computed using the autocorrelation method and displayed using a waterfall plot.

window Generate and display four window functions and the magnitude of their Fourier transforms. The windows displayed are the Hanning, Hamming, Blackman, and steep Blackman.

18.31 Sound demos

The demos in this library assume that a program called ptplay is in your path, and that it accepts data of an appropriate format and will play it over the computer sound card at an 8 kHz sample
The samples are written into a file before they are played. Since a large number of samples must be generated, these demos can take some time to run.

**chirpplay**  Chirp generator.

**fmplay**  Sound generator using FM modulation.

**speech**  Read a speech signal from a file, and encode it at two bits per sample using adaptive differential pulse code modulation with a feedback-around-quantizer structure. The signal is then reconstructed from the quantized data. The original and reconstructed speech are played.

**KSchord**  Simulation of plucked string sounds using the Karplus-Strong algorithm.

**vox**  Coarticulation with an Adaptive Vocoder. The resulting FM synthesized sound is played.

**blockVox**  A block processed version of the vox demo.

**lossySpeech**  Illustrate the effect on speech of a zero-substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability. This demo also appears in the basic demos library.

**lossySpeechPrevCell**  Illustrate the effect on speech of a previous cell substitution policy in a network (such as ATM) with 48 byte packets and a variable loss probability.

**perfectReconstruction**  Eight-channel perfect reconstruction one-dimensional analysis/synthesis filterbank. The incoming speech signal is split into eight adjacent frequency bins and then reconstructed. The original and reconstructed speech are played.

**subbandcoding**  Four channel sub-band speech coding with APCM at 16kps.

### 18.32 Image processing demos

The demos in this library read images from files on the hard disk, process them, and then display them. Some of the demos process short sequences of images, thus illustrating video processing in MLDesigner. They all use the image classes described in ’’Image processing primitives’’. The set of demos in this library does not reflect the richness of possibilities. See the DE domain for more image and video signal processing applications in the context of packet-switched network simulations. The video demos require that the Utah Raster Toolkit be installed and available in the user’s path.

**BlendImage**  Combine two images and display the result.

**bwDither**  Demonstrate four different forms of black and white dithering: error diffusion, clustered dither, dispersed dither, and use custom mask.

**contrastEnhance**  Contrast enhancement by histogram modification.
18.33 Vector Quantization demonstrations

The Vector demos perform vector quantization procedures on images.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fullVQCodebk</td>
<td>Generate a codebook for full search vector quantization.</td>
</tr>
<tr>
<td>fullVQ</td>
<td>Full search vector quantization using codebook generated by fullVQCodebk.</td>
</tr>
<tr>
<td>SGVQCodebk</td>
<td>Generate codebooks for shape-gain vector quantization.</td>
</tr>
<tr>
<td>SGVQ</td>
<td>Shape-gain vector quantization using codebook from SGVQ-Codebk.</td>
</tr>
<tr>
<td>MRVQCodeBk</td>
<td>Generate codebooks for mean-removed vector quantization using independent quantizer structure.</td>
</tr>
<tr>
<td>MRVQmeanCB</td>
<td>Generate codebook for mean-removed vector quantization.</td>
</tr>
<tr>
<td>MRVQshapeCB</td>
<td>Generate the shape codebook for mean-removed quantization using alternative structure. This system uses the codebook generated by MRVQmeanCB</td>
</tr>
<tr>
<td>MRVQ</td>
<td>Mean-removed vector quantization</td>
</tr>
</tbody>
</table>

18.34 Fix demos

These demos illustrate the use of fixed-point primitives in the SDF domain. These primitives are used to model hardware implementations with finite precision.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixConversion</td>
<td>Illustration of the different masking options available.</td>
</tr>
<tr>
<td>fixFIR</td>
<td>Effect of filter tap precision on the frequency response.</td>
</tr>
<tr>
<td>fixIIRdf</td>
<td>Comparison of a fourth-order direct-form IIR filter implemented with floating-point arithmetic and a similar filter implemented with fixed-point arithmetic.</td>
</tr>
<tr>
<td>fixMpyTest</td>
<td>Testing of fixed-point multiplication over a range of numbers by com-</td>
</tr>
</tbody>
</table>
comparison against floating-point multiplication. The results should be the same.

### 18.35 Tcl/Tk demos

These demos allow interaction with the simulation. The interactivity is provided by the Tcl scripting language controlling the Tk graphics toolkit. Tcl is integrated throughout MLDesigner. Tk has been integrated into the graphical user interfaces for MLDesigner, but not in the ptcl textual interpreter. Therefore, these primitives do not work in ptcl.

- **animatedLMS**: See "Digital signal processing demos".
- **animatedLMSCx**: See "Digital signal processing demos".
- **buttons**: Demonstrate TkButtons.
- **phasedArray**: Demonstrate TkSlider by creating a vertical array of radar sensors that can be move in the horizontal plane. Note that small movements of the sensors radically change the polar gain plot. This simulation demonstrates the importance of sensor calibration to performance of the sensor array.
- **sinWaves**: Demonstrate TkBarGraph by generating and displaying a complex exponential.
- **tclScript**: Demonstrate TclScript by generating two interactive X windows each with a circle that move in the playing field.
- **tkMeter**: Demonstrate TkMeter by creating three bar meters. The first oscillates sinusoidally. The second displays a random number between zero and one. The third displays a random walk.
- **tkShowValues**: Demonstrate TkShowValues and TkText by displaying the ASCII form of two ramp sequences.
- **xyplot**: Demonstrate the dynamic plotting capabilities of the xyplot primitive.

### 18.36 Matrix demos

These systems demonstrate the use of matrix particles in MLDesigner. Matrices are also used in the SDF domain to represent images. See "Image and video processing demos". The demonstrations below are primarily to test matrix operations.

- **MatrixTest1**: Demonstrate the use of the Matrix primitives that have one input. These include the operations inverse, transpose, and multiply by a scalar gain for all matrix types. Also conjugate and Hermitian transpose are available for the complex matrix type.
- **MatrixTest2**: Demonstrate the use of some simple Matrix primitives with two inputs. These include multiply, add, and subtract.
18.37 MATLAB Demos

MatrixTest3: Demonstrate the use of the Matrix conversion primitives. These convert between the scalar particles and the matrix particles as well as between the various matrix types.

initDelays: Illustrate the use of initializable delays with the matrix class.

Kalman_M: Compare the convergence properties of a Kalman filter to those of an LMS filter when addressing the problem of adaptive equalization of a process in noise.

SVD_MUSIC_1: Show the use of singular-value decomposition (SVD) and the Multiple-Signal Characterization (MUSIC) algorithm to identify the frequency of a single sinusoid in a signal that has two different signal to noise ratios.

SVD_MUSIC_2: Demonstrate the use of the Multiple-Signal Characterization (MUSIC) algorithm to identify three sinusoids in noise that have frequencies very close to each other.

18.37 MATLAB Demos

These demos demonstrate the use of the MATLAB primitives. The MATLAB primitives convert input values into MATLAB matrices, apply a sequence of MATLAB commands to the matrices, and output the result as MLDesigner matrices. The filterPrototype demonstration shows how to use MATLAB to compute parameters of primitives (see sec. 18.17 for more details).

matlab_hilb: This demo uses MATLAB as a signal source to produce a Hilbert matrix. The Hilbert matrix is an ill-conditioned matrix used to test the robustness of numerical linear algebra routines. The matrix element (i,j) has the value of 1/(i + j - 1). The matrix values appear similar to the coefficients of a discrete Hilbert transformer.

matlab_eig: This demo shows the use of MATLAB to perform eigen decomposition of a 2 × 2 Hermitian symmetric complex matrix. A matrix of eigenvectors and a matrix of eigenvalues are produced. The eigenvalues are real because the input matrix is Hermitian symmetric.

sombrero: This demo is an entire system composed of a cascade of four MATLAB primitives. The MATLAB primitives are used a signal source and a signal sink. The overall system generates and plots a mathematical model of a two-dimensional sinc function that resembles a sombrero.

filterPrototype: This system uses a half-band low-pass filter prototype for the low-pass and high-pass filters. All parameters are computed using MATLAB.
Chapter 19

DDF Domain

19.1 Introduction

The dynamic data flow (DDF) domain in MLDesigner is a superset of the synchronous data flow (SDF) and Boolean data flow (BDF) domains. In the SDF domain, a primitive consumes and produces a fixed number of particles per invocation (or "firing"). This static information (the number of particles produced or consumed for each primitive) makes possible compile-time scheduling. In the BDF domain, some actors with data-dependent production or consumption are allowed. The BDF schedulers attempt to construct a compile-time schedule; however, they may fail to do so and fall back on a DDF scheduler. In the DDF domain, the schedulers make no attempt to construct a compile-time schedule. For this reason, there are few constraints on the production and consumption behavior of primitives in this domain.

In DDF, a run-time scheduler detects which primitives are runnable and fires them one by one until no primitive is runnable (the system is deadlocked), or until a specified stopping condition has been reached. A primitive is runnable if it has enough data on its inputs to satisfy its requirements. Thus, the only constraint on DDF primitives is that they must specify on each firing how much data they require on each input to be fired again later.

In practice, primitives in the DDF domain are written in a slightly simpler way. They are either SDF primitives, in which case the number of particles required at each input is a constant, or they are dynamic, in which case they always alert the scheduler before finishing a firing that to be refired they expect some specific number of particles on one particular input. The input that a primitive is waiting for data on is called the waitPort.

Since the DDF domain is a superset of the SDF domain, all SDF primitives can be used in the DDF domain. Similarly for BDF primitives. Besides the SDF primitives, the DDF domain has some DDF-specific primitives that will be described in this chapter. The DDF-specific primitives overcome the main modeling limitation of the SDF domain in that they can model dynamic constructs such as conditionals, data-dependent iteration, and recursion. All of these except recursion are also supported by the BDF domain. It is even possible, in principle, to dynamically modify a DDF graph as it executes (the implementation of recursion does exactly this). The lower run-time efficiency of dynamic scheduling is the price we pay for the enhanced modeling power.
19.2 The DDF Schedulers

In MLDesigner, a scheduler determines the order of execution of blocks. This would seem to be a simple task in the DDF domain, since there is nothing to do at setup time, and at run time, the scheduler only needs to determine which blocks are runnable and then fire those blocks. Experience dictates, however, that this simple-minded policy is not adequate. In particular, it may use more memory than is required (it may even require an unbounded amount of memory when a bounded amount of memory would suffice). It may also be difficult for a user to specify for how long an execution should proceed.

In the SDF domain, an iteration is well-defined. It is the minimum number of firings that brings the buffers back to their original state. In SDF, this can be found by a compile-time scheduler by solving the balance equations. In both BDF and DDF, it turns out that it is undecidable whether such a sequence of firings exists. This means that no algorithm can answer the question for all graphs of a given size in finite time. This explains, in part, why the BDF domain may fail to construct a compile-time schedule and fall back on the DDF schedulers.

There are three simple and obvious criteria that a DDF scheduler should satisfy:

1. The scheduler should be able to execute a graph forever if it is possible to execute a graph forever. In particular, it should not stop prematurely if there are runnable primitives.
2. The scheduler should be able to execute a graph forever in bounded memory if it is possible to execute the graph forever in bounded memory.
3. The scheduler should execute the graph in a sequence of well-defined and determinate iterations so that the user can control the length of an execution by specifying the number of iterations to execute.

Somewhat surprisingly, it turns out to be extremely difficult to satisfy all three criteria at once. The first few versions of the DDF scheduler did not satisfy (2) or (3). The older scheduler is still available (set the useFastScheduler target parameter to YES), but its use is not recommended. Its behavior is somewhat unpredictable and sometimes counterintuitive. For example, told to run a graph for one iteration, it may in fact run it forever. Nonetheless, it is still available because it is significantly faster than the newer schedulers. However, there is still no satisfying implementation in MLDesigner which combines both the advantages of the old scheduler and the needed criteria.

The reason that these criteria are hard to satisfy is fundamental. It has already pointed out that it is undecidable whether a sequence of firings exists that will return the graph to its original state. This fact can be used to show that it is undecidable whether a graph can be executed in bounded memory. Thus, no finite analysis can always guarantee (2). The trick is that the DDF scheduler in fact has infinite time to run an infinite execution, so, remarkably, it is still possible to guarantee condition (2). The new DDF schedulers do this.

Regarding condition (1), it is also undecidable whether a graph can be executed forever. This question is equivalent to the halting problem, and the DDF model of computation is sufficiently rich that the halting problem cannot always be solved in finite time.

Condition (3) is more subtle and centers around the desire for determinate execution. This means a user should be able to tell immediately what primitives will fire in one iteration, knowing the
state of the graph. In other words, which primitives fire should not depend on arbitrary decisions made by the scheduler, like the order in which it examines the primitives.

To illustrate that this is a major issue, suppose we naively define an iteration to consist of “firing all enabled primitives at most once.” Consider the simple example in fig. 19.1. Primitive A is enabled, so we can fire it. Suppose this makes primitive B enabled. Should it be fired in the same iteration? Will the order in which we fire enabled primitives or determine whether primitives are enabled impact the outcome?

![Figure 19.1: A simple example used to illustrate the notion of an iteration](image)

Two policies have been implemented in DDF. These are explained below.

### 19.2.1 DDF Backward Scheduler

A Backward propagation scheduler has been introduced in the DDF domain. This scheduler computes needed firings backward, from sinks to sources. The prerequisite here is that sink primitives in the system have the **EndCondition** parameter set to *Yes* and the **NumOfItems** set to a reasonable value.

Blocking for converted COSSAP primitives is also supported.

To use the **useBackwardPropagationSimulator** you must set the target parameter to YES and reset the default scheduler parameter to NO as it is only possible to use one scheduler for a simulation. One of the target parameters ”useFastScheduler”, ”useForwardScheduler”, ”useBackwardPropagationScheduler”, or ”restructure” must be set to ”YES”. An error message is displayed if more than one scheduler is selected.

There are some subtleties in DDF scheduling. Due to these subtleties three DDF schedulers have been implemented, all accessible by setting appropriate target parameters. In the following section these schedulers are explained.

### 19.2.2 The default scheduler

The default scheduler, realized in the class **DDFSimpleSched**, first scans all primitives and determines which are enabled. In a second pass, it then fires the enabled primitives. Thus, the order in which the primitives fire has no impact on which ones fire in a given iteration.

Unfortunately, as stated, this simple policy still does not work. Suppose that primitive A in fig. 19.1 produces two particles each time it fires, and actor B consumes 1. Then our policy will be to fire actor A in the first iteration and both A and B in all subsequent iterations. This violates criterion (2), because it will not execute in bounded memory. More importantly, it is counterintuitive. Thus, the **DDFSimpleSched** class implements a more elaborate algorithm.
One iteration, by default, consists of firing all enabled and non-deferrable primitives once. If no primitives fire, then one deferrable primitive is carefully chosen to be fired. A **deferrable primitive** is one with any output arc (except a self-loop) that has enough data to satisfy the destination actor. In other words providing more data on that output arc will not help the downstream actor become enabled; it either already has enough data, or it is waiting for data on another arc. If a deferrable primitive is fired, it will be the one that has the smallest maximum output buffer sizes.

The algorithm is formally given with the following algorithm:

```plaintext
At the start of the iteration compute {
    E = enabled actors 
    D = deferrable actors
}

One default iteration consists of {
    if (E-D != 0) fire primitives in (E-D)
    else if (D != 0) fire the minimax primitive in D
    else deadlocked.
}

The minimax primitive is the one with the smallest maximum number of tokens on its output paths.
```

This default iteration is defined to fire actors at most once. Sometimes, a user needs several such **basic iterations** to be treated as a single iteration. For example, a user may wish for a **user iteration** to include one firing of an XMgraph primitive, so that each iteration results in one point plotted. The basic iteration may not include one such firing. Another more critical example is a wormhole that contains a DDF system but will be embedded in an SDF system. In this case, it is necessary to ensure that one user iteration consists of enough firings to produce the expected number of output particles.

This larger notion of an iteration can be specified using the target pragma mechanism to identify particular primitives that must fire some specific number of times (greater than or equal to one) in each user iteration. To use this, make sure the domain is DDF and the target is DDF-default.

Then in pigi, place the mouse over the icon of the primitive in question, and issue the edit-pragmas command ("a"). One pragma (the one understood by this target) will appear; it is called firingsPerIteration. Set it to the desired value. This will then define what makes up an iteration.

### 19.2.3 The clustering scheduler

If you set the target parameter restructure to YES, you will get a scheduler that clusters SDF actors when possible and invokes the SDF scheduler on them. The scheduler is implemented in the class DDFClustSched.

**NOTE:** As of this writing, this scheduler will not work with wormholes, and will issue a warning. Nonetheless, it is an interesting scheduler for two reasons, the first of which is its clustering behavior. The second is that it uses a different definition of a basic iteration.
In this definition, a basic iteration (loosely) consists of as many firings as possible subject to the constraint that no actor fires more than once and that deferrable actors are avoided if possible.

The complete algorithm is implemented as this:

The following sets are updated every time a primitive fires:
- \( E \) = enabled actors
- \( D \) = deferrable actors
- \( S \) = source actors
- \( F \) = actors that have fired once already in this iteration

One default iteration consists of:

```plaintext
while \( (E-D-F) \neq 0 \) {
    fire actors in \( (E-D-F) \)
}
if \( F == 0 \) {
    // All enabled actors are deferrable.
    // Try the non-sources first.
    if \( (E-S) \neq 0 \) {
        fire \( (E-S) \);
    } else {
        fire \( (S) \);
    }
}
if \( F == 0 \) deadlock
```

Use of this scheduler is not advised at this time, however. For one thing, the implementation of clustering adds enough overhead that this scheduler is invariably slower than the default scheduler.

19.2.4 The fast scheduler

In case the new definition of an iteration is inconvenient for legacy systems, we preserve an older and faster scheduler that is not guaranteed to satisfy criteria (2) and (3) above. The basic operation of the fast scheduler is to repeatedly scan the list of primitives in the domain and execute the runnable primitives until no more primitives are runnable, with certain constraints imposed on the execution of sources. For the purpose of determining whether a primitive is runnable, the primitives are divided into three groups. The first group of the primitives have input ports that consume a fixed number of particles. All SDF primitives, except those with no input ports, are included in this group. For this group, the scheduler simply checks all inputs to determine whether the primitive is runnable.

The second group consists of the DDF-specific primitives where the number of particles required on the input ports is unspecified. An example is the `EndCase` primitive (a multi-input version of the BDF `Select` primitive). The `EndCase` primitive has one control input and one multiport input for data. The control input value specifies which data input port requires a particle. Primitives in this group must specify at run time how many input particles they require on each input
19 DDF Domain

Port. Primitives specify a port with a call to a method called waitPort and the number of particles needed with a call to waitNum. To determine whether a primitive is runnable, the scheduler checks whether a specified input port has the specified number of particles.

For example, in the EndCase primitive, the waitPort points to the control input port at the beginning. If the control input has enough data (one particle), the primitive is fired. When it is fired, it checks the value of the particle in the control port, and changes the waitPort pointer to the input port on which it needs the next particle. The primitive will be fired again when it has enough data on the input port pointed by waitPort. This time, it collects the input particle and sends it to the output port. See fig. 19.2.

![Figure 19.2: (a) The EndCase primitive waits on the control port. (b) The primitive fires when data arrives on the control port (the value of the data is 0). (c) Now the primitive waits for input to arrive on input port 0. (d) The primitive fires again when data arrives on input port 0. (e) The data that arrived on input port 0 is transmitted by the output port of the EndCase primitive.](image)

The third group of primitives comprises sources. Sources are always runnable. Source primitives introduce a significant complication into the DDF domain. In particular, since they are always runnable, it is difficult to ensure that they are not invoked too often. This scheduler has a reasonable but not foolproof policy for dealing with this. Recall that the DDF domain is a superset of the SDF domain. The definition of one iteration for this scheduler tries to obtain the same results as the SDF scheduler when only SDF primitives are used. In the SDF domain, the number of firings of each source primitive, relative to other primitives, is determined by solving the balance equations. However, in the DDF domain, the balance equations do not apply in the same form. The technique used instead is lazy-evaluation.

19.2.5 Lazy evaluation

At the beginning of each iteration of a DDF application, we fire all source primitives exactly once, and temporarily declare them "not runnable." We also fire all primitives that have enough initial tokens on their inputs. After that, the scheduler starts scanning the list of primitives in the domain. If a primitive has some particles on some input arcs, but is not runnable yet, then the primitive initiates the (lazy) evaluation of those primitives that are connected to the input ports requiring more data. This evaluation is "lazy" because it occurs only if the data it produces are actually needed. The lazy-evaluation technique ensures that the relative number of firings of source primitives is the same under the DDF scheduler as it would be under the SDF scheduler.

We can now define what is meant by one iteration in DDF. An iteration consists of one firing of each source primitive, followed by as many lazy-evaluation passes as possible, until the system deadlocks. One way to view this (loosely) is that enough primitives are fired to consume all of
the data produced in the first pass, where the source primitives were each fired once. This may involve repeatedly firing some of the source primitives. However, a lazy-evaluation is only initiated if a primitive in need of inputs already has at least one input with enough tokens to fire. Because of this, in some circumstances, the firings that make up an iteration may not be exactly what is expected. In particular, when there is more than one sink primitive in the system, and the sink primitives fire at different rates, the ones firing at higher rates may not be fired as many times as expected. It is also possible for one iteration to never terminate.

When a DDF wormhole is invoked, it will execute one iteration of the DDF system contained in it. This is a serious problem in many applications, since the user may need more control over what constitutes one firing of the wormhole.

19.3 Inconsistency in DDF

So far, an error-free program has been assumed. In the SDF domain, compile-time analysis detects errors due to inconsistent rates of production and consumption of tokens because the balance equations cannot be solved. In DDF, however, such inconsistencies are harder to detect. Our strategy is to detect them at run time, an approach that has two disadvantages. First, it is costly, as will be explained shortly. Second, it is not easy to isolate the sources of errors.

A data flow graph is called consistent if on each arc, in the long run, the same number of particles are consumed as produced [Lee91b]. One source of inconsistency is the sample-rate mismatch that is common to the SDF domain. The DDF domain has more subtle error sources, however, due to the dynamic behavior of DDF primitives. In an inconsistent graph, an arc may queue an unbounded number of tokens in the long run. To prevent this, we examine the number of tokens on each arc to detect whether the number is greater than a certain limit (the default is 1024). If we find an arc with too many tokens, we consider it an error and halt the execution. We can modify the limit by setting the target parameter named maxBufferSize. The two new schedulers will interpret a negative number here to be infinite capacity. An inconsistent system will run until your computer runs out of memory.

The value of the maxBufferSize parameter will be the maximum allowed buffer size. Since the source of inconsistency is not unique, isolating the source of the error is usually not possible. We can just point out which arc has a large number of tokens. Of course, if the limit is set too high, some errors will take very long to detect. Note however that there exist perfectly correct DDF systems (which are consistent) that nonetheless cannot execute in bounded memory. It is for this reason that the new schedulers support infinite capacity.

19.4 The default-DDF target

The DDF domain only has one target. The parameters of the target are:

maxBufferSize (INT) Default = 1024
The capacity of an arc (in particles). This is used for the runtime detection of inconsistencies, as explained above. If any arc exceeds this
capacity, an error is flagged and the simulation halts. A negative number is interpreted as infinite capacity (unless useFastScheduler is YES). The value of this parameter does not specify how much memory is allocated for the buffers, since the memory is allocated dynamically.

schedulePeriod (FLOAT) Default = 0.0
This defines the amount of time taken by one iteration (defined above) of the DDF schedule. This is used only for interface with timed domains, such as DE. Note that if you want the count given in the debug panel of the run control panel to indicate the number of iterations, you should set this parameter to one.

runUntilDeadlock (INT) Default = NO
Unless useFastScheduler is set, this modifies the definition of a single iteration to invoke all primitives as many times as possible, until the system halts. It is risky to use this because the system may not halt. But in wormholes it is sometimes useful.

restructure (INT) Default = NO
This specifies that the experimental scheduler DDFClustSched should be used. This scheduler attempts to form SDF clusters for more efficient execution. Its use is not advised at this time, however, since it does not work properly with wormholes and is slower than the default scheduler.

useFastScheduler (INT) Default = NO
This specifies that the older and faster DDF scheduler (from version 0.5.2) should be used. It is difficult, however, to control the length of a run with this scheduler.

numOverlapped (INT) Default = 1 For the fast scheduler only, this gives the number of iteration cycles that can be overlapped for execution. When a DDF system starts up, it normally begins by firing each source primitive once, as explained above. It then goes into a lazy evaluation mode. Setting this parameter to an integer N larger than one allows the scheduler to begin with N firings of the source primitives instead of just one. This can make execution more efficient, because primitives downstream from the sources will be able to fire multiple times in each pass through the graph. The default value of this parameter is 1.

logFile (STRING)
The default is the empty string. If non-empty, this gives the name of a file to be used for recording scheduler information.

19.5 An overview of DDF primitives

Case
Route an input particle to one of the outputs depending on the control particle. The control particle should be between zero and N - 1, inclusive, where N is the number of outputs.
19.6 An overview of DDF demos

Depending on the control particle, consume a particle from one of the data inputs and send it to the output. The control particle should have value between zero and N - 1, inclusive, where N is the number of inputs.

**DownCounter**

Given an integer input with value N, produce a sequence of output integers with values (N - 1), (N - 2), ... 1, 0.

**LastOfN**

Given a control input with integer value N, consume N particles from the data input and produce only the last of these at the output.

**Repeater**

Given a control input with integer value N, and a single input data particle, produce N copies of the data particle on the output.

**Self**

This is a first exploration of recursion and higher-order functions in data flow. It is still experimental, so do not expect it to be either efficient or bug-free. The primitive "represents" the module given by the parameter recurGal, which must be above it in the hierarchy. That is, when the Self primitive fires, it actually invokes the module that it represents. Since that module is above the Self primitive in the hierarchy, it contains the Self primitive somewhere within it. Thus, this primitive implements recursion. Since the Self primitive takes an argument (recurGal) that specifies the function to invoke, it is itself a higher-order function. The instance of the recurGal module is not created until it is actually needed, so the number of instances (the depth of the recursion) does not need to be known a priori. If the parameter reinitialize is NO or FALSE, then the instance of the module is created the first time it fires and reused on subsequent firings. If reinitialize is YES or TRUE, then the module is created on every firing and destroyed after the firing. Inputs are sent to the instance of the module and outputs are retrieved from it. The inputs of the named module must be named "input#?" and the outputs must be named "output#?", where "#?" is replaced with an integer starting with zero. This allows the inputs and outputs of this primitive to be matched unambiguously with the inputs and outputs of the referenced module.

19.6 An overview of DDF demos

These demos illustrate dynamic data flow principles.

**eratosthenes**

The sieve of Eratosthenes is a recursive algorithm for computing prime numbers. This demo illustrates the implementation of recursion in the DDF domain. This is a concept demonstration only.

**errorDemo**

An example of an inconsistent DDF system. An inconsistent DDF program is one where the long term average number of particles produced on an arc is not the same as the average long term number of particles consumed. This error is detected by bounding the buffer sizes and
detecting overflow.

**ifThenElse**
This demo illustrates the use of an SDF wormhole to implement a dynamically scheduled construct using the DDF domain. An if-then-else is such a dynamically scheduled construct. The top level schematic represents an SDF system, while the inside schematic represents a DDF system (implementing an if-then-else).

**fibonnacci**
Generate the Fibonacci sequence using a rather inefficient recursive algorithm that is nonetheless a good example of how to realize recursion.

**loop**
This demo illustrates data-dependent iteration. Input integers are repeatedly multiplied by 0.5 until the product is less than 0.5. Turn on animation to see the iteration.

**picture**
Construct a two-dimensional random walk using a hierarchy of nested wormholes. The outermost SDF domain has a wormhole called "draw-line" which internally uses the DDF domain. That wormhole, in turn, has a wormhole called "display" which internally uses the SDF domain.

**repeat**
This simple demo shows the effect of running a DDF scheduler on an SDF system. The firingsPerIteration pragma is used to control the meaning of an iteration.

**repeater**
This is a simple illustration of the Repeater primitive, used in an SDF wormhole (DDF inside SDF).

**router**
This is a simple illustration of the EndCase primitive.

**SDFinDDF**
This rather trivial demo illustrates the use of a DDF wormhole whose inside domain is SDF. The top-level system (in the DDF domain) has an if-then-else overall structure, implemented of a matching pair of Case and EndCase primitives. The inside system (in the SDF domain) multiplies the data value by a ramp.

**threshtest**
This demo shows that Karp & Miller style thresholds are supported in DDF. The Thresh primitive is a dummy that implements a settable threshold.

**timing**
This demo illustrates the use of the DDF domain to implement asynchronous signal processing systems. In this case, the system performs baud-rate timing recovery using an approximate minimum mean-square-error (MMSE) technique.

### 19.7 Mixing DDF with other domains

The mixture of the DDF domain with other domains requires a conversion between different computational models. In particular, some domains associate a *time stamp* with each particle, and other domains do not. Thus, a common function at the EventHorizon is the addition of time.
stamps, the stripping off of time stamps, interpolation between time stamps, or removal of redundant repetitions of identical particles. In this section, DDF-specific features on the domain interface will be discussed.

A module or system implemented using DDF may have a wormhole which contains a subsystem implemented in another domain. The DDF wormhole looks exactly like a DDF primitive from the outside. However, there are certain technical restrictions. In particular, it cannot have dynamic input portholes, meaning the number of particles consumed by the wormhole inputs is a compile-time constant. The wormhole is therefore fired when all input ports have new particles. When it is fired, it consumes the input data, invokes the scheduler of the inner domain, and retrieves the output particles. Thus, in all respects except one, the DDF wormhole behaves like an SDF wormhole (see “Wormholes” on page 12-4 for more information). The one exception is that the DDF wormhole need not consistently produce outputs.

When a DDF system is embedded within another domain, you may need to explicitly control what constitutes a firing of the subsystem. Specifically, by setting the firingsPerIteration pragma of a primitive in the DDF subsystem, you control how many firings of that primitive are required to complete an iteration. Zero means “don’t care.”

Note that some work has been done with a CGDDF target which recognizes and implements certain commonly used programming constructs. See “CG Domain” on page 13-1 for more information.
Chapter 20

BDF Domain

20.1 Introduction

Boolean-controlled data flow (BDF) is a domain that can be thought of as a generalization of synchronous data flow (SDF). It supports dynamic flow of control but still permits much of the scheduling work to be performed at compile time. The dynamic data flow (DDF) domain, by contrast, makes all scheduling decisions at run time. Thus, while BDF is a generalization of SDF, DDF is still more general. Accordingly, the BDF domain permits SDF actors to be used, and the DDF domain permits BDF actors to be used. This chapter will assume that the reader is familiar with the SDF domain.

The BDF domain can execute any actor that falls into the class of Boolean-controlled data flow actors. For an actor to be SDF, the number of particles read by each input porthole, or written by each output porthole, must be constant. Under BDF, a generalization is permitted: the number of particles read or written by a porthole may be either a constant or a two-valued function of a particle read on a control porthole for the same primitive. One of the two values of the function must be zero. The effect of this is that a porthole might read tokens only if the corresponding control particle is zero (FALSE) or nonzero (TRUE). The control porthole is always of type integer, and it must read or write exactly one particle. Although the particles on the control porthole are of integer type, we treat them as Booleans, using the C/C++ convention that zero is false and nonzero is true.

We say that a porthole that conditionally transfers data based on a control token is a conditional porthole. A conditional input porthole must be controlled by a control input. A conditional output porthole may be controlled by either a control input or a control output. These restrictions permit the run-time flow of control to be determined by looking only at the values of particles on control ports. The compile-time scheduler determines exactly how the flow of control will be altered at run time by the values of these particles. It constructs what we call an annotated schedule, which is a compile-time schedule where each firing is annotated with the run-time conditions under which the firing should occur.

The theory that describes graphs of BDF actors and their properties is called the token flow model. Its properties are summarized in [BL93b] and developed in much more detail in [Buc93].

The BDF scheduler performs the following functions. First, it performs a consistency check anal-
ogous to the one performed by the SDF scheduler to detect certain types of errors corresponding to mismatches in particle flow rates [Lee91b]. Assuming that no error is detected, it then applies a clustering algorithm to the graph, attempting to map it into traditional control structures such as if-then-else and do-while. If this clustering process succeeds in reducing the entire graph to a single cluster, the graph is then executed with the quasi-static schedule corresponding to the clusters. (It is not completely static since some actors will be conditionally executed based on control particle values, but the result is “as static as possible.”) If the clustering does not succeed, then the resulting clusters may optionally be executed by the same dynamic scheduler as is used in the DDF domain. Dynamic execution of clusters is enabled or disabled by setting the allowDynamic parameter of the default-BDF target.

20.2 The default-BDF target

At this time, there is only one BDF target. The parameters of the target are:

- **logFile** (STRING) The default is the empty string. The filename to which to report various information about a run. If this parameter is empty (the default), there will be no reporting. If the parameter is cerr or cout, messages will go to the Unix standard error or standard output, respectively.

- **allowDynamic** (INT) Default = NO If TRUE or YES, then dynamic scheduling will be used if the compile-time analysis fails to completely cluster the graph. As shown in [Buc93], there will always be some valid graphs that cannot be clustered.

- **requireStronglyConsistent** (INT) Default = NO If TRUE or YES, then a graph will be rejected if it is not “strongly consistent” [Lee91b]. This will cause some valid systems, even systems that can be successfully statically scheduled, to be rejected.

- **schedulePeriod** (FLOAT) Default = 10000.0 This defines the amount of time taken by one iteration of the BDF schedule. The notion of “iteration” is defined in the SDF chapter, in the section 5.1.3.

20.3 An overview of BDF primitives

- **CondGate** If the value on the “control” input is nonzero, the input particle is copied to output. Otherwise, no input is consumed (except the control particle) and no output is produced. This is effectively one half of a Select.

- **Fork** Copy the input particle to each output. The SDF fork is not used here because the BDF domain requires some extra steps to assert that each output of a fork is logically equivalent if the input is a Boolean signal.

- **Not** Output the logical inverse of the Boolean input. Again, the equiva-
lent SDF logic block is not adequate because extra steps are needed to assert the logical relationship between the input and the output.

Select
If the value on the "control" porthole is nonzero, N tokens (from the parameter N) from "trueInput" are copied to the output; otherwise, N tokens from "falseInput" are copied to the output.

Switch
Switch input particles to one of two outputs, depending on the value of the control input. The parameter N gives the number of particles read in one firing. If the particle read from the control input is TRUE, then the values are written to "trueOutput"; otherwise they are written to "falseOutput".

20.4 An overview of BDF demos

These demos with icons illustrate Boolean-controlled data flow principles. A useful way to understand these principles when running BDF demos is to display the schedule after a run. This must be done before the control panel is dismissed, because dismissing the control panel destroys the scheduler.

bdfTiming
This demo is identical to the DDF timing demo, except that it uses BDF Switch and Select primitives instead of DDF Case and EndCase. The static schedule has some simple if-then constructs to implement conditional firing.

dataIter
This simple system, which does nothing interesting, is surprisingly difficult to schedule statically. It requires nesting an if-then within a do-while within a manifest iteration.

ifThenElse
This simple system uses Switch and Select primitives to construct an if-then-else.

insanity
This peculiar system applies two functions, log and cosine, but the order of application is chosen at random. The BDF clustering algorithm fails to complete on this graph. If the allowDynamic parameter of the target is set to YES, then the scheduler will construct four SDF sub-schedules, which must then be invoked dynamically.

loop
This system illustrates the classic data flow mechanism for implementing data-dependent iteration (a do-while). A sequence of integers (a ramp) is the overall input. Each input value gets multiplied by 0.5 inside the loop until its magnitude is smaller than 0.5. Then that smaller result is sent to the output.

loopTheLoop
This system is similar to the loop demo, except that a second do-while loop is nested within the first.

mandelbrot
This system calculates the Mandelbrot set and uses Matlab to plot the output. Matlab must be installed on the local workstation to view the output of this demo.
Chapter 21

HOF Domain

21.1 Introduction

A function is higher-order if it takes a function as an argument and/or returns a function. A classic example is `mapcar` in Lisp, which takes two arguments, a function and a list. Its behavior is to apply the function to each element of the list and to return a list of the results. The HOF domain implements a similar function, in the form of a primitive called `Map`. The `Map` primitive replaces itself with another primitive (or module) until all parameters of the replacement block and the system have been fulfilled. This essentially means you can control the number of instances of a primitive or module within a simulation by changing the buswidth on connections or you can add more complex functions by changing one parameter in a replacement block. Many other useful higher-order functions are also provided by this domain.

The HOF domain provides a collection of primitives designed to be usable in all other MLDesigner domains. To preserve this generality, not all interesting higher-order functions are implemented in the HOF domain library. As a consequence, some individual domains may also define higher-order functions and the relevant primitives (with domain specific behavior) will be found in their respective libraries.

A common feature shared by all the primitives in this domain is that they perform all of their operations in the `preinitialize` method. Moreover, their basic operation is always to disconnect themselves from the graph in which they appear and then to self-destruct. Since the pre-initialization method of the primitives in a system is invoked before the pre-initialization method of the scheduler, the scheduler never sees the HOF primitives. They will have self-destructed by the time the scheduler is invoked. This is why these primitives will work in any domain. In code generation domains, an important feature of the HOF primitives is that they add no run-time overhead at all, since they self-destruct before code generation begins, and therefore do not appear in any form in the generated code.

Many of the HOF primitives are replaced by one or more instances of another primitive or module, called the replacement block. Replacement blocks are generally included in the same position originally occupied by the HOF primitive, but different HOF primitives will connect these replacement blocks in different ways.

Some HOF primitives have no replacement block. Before they self-destruct, they will typi-
cally only alter the connections in the graph without adding any new blocks. An example is the BusMerge block, which merges two busses into one wider bus. These primitives are called bus manipulation primitives.

The experienced reader may have some difficulty connecting the concept of higher-order functions, as implemented in this domain, to that used in functional programming. This issue is covered in some depth in [LP95], but we can nonetheless give a brief motivation here. In functional languages, there is no syntactic difference between a function argument that is a data value, one that is a stream (an infinite sequence of data values), and one that is a function. In visual programming, however, functions typically have two very different syntaxes for their arguments. MLDesigner is no exception. Primitives and modules in MLDesigner are functions with two kinds of arguments: input streams and parameters. The HOF domain only contains primitives where a parameter may be function. It does not contain any primitives that will accept functions at their input portholes as part of an input stream, or produce functions at their output portholes. Although in principle such higher-order functions can be designed in MLDesigner, their behavior would not be independent of their domain, so the HOF domain would be the wrong place for them.

21.2 Using the HOF domain

The primitives contained in the HOF domain are designed for use in all MLDesigner domains. This section covers using HOF primitives in the SDF domain only.

21.3 The Map primitive and its variants

The Map primitive is the most basic of all HOF primitives. It has the following parameters:

- **blockname**: The name of the replacement block.
- **where_definition**: The full path and facet name for the definition of blockname. Only needed if the primitive is not a built-in primitive.
- **parameter_map**: Set the parameters of the replacement block and define a function to perform.
- **input_map**: Defines how to connect the inputs.
- **output_map**: Defines how to connect the outputs.

The name of the replacement block is given by the **blockname** parameter. If the replacement block is a module, then the **where_definition** parameter should give the full name (including the full path) of a facet that, when compiled, will define the block. This path name may (and probably should) begin with the environment variable $MLD or ~username to avoid linking problems after changes are made to the file system. If the replacement block is a built-in primitive, then there is no need to give a value to the **where_definition** parameter.

The Map primitive replaces itself in the graph with as many instances of the replacement block as
needed to satisfy all of the inputs to, or outputs from, the Map primitive. Consider the following example (see fig. 21.1).

### 21.3.1 Example

The system seen here is modeled in the SDF Domain. We first created a Library called HOF Exercises and then created a system in this library called Example. If you would like to recreate this system proceed as follows:

1. In the Library View expand the library MLD Libraries/SDF Domain/Sources and click and drag the Impulse primitive into the system. Repeat this process until you have three instances of the Impulse primitive in the system.
2. Expand the library MLD Libraries/SDF Domain/Sinks and drag one instance of the XMGraph primitive into the system.
3. Expand the library MLD Libraries/HOF Domain and drag one instance of the Map primitive into your system.

You can now connect the model instances as shown in fig. 21.1.

4. Select the Add Bus icon with a single mouse click and move the cursor to the position over the connection between the Map model instance and XMgraph.
5. A single mouse-click places a bus on the connection. A right mouse click reverts the cursor mode back to selection mode.
6. Select the undefined bus property with a single mouse-click.
7. Define the Bus Width in the Relation Properties window (bottom left corner of the GUI) by typing 3 in the Bus Width input field.

You now need to define the replacement block for the Map primitive. We want to define the replacement block as the RaisedCosine built-in primitive found in MLD Libraries/SDF Domain/DSP. We also want to perform a function on the parameter excessBW (excess Band Width) defined in the primitive. Go to the library MLD Libraries/SDF Domain/DSP and open the RaisedCosine primitive. The primitive’s parameter names and input and output naming conventions must be observed. Close the primitive once you are familiar with the parameters here.

8. Select the Map model instance with a single mouse-click.
9. The Instance Properties window is activated (bottom left corner of the GUI)
10. In the input field of the parameter blockname type RaisedCosine
11. In the input field of the parameter parameter_map type excessBW = 1.0/instance
12. In the input fields input_map and output_map type signalIn and signalOut respectively.

Set the RunLength to 5 and switch to simulation mode. Run the simulation and observe the results (see fig. 21.2).
The replacement block is specified as the built-in `RaisedCosine` primitive. Since this is built-in, there is no need to specify where it is defined, so the `where_defined` parameter is blank. The `RaisedCosine` primitive has a single input named `signalIn` and a single output named `signalOut`, so these names are given as the values of the `input_map` and `output_map` parameters. The `parameter_map` parameter specifies the values of the `excessBW` parameter for each instance of the replacement block to be created; `excessBW` specifies the excess bandwidth of the raised cosine pulse generated by the primitive. The syntax of the `parameter_map` parameter is discussed in detail below, but we can see that the value of the `excessBW` parameter will be 1.0 for the first instance of the `RaisedCosine` primitive, 0.5 for the second, and 0.33 for the third.

![Diagram of the system](image)

Figure 21.1: An example of the use of the `Map` primitive to plot three different cosine pulses

The last connection on the right in fig. 21.1 is a Bus, which is much like a delay in that the icon is placed directly over the arc. Its single parameter specifies the number of connections that the single wire represents. Here, the bus width has to be three or the `Map` primitive will issue an error message. This is because there are three inputs to the `Map` primitive, so three instances of the `RaisedCosine` primitive will be created. The three outputs from these three instances need somewhere to go. The block diagram in fig. 21.1 is equivalent to that in fig. 21.3. Indeed, once the pre-initialization method of the `Map` primitive has run, the topology of the MLDdesigner system will be exactly as shown in fig. 21.3. The `Map` primitive itself will not appear in the topology, so examining the topology after converting the system to a `PTCL` file, for example, the `PTCL print` command will not show a `Map` primitive instance.

In fig. 21.3, the number of instances of the `RaisedCosine` primitive is specified graphically. In fig. 21.1, it is specified by implication, through the number of instances of the `Impulse` primitive. Neither of these really takes advantage of higher-order functions. The block diagram in fig. 21.4 is equivalent to both fig. 21.1 and fig. 21.3, but can be more easily modified to include more or fewer instances of the `RaisedCosine` primitive. It is only necessary to modify parameters, not the graphical representation. For example, if the value of the bus parameters in fig. 21.4 were changed from 3 to 10, the system would then plot ten raised cosines instead of three.
21.3 The Map primitive and its variants

Figure 21.2: The plot that results from running the system

Figure 21.3: A block diagram equivalent to the demo above, but without higher-order functions

Figure 21.4: A block diagram equivalent to the two demos above, except that the number of instances can be specified by a parameter
The left-most primitive in fig. 21.4 is a variant of the Map primitive called Src. It has no inputs, and is used when the replacement block is a pure source block with no input.

### 21.3.2 MapGR and SrcGR primitive

The MapGr and SrcGr primitives work just like the Map and Src primitives, except that the replacement block is graphically defined rather than textually.

![Diagram](image_url)

**Figure 21.5: A block diagram equivalent to the last demo, except that the replacement blocks are specified graphically.**

A more complex application of the MapGr primitive is shown in fig. 21.6. Here, the replacement block is a Commutator, which can take any number of inputs. The bus connected to its input multiporthole determines how many inputs will be used in each instance created by the MapGr primitive. In the example in fig. 21.6, it is set to 2. Thus, each instance of the replacement block processes two input streams and produces one output stream. Consequently, the input bus must be twice as wide as the output bus, or the MapGr primitive will issue an error message.

The Parameter **parameter_map** should be set in order to make use of the advantages of higher order functions. Map the parameter of the SrcGr instance to the value parameter of the Ramp primitive and use the variable **instance_number** to give different values to the particle on the output port of the ramp with each iteration. Enter the following in the appropriate field.

```
step = instance_number
```

Set the **RunLength** to 20 and Save the system. Switch to Simulation mode and click **Go**. This example produces the plot shown in fig. 21.7. A key advantage of higher-order functions becomes apparent when we realize that the parameters can be easily changed. It is quick and easy to change the amount of times a module or primitive is instantiated by simply changing the bus width.

### 21.3.3 Setting parameter values

The **parameter_map** parameter of the Map primitive and related primitives can be used to set parameter values in the replacement blocks. The **parameter_map** is a string array, a list of strings. The strings are in pairs, where the pairs are separated by spaces, and there are four acceptable forms for each pair:
21.3 The Map primitive and its variants

Figure 21.6: A more complicated example using higher-order functions with the number of replacement blocks graphically defined.

Figure 21.7: The plot created by running the system above.
name value
name(number) value
name = value
name(number) = value

There should be no spaces between name and (number), and the name cannot contain spaces, =, or (. In all cases, name is the name of a parameter in the replacement block. In the first and third cases, the value is applied to all instances of the replacement block. In the second and fourth cases, it is applied only to the instance specified by the instance number, (which starts with 1). The third and fourth cases just introduce an optional equal sign, for readability. If the = is used, there must be spaces around it.

The value can be any usual MLDesigner expression for giving the value of a parameter. If this expression has spaces in it, however, then the value should appear in quotation marks so that the whole expression is kept together. In case the value should contain quotes, then each of these quotes should be preceded by a backslash:

```
Parameter1 = ""string 1" "string 2" "string 3"
Parameter2 = "Root.String{"a data structure}"
```

In the example above, Parameter1 is of type stringarray and is initialized with an array containing 3 strings, while Parameter2 is a data structure parameter of type Root.String.

If the string instance_number appears anywhere in value, it will be replaced with the instance number of the replacement block. Note that it need not be a separate token. For example, the value xxxinstance_numberyyy will become xxx1yyy for the first instance, xxx2yyy for the second, etc. After all appearances of the string instance_number have been replaced, value is evaluated using the usual MLDesigner expression evaluator for initializing String Array states.

For example, in fig. 21.1, the Map primitive has a blockname of RaisedCosine, and a parameter_map of

```
excessBW = 1.0/instance_number
```

When the system is run, the Map primitive will create three instances of RaisedCosine. The first instance will have its excessBW parameter set to 1.0 (which is 1/1), the second instance of RaisedCosine will have a excessBW of 0.5 (1/2) and the third will have an excessBW of 0.33 (1/3). Since the other RaisedCosine parameters are not mentioned in the parameter_map, they are set to their default values.

As a further example, suppose parameter_map of the Map primitive in fig. 21.1 were set to

```
excessBW(1) 0.6 excessBW(2) 0.5 excessBW(3) 0.4 length 128
```

The first RaisedCosine would then have an excessBW of 0.6, the second would have an excessBW of 0.5 and the third would have 0.4 for its excessBW. All three of the Raised-Cosine primitives would have a length of 128 instead of the default length.
21.3.4 Number of replacement blocks

The number of instances of the replacement block is determined by the number of input or output connections that have been made to the \texttt{Map} primitive. Suppose the \texttt{Map} primitive has \( M_I \) inputs and \( M_O \) outputs connected to it. Suppose further that the replacement block has \( B_I \) input ports and \( B_O \) output ports. Then

\[
N = \frac{M_I}{B_I} = \frac{M_O}{B_O}
\]

is the number of instances that will be created. This must be an integer. Moreover, the number of input and output connections must be compatible (must satisfy the above equality), or you will get an error message like: "too many inputs for the number of outputs."

21.3.5 How the inputs and outputs are connected

The first \( B_I \) inputs to the \texttt{Map} primitive will be connected to the inputs of the first instance of the replacement block. To determine in what order these \( B_I \) connections should be made, the names of the inputs to the replacement block should be listed in the \texttt{input_map} parameter in the order in which they should be connected. There should be exactly \( B_I \) names in the \texttt{input_map} list. The next \( B_I \) inputs to the \texttt{Map} primitive will be connected to the next replacement block, again using the ordering specified in \texttt{input_map}. Similarly for the outputs. If there are no inputs at all, then the number of instances is determined by the outputs, and vice versa.

For \texttt{MapGr} and its variants, there is no \texttt{input_map} or \texttt{output_map} parameter; all connections are specified graphically.

For both \texttt{Map} and \texttt{MapGr}, if you want to replicate a module then you need to create a module representing the group to be replicated. The same is true of the remaining HOF primitives that generate multiple instances of a block.

21.3.6 A note about data types

All the HOF primitives show their input and output data types as ANYTYPE. In reality, the type constraints are those of the replacement blocks, which might have portholes of specific types. The HOF primitives rewire the schematic before any attempt is made to determine porthole types, so the actual assignment of particle types is the same as if the schematic had been written out in full without using any HOF primitives.

21.4 Other higher-order control structures

The \texttt{Map} primitive and its variants apply instances of their replacement block in parallel to the set of input streams. Another alternative is provided by the \texttt{Chain} primitive, which strings together some specified number of instances of the replacement block in series. The parameters are similar to those of the \texttt{Map} primitive, except for the addition of \texttt{internal_map}. The \texttt{internal_map} parameter specifies connections made between successive instances of the replacement block in the cascade. It should consist of an alternating list of output and input names for the replacement
An example of the use of the Chain primitive is a string of biquad filters in series. The IIR filter primitive, which can be used to create a biquad filter, has an input named `signalIn` and an output named `signalOut`. To have a string of these primitives in series, one would want the output of the first IIR primitive in the series to be connected to the input of the second primitive. And the output of the second primitive should be connected to the input of the third, etc. Thus, a Chain primitive that is a series of biquad filters would have an internal map of

\[
\text{signalOut} \rightarrow \text{signalIn}
\]
to specify that the output of one block is connected to the input of the next.

Another variant is the IfElse block. This primitive is just like Map, except that it has two possible replacement blocks. If the condition parameter is TRUE, then the true_block is used. Otherwise, the false_block is used. It is important to realize that the condition parameter is evaluated at pre-initialization time. Once a replacement block has been selected, it cannot be changed. There are two uses for this block. It can be used to parameterize a module in such a way that the parameter determines which of two functions is used within the computation. More interestingly, it can be used to implement statically-evaluated recursion.

## 21.5 Statically evaluated recursion

The Map primitive and its variants replace themselves with an instance of the block specified as the replacement block. What if that block is a module within which the very Map primitive in question sits? This is a recursive reference to the module, but a rather awkward one. In fact, in such a configuration, the pre-initialization phase of execution will never terminate. The user has to manually abort such an execution in order to get it to terminate.

The IfElse primitive, however, can conditionally specify one of two replacement blocks. The condition parameter determines which block. One of the two replacement blocks can be a recursive reference to a module as long as the condition parameter is modified. When the condition parameter changes state, going from TRUE to FALSE or FALSE to TRUE, then the choice of replacement block inside the new module instance will change. This can be used to terminate the recursion.

Consider the example shown in fig. 21.8. This module has a single parameter, `log2framesize`. It will read \(2^{\log_2 \text{framesize}}\) input particles and rearrange them in bit reversed order. That is, they will emerge from the module as if their binary address had been interpreted with the high-order bit reinterpreted as a low-order bit, and vice versa. Suppose for example that \(\log_2 \text{framesize} = 3\), and that the input sequence is 10,11,12,13,14,15,16,17. Then the output sequence will be 10,14,12,16,11,15,13,17.

To accomplish this, the bit_reverse module uses two IfElse primitives, each with a conditional recursive reference to the bit_reverse module.

---

1 For those unfamiliar with bit-reversed addressing, here is a quick introduction. Since \(\log_2 \text{framesize} = 3\), the module will read \(2^3 = 8\) values at a time. The first value (10) has address 0 (since computers always seem to count from zero) which is 000 in binary. Reversed, its address is still 000 so it is output first. The second value (11) has address 1 which is 001 in binary. Reversed, its address is 100 binary which is 4. Thus the value (11) is output in the
21.6 Bus manipulation primitives

Figure 21.8: A recursive system, where the IfElse HOF primitive replaces itself with an instance of the same system until its condition parameter gets to zero.

The condition is \(\log_2\text{framesize}-1\), and the \(\log_2\text{framesize}\) parameter for the inside instances of the module is set to \(\log_2\text{framesize}-1\). When \(\log_2\text{framesize}\) gets to zero, the replacement block becomes a Gain primitive with unity gain (which of course has no effect). This terminates the recursion.

The \texttt{bit\_reverse} module performs the sort of data manipulation that is at the heart of the decimation-in-time FFT algorithm. See [Lee94] for an implementation of that algorithm using these same techniques (or see the demos).

21.6 Bus manipulation primitives

One consequence of the introduction of higher-order functions into MLDesigner is that busses have suddenly become much more useful than they used to be.

Fortunately, while increasing the demand for busses, higher-order functions also provide a cost effective way to manipulate busses. Like the Map primitive and its variants, the bus manipulation primitives in the HOF domain modify the graph at pre-initialization time and then self-destruct. Thus, they can operate in any domain, and they introduce no run-time overhead.

An example of the use of the BusMerge primitive is shown in fig. 21.9. The BusMerge primitive rewrites the graph at pre-initialization time and then self-destructs. Thus, it introduces zero run-time overhead. A good example can be found in graph.

fifth spot. As a final example, the seventh value (16) has address 6 which is \texttt{110} in binary. Reversed, its binary value is \texttt{011} which is 3 and the value (16) is output forth. After the first 8 values are read, the cycle is repeated for the next 8 values.
21 HOF Domain

MLD Libraries/Demos/HOF Demo/sdf.
The system fft2d shows how versatile the bus mechanism can be.

![Diagram](image_url)

Figure 21.9: System with BusMerge instances

### 21.6.1 NOP Primitives

A more interesting bus manipulation primitive is the **Nop** primitive, so called because it really performs no function at all. It can have any number of inputs, but the number of outputs must be the same as the number of inputs. All it does is connect its inputs to its outputs (at pre-initialization time) and then self-destruct.

The icon on the left has three individual input ports, and simply combines them into an output multiporthole. This multiporthole would normally be connected to a bus, which must be of width three. Thus, this icon provides a way to create a bus from individual connections. The next icon is similar, except that it has five input lines. The next two icons do the reverse. They are used to break out a bus into its individual components.

Examples of the uses of **Nop** primitives are shown in fig. 21.10. Three signals are individually generated at the left by three different source primitives. These signals are then combined into a bus of width three using a **Nop** primitive. The bus is then broken out into three individual lines, which are fed to three **Gain** primitives. The most interesting use of the **Nop** primitive, however, is the one on the right. The **XMgraph** primitive shown there has a multiporthole input. The **Nop** primitive is simply deposited on top of the multiporthole to provide it with three individual inputs. Why do this? Because when connecting multiple signals to a multiporthole input, it is difficult to control which input line goes to which specific porthole in the multiporthole set. Putting the **Nop** primitive on the porthole gives us this control with no additional runtime cost.

Recall that many primitives in MLDesigner, that have multiportholes, have special primitives that define the number of input or output ports that are visible in a design. The long lists of primitives have been replaced with a Select Special Primitive dialog. When you drag a primitive with multiple ports into a system or module you have the option to choose from a number of predefined port layouts. If the particular primitive you need is not in the list, you can create a special primitive by selecting the appropriate menu option from the context menu of the primitive in question.
21.7 An overview of the HOF primitives

21.7.1 Bus manipulation primitives

The top group in the main HOF library are the bus manipulation primitives, summarized below:

**BusMerge**
Bridge inputs to outputs and then self-destruct. This primitive merges two input busses into a single bus. If the input bus widths are $M_1$ and $M_2$, and the output bus width is $N$, then we require that $N = M_1 + M_2$. The first $M_1$ outputs come from the first input bus, while the next $M_2$ outputs come from the second input bus.

**BusSplit**
Bridge inputs to outputs and then self-destruct. This primitive splits an input bus into two. If the input bus width is $N$, and the output bus widths are $M_1$ and $M_2$, then we require that $N = M_1 + M_2$. The first $M_1$ inputs go the first output bus, while the next $M_2$ inputs go to the second output bus.

**BusInterleave**
Bridge inputs to outputs and then self-destruct. This primitive interleaves two input busses onto a single bus. The two input busses must have the same width, which must be half the width of the output bus. The input signals are connected to the output in an alternating fashion.

**BusDeinterleave**
Bridge inputs to outputs and then self-destruct. This primitive de-interleaves a bus, producing two output busses of equal width. The input bus must have even width. The even numbered input signals are connected to the first output bus, while the odd numbered input signals are connected to the second output bus.
Nop  Bridge inputs to outputs and then self-destruct. This primitive is used to split a bus into individual lines or combine individual lines into a bus. It is also used to break out multi-inputs and multi-outputs into individual ports. These icons are labeled BusCreate and BusSplit, suggesting their usual function.

### 21.7.2 Map-like primitives

**Map**  Map one or more instances of the named block to the input stream(s) to produce the output stream(s). This is implemented by replacing the Map primitive with one or more instances of the named block at pre-initialization time. The replacement block(s) are connected as specified by input_map and output_map, using the existing connections to the Map primitive. Their parameters are determined by parameter_map.

**Src**  This is identical to the Map primitive, except that the replacement block is a source block (it has no inputs).

**MapGr**  A variant of the Map primitive where the replacement block is specified by graphically connecting it. There must be exactly one block connected in the position of the replacement block. The Nop primitives are the only exception: they may be used in addition to the one replacement block in order to control the order of connection.

**SrcGr**  This is identical to the MapGr primitive, except that the replacement block is a source block (it has no inputs).

**Chain**  Create one or more instances of the named block connected in a chain. This is implemented by replacing the Chain primitive with instances of the named blocks at pre-initialization time. The replacement block(s) are connected as specified by input_map, internal_map, and output_map. Their parameters are determined by parameter_map. If pipeline is YES, then a unit delay is put on all internal connections.

**IfElse**  This primitive is just like Map, except that it chooses one of two named blocks to replace itself. If the condition parameter is TRUE, then the true_block is used. Otherwise, the false_block is used. This can be used to parameterize the use of a given block, or, more interestingly, for statically evaluated recursion.

**IfElseGr**  A variant of the IfElse primitive where the two possible replacement blocks are specified graphically rather than textually. There must be exactly one block connected in the position of each of the two replacement blocks. The Nop primitives are the only exception: they may be used in addition to the two replacement blocks in order to control the order of connection. As of this writing, this primitive cannot be used with recursion.
21.8  An overview of HOF demos

The HOF demos at MLD_Libraries/HOF Domain/Demos are divided by domain. As of this writing, only the SDF, DE, and CGC domains have demos in this library.

21.8.1  HOF demos in the SDF domain

addingSinWaves  This demo generates a number of sine waves given by the parameter number_of_sine_waves and adds them all together. The amplitude of each sine wave is controlled by a Tk slider that is inserted into the control panel when the system is run. The frequency in radians of each sine wave (relative to a sample rate of $2\pi$) is instance_number multiplied by $\pi/32$. Thus, the first sine wave will have a period of 64 samples. The second will have a period of 32. The third will have a period of 16, etc. The sum of these sine waves is displayed in bar-graph form.

busManipulations  This demo illustrates the use of the Nop primitive for manipulating busses and multiportholes. Note the combination of signals into a bus and the breakout of the multiporthole on the XMgraph.

cascadedBiquads  The Chain HOF primitive is used to construct a cascade of two second-order direct-form recursive filters (biquads). The frequency response of the cascade is compared against the frequency response of a direct-form fourth-order filter with the same transfer function.

fft  This system implements a recursive definition of a decimation-in-time fast Fourier transform, comparing its output against that of a direct implementation in C++. The system is configured to use 32 point FFTs to implement a 256 point FFT. The granularity is controllable with the parameters, and can be taken all the way down to the level of multipliers and adders. This system is discussed in detail in [Lee94].

fft2d  This system generates a square as in the square demo, and then computes its two-dimensional FFT.

fourierSeries  This system generates a number of sinusoids as given by the number_of_terms parameter. These are then weighted by the appropriate Fourier series coefficients so that the sum of the sinusoids gives the finite Fourier series approximation for a square wave with period given by the period parameter.

fourierSeriesMma  This system is similar to the fourierSeries system above, but uses Mathematica to calculate parameter values. Mathematica must be licensed on the local workstation for this demo to run.

phased_array  This system models a planar array of sensors with beamforming and steering, such as might be used with a microphone array or a radar system. The sensors can be positioned arbitrarily in a plane. With the default parameters, 16 sensors are uniformly spaced along the vertical
axis, half a wavelength apart, except for one, the fourth, which is offset along the horizontal axis by one tenth of a wavelength. The gain of the array as a function of direction is plotted in both polar and rectangular form (the latter in dB). A Hamming window is applied to the sensor data, as is a steering vector which directs the beam downwards. Zoom into the center of the polar plot to see the effect of the offset sensor. Try changing the parameter_map of the left-most MapGr higher-order function to realign the offset sensor, and observe the effect on the gain pattern.

RadarChainProcessing  This system simulates radar without beamforming. In this simulation, we simulate the effect of an electromagnetic signal traveling from a transmitter to targets and going back to receivers. The delay of the returned signal is used to provide information on the range of the target. The frequency shift, or Doppler effect, is used to provide information on the speed of the target. Thus, with these parameters, we estimate the targets properties as in a narrow band radar. The system has been converted from a data parallel form that uses a five-dimensional data array to a functional parallel form that uses higher-order functions to produce streams of streams. The five dimensions are range bin, doppler filters, number of sensors, number of targets and number of pulses.

sawtooth  This demo generates increasing sawtooth waveforms by interleaving pairs of ramps with slopes differing by one.

scramble  This system demonstrates the bit_reverse module shown above in fig. 21.8 and explained in the accompanying text.

square  This system demonstrates the BusMerge HOF primitive. It generates an image consisting of a light square on a dark background. The image is first represented using a bus, where each connection in the bus represents one row. The Commutator primitive then rasterizes the image.

wildColors  Creates a number of random sequences and plots them in a pair of bar graphs.

### 21.8.2 HOF demos in the DE domain

At this time, there are only two simple demos in the DE domain. These can be found in the Library view MLD Libraries/Demos/HOF Demo/de

poisson  This system generates any number of Poisson processes (default 10) and displays them together. To distinguish them, each process produces events with a distinct value.

exponential  Combine a number of Poisson processes and show that the inter-arrival times are exponentially distributed by plotting a histogram. Notice that the histogram bin centered at zero is actually only half as a wide
as the others (since the inter-arrival time cannot be negative), so the histogram displays a value for the zero bin that is half as high as what would be expected.

### 21.8.3 HOF demos in the CGC domain

These Demos are not supported at present and may contain bugs.

- **busses**
  Create a set of ramps of different slopes and display them in both a bar chart and using pxgraph.

- **scrambledCGC**
  This system demonstrates recursion in code generation by taking a ramp in and reordering samples in bit-reversed order.

- **soundHOF**
  This system produces a sound made by adding a fundamental and its harmonics in amounts controlled by sliders.

- **wildColorsCGC**
  This system is a CGC version of the SDF demo wildColors. It creates a number of random sequences and plots them in a pair of bar graphs.
Chapter 22

DE Domain

22.1 Introduction

The discrete event (DE) domain in MLDesigner provides a general environment for time-oriented simulations of systems such as queuing networks, communication networks, and high-level models of computer architectures. In this domain, each Particle represents an event that corresponds to a change of the system state. The DE schedulers process events in chronological order. Since the time interval between events is generally not fixed, each particle has an associated time stamp. Time stamps are generated by the block producing the particle based on the time stamps of the input particles and the latency of the block.

22.2 The DE target and its schedulers

The DE domain, at this time, has only one target. This target has three parameters:

- **timeScale** (FLOAT) Default = 1.0
  A scaling factor relating local simulated time to the time of other domains that might be communicating with DE.

- **usedScheduler** (Enum) Default = Priority Free Scheduler
  Indicates which scheduler to use to simulate the model: Calendar Queue Scheduler, Mutable Calendar Queue Scheduler, Priority Free Scheduler, Priority Scheduler, Resource Contention Scheduler, Simple DE Scheduler.

The DE schedulers in MLDesigner determine the order of execution of the blocks. There are six schedulers that have been implemented which are distributed with the domain. They expect particular behavior (operational semantics) on the part of the primitives. In this section, we describe the semantics.
22 DE Domain

22.3 Events and chronology

A DE primitive models part of a system response to a change in the system state. The change of state, which is called an event, is signaled by a particle in the DE domain. Each particle is assigned a time stamp indicating when (in simulated time) it is to be processed. Since events are irregularly spaced in time and system responses are generally very dynamic, all scheduling actions are performed at run-time. At run-time, the DE scheduler processes the events in chronological order until simulated time reaches a global "stop time."

Each scheduler maintains a global event queue where particles currently in the system are sorted in accordance with their time stamps; the earliest event in simulated time being at the head of the queue. The difference between the two schedulers is primarily in the management of this event queue. The default DE Scheduler mechanism handles large event queues much more efficiently than the alternative, a more direct DE scheduler, which uses a single sorted list with linear searching. The alternative scheduler can be selected by changing a parameter in the default DE target.

Each scheduler fetches the event at the head of the event queue and sends it to the input ports of its destination block. A DE primitive is executed (fired) whenever there is a new event on any of its input portholes. Before executing the primitive, the scheduler searches the event queue to find out whether there are any simultaneous events at the other input portholes of the same primitive, and fetches those events. Thus, for each firing, a primitive can consume all simultaneous events for its input portholes. After a block is executed it may generate some output events on its output ports. These events are put into the global event queue. Then the scheduler fetches another event and repeats its action until the given stopping condition is met.

It is worth noting that the particle movement is not through Geodesics, as in most other domains, but through the global queue in the DE domain. Since the geodesic is a FIFO queue, we cannot implement the incoming events which do not arrive in chronological order if we put the particles into geodesics. Instead, the particles are managed globally in the event queue.

22.4 Event generators

Some DE primitives are event generators that do not consume any events, and hence cannot be triggered by input events. They are first triggered by system-generated particles that are placed in the event queue before the system is started. Subsequent firings are requested by the primitive itself, which gives the time at which it wishes to be refired. All such primitives are derived from the base class RepeatStar.

RepeatStar is also used by primitives that do have input portholes, but also need to schedule themselves to execute at particular future times whether or not any outside event will arrive then. An example is PSServer.

In a RepeatStar, a special hidden pair of input and output ports is created and connected together. This allows the primitive to schedule itself to execute at any desired future time(s), by emitting events with appropriate time stamps on the feedback loop port. The hidden ports are in every
way identical to normal ports, except that they are not visible in the graphical user interface. The programmer of a derived primitive sometimes needs to be aware that these ports are present. For example, the primitive must not be declared to be a delay primitive (meaning that no input port can trigger a zero-delay output event) unless the condition also holds for the feedback port (meaning that refire events don’t trigger immediate outputs either). See the Programmer’s Manual for more information on using RepeatStar.

22.5 Simultaneous events

A special effort has been made to handle simultaneous events in a rational way. As noted above, all available simultaneous events at all input ports are made available to a primitive when it is fired. In addition, if two distinct primitives can be fired because they both have events at their inputs with identical time stamps, some choice must be made as to which one to fire. A common strategy is to choose one arbitrarily. This scheme has the simplest implementation, but can lead to unexpected and counter-intuitive results from a simulation.

The choice of which to fire is made in MLDesigner by statically assigning priorities to the primitives according to a topological sort. Thus, if one of two enabled primitives could produce events with zero delay that would affect the other, as shown in fig. 22.1, then that primitive will be fired first. The topological sort is actually even more sophisticated than we have indicated. It follows triggering relationships between input and output portholes selectively, according to assertions made in the primitive definition. Thus, the priorities are actually assigned to individual portholes, rather than to entire primitives. See the Programmer’s Manual for further details.

![Figure 22.1:](image)

Figure 22.1: When DE primitives are enabled by simultaneous events, the choice of which to fire is determined by priorities based on a topological sort. Thus if B and C both have events with identical time stamps, B will take priority over C. The delay on the path from C to A serves to break the topological sort.

There is a pitfall in managing time stamps. Two time stamps are not considered equal unless they are exactly equal, to the limit of double-precision floating-point arithmetic. If two time stamps were computed by two separate paths, they are likely to differ in the least significant bits, unless all values in the computation can be represented exactly in a binary representation. If simultaneity is critical in a given application, then exact integral values should be used for time stamps. This will work reliably as long as the integers are small enough to be represented exactly as double-precision values. Note that the DE domain does not enforce integer timestamps – it is up to the primitives being used to generate only integer-valued event timestamps, perhaps by rounding off their calculated output event times.
22.6 Delay-free loops

Many primitives in the DE domain produce events with the same time stamps as their input events. These zero-delay primitives can create some subtleties in a simulation. An event-path consists of the physical arcs between output portholes and input portholes plus zero-delay paths inside the primitives, through which an input event instantaneously triggers an output event. If an event-path forms a loop, we call it a delay-free loop. While a delay-free loop in the SDF domain results in a deadlock of the system, a delay-free loop in the DE domain potentially causes unbounded computation. Therefore, it is advisable to detect the delay-free loop at compile time. If a delay-free loop is detected, an error is signaled.

Detecting delay-free loops reliably is difficult. Some primitives, such as Server and Delay, take a parameter that specifies the amount of delay. If this is set to zero, it will fool the scheduler. It is the user’s responsibility to avoid this pathological case. This is a special case of a more general problem, in which primitives conditionally produce zero-delay events. Without requiring the scheduler to know a great deal about such primitives, we cannot reliably detect zero-delay loops. What appears to be a delay-free path can be safe under conditions understood by the programmer. In such situations, the programmer can avoid the error message placing a delay element on some arc of the loop. The delay element is the small green diamond found at the top of every primitive library. It does not actually produce any time delay in simulated time. Instead, it declares to the scheduler that the arc with the delay element should be treated as if it had a delay, even though it does not. A delay element on a directed loop thus suppresses the detection of a delay-free loop.

Another way to think about a delay marker in the DE domain is that it tells the scheduler that it’s OK for a particle crossing that arc to be processed in the “next” simulated instant, even if the particle is emitted with timestamp equal to current time. Particles with identical timestamps are normally processed in an order that gives data flow-like behavior within a simulated instant. This is ensured by assigning suitable firing priorities to the primitives. A delay marker causes its arc to be ignored while determining the data flow-based priority of primitive firing; so a particle crossing that arc triggers a new round of data flow-like evaluation.

22.7 Wormholes

"Time" in the DE domain means simulated time. The DE domain may be used in combination with other domains in MLDesigner, even if the other domains do not have a notion of simulated time. A given simulation, therefore, may involve several schedulers, some of which use a notion of simulated time, and some of which do not. There may also be more than one DE scheduler active in one simulation. The notion of time in the separate schedulers needs to be coordinated. This coordination is specific to the inner and outer domains of the wormhole. Important cases are described below.

22.7.1 SDF within DE

A common combination of domains pairs the SDF domain with the DE domain. There are two possible scenarios. If the SDF domain is inside the DE domain, as shown in fig. 22.2, then the SDF
22.7 Wormholes

subsystem appears to the DE system as a zero-delay block. Suppose, for example, that an event with time stamp $T$ is available at the input to the SDF subsystem. Then when the DE scheduler reaches this time, it fires the SDF subsystem. The SDF subsystem runs the SDF scheduler through one iteration, consuming the input event. In response, it will typically produce one output event, and this output event will be given the time stamp $T$.

If the SDF subsystem in fig. 22.2 is a multirate system, the effects are somewhat more subtle. First, a single event at the input may not be sufficient to cycle through one iteration of the SDF schedule. In this case, the SDF subsystem will simply return, having produced no output events. Only when enough input events have accumulated at the input will any output events be produced. Second, when output events are produced, more than one event may be produced. In the current implementation, all of the output events that are produced have the same time stamp. This may change in future implementations.

More care has to be taken when one wants an SDF subsystem to serve as a source primitive in a discrete-event domain. Recall that source primitives in the DE domain have to schedule themselves. One solution is to create an SDF “source” subsystem that takes an input, and then connect a DE source to the input of the SDF wormhole. We are considering modifying the wormhole interface to support mixing sources from different domains automatically.

22.7.2 DE within SDF

The reverse scenario is where a DE subsystem is included within an SDF system. The key requirement, in this case, is that when the DE subsystem is fired, it must produce output events, since these will be expected by the SDF subsystem. A very simple example is shown in fig. 22.3. The DE subsystem in the figure routes input events through a time delay. The events at the output of the time delay, however, will be events in the future. The Sampler primitive, therefore, is introduced to produce an output event at the current simulation time. This output event, therefore, is produced before the DE scheduler returns control to the output SDF scheduler. The behavior shown in fig. 22.3 may not be the desired behavior. The Sampler primitive, given an event on its control input (the bottom input), copies the most recent event from its data input (the left input) to the output. If there has been no input data event, then a zero-valued event is

Figure 22.2: When an SDF domain appears within a DE domain, events at the input to the SDF subsystem result in zero-delay events at the output of the SDF subsystem. Thus, the time stamps at the output are identically to the time stamps at the input.
Figure 22.3: A typical DE subsystem intended for inclusion within an SDF subsystem. When a DE subsystem appears within an SDF system, the DE subsystem must ensure that the appropriate number of output events are produced in response to input events. This is typically accomplished with a “Sampler” primitive.

produced. There are many alternative ways to ensure that an output event is produced. For this reason, the mechanism for ensuring that this output event is produced is not built in. The user must understand the semantics of the interacting domains, and act accordingly.

22.7.3 Timed domains within timed domains

The DE domain is a timed domain. Suppose it contains another timed domain in a DE wormhole. In this case, the inner domain may need to be activated at a given point in simulated time even if there are no new events on its input portholes. Suppose, for instance, that the inner domain contains a clock that internally generates events at regular intervals. Then these events need to be processed at the appropriate time regardless of whether the inner system has any new external stimulus.

The mechanism for handling this situation is simple. When the internal domain is initialized or fired, it can, before returning, place itself on the event queue of the outer domain, much the same way that an event generator primitive would. This ensures that the inner event will be processed at the appropriate time in the overall chronology. Thus, when a timed domain sits within a timed domain wormhole, before returning control to the scheduler of the outer domain, it requests rescheduling at the time corresponding to the oldest time stamp on its event queue, if there is such an event.

When an internal timed domain is invoked by another time domain, it is told to run until a given "stop time," usually the time of the events at the inputs to the internal domain that triggered the invocation. This "stop time" is the current time of the outer scheduler. Since the inner scheduler is requested to not exceed that time, it can only produce events with time stamp equal to that time. Thus, a timed domain wormhole, when fired, will always either produce no output events, or produce output events with time stamp equal to the simulated time at which it was fired.

To get a time delay through such a wormhole, two firings are required. Suppose the first firing is triggered by an input event at time $T$, then the inside system generates an internal event at a future time $T + \tau$. Before returning control to the outer scheduler, the inner scheduler requests that it be reinvoked at time $T + \tau$. When the "current time" of the outer scheduler reaches $T + \tau$, it reinvokes the inner scheduler, which then produces an output event at time $T + \tau$. 
With this conservative style of timed interaction, we say that the DE domain operates in the **synchronized mode**. Synchronized mode operation suffers significant overhead at run time, since the wormhole is called at every time increment in the inner timed domain. Sometimes, however, this approach is too conservative.

In some applications, when an input event arrives, we can safely execute the wormhole into the future until either (a) we reach the time of the next event on the event queue of the outer domain, or (b) there are no more events to process in the inner domain. In other words, in certain situations, we can safely ignore the request from the output domain that we execute only up until the time of the input event. As an experimental facility to improve run-time efficiency, an option avoids synchronized operation. Then, we say that the DE domain operates in the **optimized mode**. We specify this mode by setting the target parameter `syncMode` to `FALSE` (zero). This should only be done by knowledgeable users who understand the DE model of computation very well. The default value of the `syncMode` parameter is `TRUE` (one), which means synchronized operation.

### 22.8 DE Performance Issues

DE Performance can be an issue with large, long-running systems. Below we discuss a few potential solutions.

The calendar queue scheduler is not always the one to use. It works well as long as the "density" of events in simulated time is fairly uniform. But if events are very irregularly spaced, you may get better performance with the simpler scheduler, because it makes no assumptions about timestamp values. For example, it has been reported that the CQ scheduler did not behave well in a simulation that had a few events at time zero and then the bulk of the events between times 800000000 and 810000000 – most of the events ended up in a single CQ "bucket", so that performance was worse than the simple scheduler.

It also has been pointed out that both the CQ and simple schedulers ultimately depend on simple linear lists of events. If your application usually has large numbers of events pending, it might be worth trying to replace these lists with genuine priority queues (i.e., heaps, with $O(\log N)$ rather than $O(N)$ performance). But you ought to profile first to see if that’s really a time sink.

Another thing to keep in mind that the overhead for selecting a next event and firing a primitive is fairly large compared to other domains such as SDF. It helps if your primitives do a reasonable amount of useful work per firing; that is, DE encourages "heavyweight" primitives. One way to get around this is to put purely computational subsystems inside SDF wormholes. As discussed previously, an SDF-in-DE wormhole acts as a zero-delay primitive.

One way to gain a slight amount of speed is to avoid the GUI interface entirely by using ptcl, which does not have Tk primitives.
22.9 DE Libraries

The model of computation in the DE domain makes it amenable to high-level system modeling. For this reason, primitives in the DE domain are often more complicated, and more specialized than those in the SDF domain.

We have made every attempt to include in the distribution all of the reasonably generic primitives that have been developed, plus a selection of the more esoteric ones (as examples). Keep in mind that the primitive libraries of the other domains are also available through the wormhole mechanism. Users that find themselves frequently needing primitives from other domains may wish to build a library of single-primitive modules. Such modules can be used in any domain, regardless of the domain in which the single primitive resides. MLDesigner automatically implements them as a wormhole.

22.10 Source primitives

Strictly speaking, source primitives are primitives with no inputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. By convention, these primitives are fired once at time zero automatically. During this and all subsequent firings, the primitive itself must determine when its next firing should occur. It schedules this next firing with a call to the method refireAtTime(time).

- **Clock**: Generate events at regular intervals, starting at time zero.
- **Impulse**: Generate a single event at time zero.
- **Null**: Do nothing. This is useful for connecting to unused input ports.
- **Poisson**: Generate events according to a Poisson process. The first event is generated at time zero. The mean inter-arrival time and magnitude of the events are given as parameters.
- **PulseGen**: Generate events with specified values at specified moments. The events are specified in the value array, which consists of time-value pairs, given in the syntax of complex numbers.
- **TclScript**: Invoke a Tcl script. The script is executed at the start of the simulation, from within the primitive’s begin method. It may define a procedure to be executed each time the primitive fires, which can in turn produce output events.
- **TkButtons**: Output the specified value when buttons are pushed. If the allow_simultaneous_events parameter is YES, the output events are produced only when the button labeled ”PUSH TO PRODUCE EVENTS” is pushed. The time stamps of each output event is set to the current time of the scheduler when the button is pushed.
- **TkSlider**: Output a value determined by an interactive on-screen scale slider.
For convenience, some primitives are included in the source library that are not really source primitives, in the above sense. They require an input event in order to produce an output. These are listed below. The value of the input event is ignored; it is only its time stamp that matters.

**Const**
Produce an output event with a constant value (the default value is zero) when stimulated by an input event. The time stamp of the output is the same as that of the input.

**Ramp**
Produce an output event with a monotonically increasing value when stimulated by an input event. The value of the output event starts at value and increases by step each time the primitive fires.

**RanGen**
Generate a sequence of random numbers. Upon receiving an input event, it generates a random number with uniform, exponential, or normal distribution, as determined by the distribution parameter. Depending on the distribution, other parameters specify either the mean and variance or the lower and upper extent of the range.

**singen**
Generate a sample of a sine wave when triggered. This DE module contains an SDF singen module (i.e., a wormhole).

**WaveForm**
Upon receiving an input event, output the next value specified by the array parameter value (default "1 -1"). This array can periodically repeat with any period, and you can halt a simulation when the end of the array is reached. The following table summarizes the capabilities:
The first line of the table gives the default settings. The array may be read from a file by simply setting value to something of the form <filename>.

<table>
<thead>
<tr>
<th>haltAtEnd</th>
<th>periodic</th>
<th>period</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>YES</td>
<td>0</td>
<td>The period is the length of the array</td>
</tr>
<tr>
<td>No</td>
<td>YES</td>
<td>N &gt; 0</td>
<td>The period is N</td>
</tr>
<tr>
<td>NO</td>
<td>NO</td>
<td>anything</td>
<td>Output the array once, then zeros</td>
</tr>
<tr>
<td>YES</td>
<td>anything</td>
<td>anything</td>
<td>Stop after outputting the array once</td>
</tr>
</tbody>
</table>

Table 22.1: WaveForm capabilities

22.11 Sink primitives

The sink primitives have no outputs. They display signals in various ways, or write them to files. Several of the primitives in this library are based on the pxgraph program. This program has many options described in ch. 8.3. The differences between primitives often amount to little more than the choice of default options. Some, however, preprocess the signal in useful ways before passing it to the pxgraph program.
BarGraph

Generate a plot with the pxgraph program that uses a zero-order hold to interpolate between event values. Two points are plotted for each event, one when the event first occurs, and the second when the event is supplanted by a new event. A horizontal line then connects the two points. If draw_line_to_base is YES then a vertical line to the base of the bar graph is also drawn for each event occurrence.

Printer

Print the value of each arriving event, together with its time of arrival. The fileName parameter specifies the file to be written; the special names stdout and cout (specifying the standard output stream), and stderr and cerr (specifying the standard error stream), are also supported.

Xhistogram

Generate a histogram with the pxgraph program. The parameter binWidth determines the width of a bin in the histogram. The number of bins will depend on the range of values in the events that arrive. The time of arrival of events is ignored. This primitive is identical to the SDF primitive Xhistogram, but is used often enough in the DE domain that it is provided here for convenience.

XMgraph

Generate a plot with the pxgraph program with one point per event. Any number of event sequences can be plotted simultaneously, up to the limit determined by the XGraph class. By default, a straight line is drawn between each pair of events.

TclScript

Invoke a Tcl script. The script is executed at the start of the simulation, from within the primitive’s begin method. It may define a procedure to be executed each time the primitive fires, which can in turn read input events. There is a chapter of the Programmer’s Manual that explains how to write these scripts.

TkBarGraph

Take any number of inputs and dynamically display their values in bar-chart form.

TkMeter

Dynamically display the value of any number of input signals on a set of bar meters.

TkPlot

Plot Y input(s) vs. time with dynamic updating. Retracing is done to overlay successive time intervals, as in an oscilloscope. The style parameter determines which plotting style is used: dot causes individual points to be plotted, whereas connect causes connected lines to be plotted. The repeat_border_points parameter determines whether rightmost events are repeated on the left. Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.

TkShowEvents

Display input event values together with the time stamp at which they occur. The print method of the input particles is used to show the value, so any data type can be handled, although the space allocated on the screen may need to be adjusted.
22.12 Control primitives

Control primitives manipulate the flow of tokens. All of these primitives are polymorphic; they operate on any data type.

**Discard**
Discard input events that occur before the threshold time. Events after the threshold time are passed immediately to the output. This primitive is useful for removing transients and studying steady-state effects.

**Fork**
Replicate input events on the outputs with zero delay.

**LossyInput**
Route inputs to the sink output with the probability \( \text{lossProbability} \) set by the user. All other inputs are sent immediately to the save output.

**Merge**
Merge input events, keeping temporal order. Simultaneous events are merged in the order of the port number on which they appear, with port #1 being processed first.

**PassGate**
If the gate (bottom input) is open, then particles pass from input (left input) to output. When the gate is closed, no outputs are produced. If input particles arrive while the gate is closed, the most recent one will be passed to output when the gate is reopened.

**Router**
Route an input event randomly to one of its outputs. The probability is equal for each output. The time delay is zero.

---

**TkShowValues**
Display the values of the inputs in textual form. The print method of the input particles is used, so any data type can be handled, although the space allocated on the screen may need to be adjusted.

**TkStripChart**
Display events in time, recording the entire history. The supported styles are **hold** for zero-order hold, **connect** for connected dots, and **dot** for unconnected dots. An interactive help window describes other options for the plot.

**TkText**
Display the values of the inputs in a separate window, keeping a specified number of past values in view. The print method of the input particles is used, so any data type can be handled.

**TkXYPlot**
Plot \( Y \) input(s) vs. \( X \) input(s) with dynamic updating. Time stamps are ignored. If there is an event on only one of a matching pair of \( X \) and \( Y \) inputs, then the previously received value (or zero if none) is used for the other. The style parameter determines which plotting style is used: dot causes individual points to be plotted, whereas connect causes connected lines to be plotted.
Drawing a box in the plot will reset the plot area to that outlined by the box. There are also buttons for zooming in and out, and for resizing the box to just fit the data in view.

**Beep**
Cause a beep on the terminal when fired.
**Sampler**
Sample the input at the times given by events on the clock input. The data value of the clock input is ignored. If no input is available at the time of sampling, the latest input is used. If there has been no input, then a zero particle is produced. The exact meaning of zero depends on the particle type.

**LeakBucket**
Discard inputs that arrive too frequently. That is, any input event that would cause a queue of a given size followed by a server with a given service rate to overflow are discarded. Inputs that are not discarded are passed immediately to the output.

**Case**
Switch input events to one of \( N \) outputs, as determined by the last received control input. The value of the control input must be between 0 and \( N - 1 \), inclusive, or an error is flagged.

**EndCase**
Select an input event from one of \( N \) inputs, as specified by the last received control input. The value of the control input must be between 0 and \( N - 1 \) inclusive, or an error is flagged.

### 22.13 Conversion primitives

This library is intended to house a collection of primitives for format conversions of various types. As of this writing, however, this collection is very limited. The first two primitives in the conversion library illustrate the consolidation of multiple data sample into single particles that can be transmitted as a unit. These primitives use the class FloatVecData, which is simply a vector of floating-point numbers.

**Packetize**
Convert a number of floating-point input samples into a packet of type FloatVecData. A packet is produced when either an input appears on the demand input or when maxLength data values have arrived. Note that a null packet is produced if a demand signal arrives and there is no data.

**UnPacketize**
Convert a packet of type FloatVecData into a number of floating-point output samples. The data input feeds packets to the primitive. Whenever a packet arrives, the previous packet, if any, is discarded; any remaining contents are discarded. The demand input requests output data. If there is no data left in the current packet, the last output datum is repeated (zero is used if there has never been a packet). Otherwise the next data value from the current input packet is output.

**MxtoImage**
Convert a Matrix to a GrayImage output. The double values of the FloatMatrix are converted to the integer values of the GrayImage representation.

**ImageToMx**
Accept a black-and-white-image from an input image packet, and copy the individual pixels to a FloatMatrix. Note that even though the GrayImage input contains all integer values, we convert to a FloatMatrix to allow easier manipulation.
This library contains primitives that model queues, servers, and time delays of various types. In the DE domain, the delay icon (the small green diamond) does not represent a time delay.

**Delay**
Send each input event to the output with its time stamp incremented by an amount given by the delay parameter.

**VarDelay**
Delay the input by a variable amount. The delay parameter gives the initial delay, and the delay is changed using the newDelay input.

**PSServer**
Emulate a deterministic, processor-sharing server. If input events arrive when it is not busy, it delays them by the nominal service time. If they arrive when it is busy, the server is shared. Hence prior arrivals that are still in service will be delayed by more than the nominal service time.

**Server**
Emulate a server. If input events arrive when it is not busy, it delays them by the service time (a constant parameter). If they arrive when it is busy, it delays them the service time plus however long it takes to become free of previous tasks.

**VarServer**
Emulate a server with a variable service time. If input events arrive when it is idle, they will be serviced immediately and will be delayed only by the service time. If input events arrive while another event is being serviced, they will be queued. When the server becomes free, it will service any events waiting in its queue.

**FIFOQueue**
Implement a first-in first-out (FIFO) queue with finite or infinite length. Events on the demand input trigger a dequeue on the outData port if the queue is not empty. If the queue is empty, then a demand event enables the next future inData particle to pass immediately to outData. The first particle to arrive at inData is always passed directly to the output, unless numDemandsPending is initialized to 0. If consolidateDemands is set to TRUE (the default), then numDemandsPending is not permitted to rise above one. The size of the queue is sent to the size output whenever an inData or demand event is processed. Input data that doesn’t fit in the queue is sent to the overflow output.

**FlushQueue**
Implement a FIFO queue that when full, discards all inputs until it empties completely.

**PriorityQueue**
Emulate a priority queue. Inputs have priorities according to the number of the input, with inData#1 having highest priority, inData#2 being next, etc. When a demand is received, outputs are produced by selecting first based on priority, and then based on time of arrival, using a FIFO policy. A finite total capacity can be specified by setting the capacity parameter to a positive integer. When the capacity is reached, further inputs are sent to the overflow output, and not
stored. The numDemandsPending and consolidateDemands parameters have the same meaning as in other queue primitives. The size of the queue is sent to the size output whenever an inData or demand event is processed.

**Stack**

Implement a stack with either finite or infinite length. Events on the "demand" input pop data from the stack to outData if the stack is not empty. If it is empty, then a demand event enables the next future inData particle to pass immediately to outData. By default, numDemandsPending is initialized to 1, so the first particle to arrive at inData is passed directly to the output. If consolidateDemands is set to TRUE (the default), then numDemandsPending is not permitted to rise above one. The size of the stack is sent to the size output whenever an inData or demand event is processed. Input data that doesn’t fit on the stack is sent to the overflow output.

The following primitive does not appear in the library.

**QueueBase**

Serve as the base class for FIFO and LIFO queues. This primitive is not intended to be used except to derive useful primitives. All inputs are simply routed to the "overflow" output. None are stored.

### 22.15 Timing primitives

This library contains primitives that are primarily concerned with either simulated or real time.

- **MeasureDelay**
  Measure the time difference between the first arrival and the second arrival of an event with the same value. The second arrival and the time difference are each sent to outputs.

- **MeasureInterval**
  The value of each output event is the simulated time since the last input event (or since zero, for the first input event). The time stamp of each output event is the time stamp of the input event that triggers it.

- **StopTimer**
  Generate an output at the stopTime of the DEScheduler under which this block is running. This can be used to force actions at the end of a simulation. Within a wormhole, it can used to force actions at the end of each invocation of the wormhole. An input event is required to enable the primitive.

- **Timeout**
  Detect time-out conditions and generate an alarm if too much time elapses before resetting or stopping the timer. Events arriving on the Set input reset and start the timer. Events arriving on the "Clear" input stop the timer. If no Set or Clear events arrive within timeout time units of the most recent Set, then that Set event is sent out the alarm output.

- **TimeStamp**
  The value of the output events is the time stamp of the input events.
22.16 Logic primitives

The time stamp of the output events is also the time stamp of the input events.

**Synchronize**
Hold input events until the time elapsed on the system clock since the start of the simulation is greater than or equal to their time stamp. Then pass them to the output.

**Timer**
Upon receiving a trigger input, output the elapsed real time in seconds, divided by timeScale, since the last reset input, or since the start of the simulation if no reset has been received. The time stamp of the output is the same as that of the trigger input. The time in seconds is related to the scheduler (simulated) time through the scaling factor timeScale.

The following primitive does not appear in the library, because it is not intended to be used directly in MLDesigner applications.

**TimeoutPrimitive**
Serve as the base class for primitives that check time-out conditions. The methods set, clear and expired are provided for setting and testing the timer.

22.16 Logic primitives

**Logic**
Apply a logical operation to any number of inputs. The inputs are integers interpreted as Booleans, where zero is a FALSE and nonzero is a TRUE. The logical operations supported are \{NOT AND NAND OR NOR XOR XNOR\}.

**TestLevel**
Detect threshold crossings if the crossingsOnly parameter is TRUE. Otherwise, it simply compares the input against the threshold.
If crossingsOnly is TRUE, then: (1) a TRUE is sent to "output" when the "input" particle exceeds or equals the "threshold" value, having been previously smaller; (2) a FALSE is sent when the "input" particle is smaller than "threshold" having been previously larger. Otherwise, no output is produced.
If crossingsOnly is FALSE, then a TRUE is sent to output whenever any input particle greater than or equal to threshold is received, and a FALSE is sent otherwise.

**FlipFlop Primitives**
Binary state is afforded in the DE logic library with the inclusion of flip flop circuits. Three synchronous sequential circuit components, FlipFlopJK, FlipFlopT and FlipFlopD, serve as basic memory elements.

22.17 Networking primitives

This library includes primitives that have been designed to model communication networks. These are illustrative of a common use of the DE domain, for modeling packet-switched networks. How-
ever, many of the primitives are specialized to a particular type of network design. Thus, they should be viewed as illustrative examples, rather than as a comprehensive library.

A NetworkCell class is used in many of these primitives. It models packetized data that is transmitted through cell-relay networks. Each NetworkCell object can carry any user data of type Message. In addition to this user data, the NetworkCell contains a destination address and a priority. These are used by primitives and modules to route the cell through the network. The definition of the NetworkCell class may be found in

$MLD/src/domains/sdf/image/kernel,
since it is used in the SDF and DE domains, and was developed primarily for modeling packet-switched video.

### 22.17.1 Cell creation and access

- **CellLoad**
  - Read in an Envelope, extract its Message, and output that Message in a NetworkCell. Append a destination and priority to the packet.

- **CellUnload**
  - Remove a Message from a NetworkCell.

- **ImageToCell**
  - Packetize an image. Each image is divided up into chunks no larger than CellSize. Each cell is delayed from its predecessor by TimePerCell. If a new input arrives while an older one is being processed, the new input is queued.

- **CellToImage**
  - Read NetworkCell packets containing image data and output whole images. The current image is sent to the output when the primitive reads image data with a higher frame id than the current image. For each frame, the fraction of input data that was lost is sent to the "lossPct" output.

### 22.17.2 Cell routing, control, and service

- **CellRoute**
  - Read NetworkCell packets from multiple input sources and route them to the appropriate output using a routing table that maps addresses into output ports.

- **PriorityCheck**
  - Read NetworkCell packets from multiple input sources. If the priority of an input NetworkCell is less than the most recent "priority" input, then the cell is sent to the "discard" output. Otherwise it is sent to the "output" port.

- **Switch4x4**
  - Implement a four-input, four-output network switch that can process objects of type NetworkCell, or any type derived from NetworkCell. Each NetworkCell object contains a destination address. This module uses the destination address as an index into its Routes array parameter to choose an output port over which the input object will leave. A prioritized queuing scheme is used.

- **VirtClock**
  - Read a NetworkCell. It identifies which virtual circuit number the cell belongs to and then computes the virtual time stamp for the cell
by applying the virtual clock algorithm (see the source code in $MLD/src/domains/de/primitives/DEVirtClock.pl).

It then outputs all cells in order of increasing virtual time stamp. Upon receiving a “demand” input, the cell with the smallest time stamp is output. An output packet is generated for every demand input unless all of the queues are empty. Demand inputs arriving when all queues are empty are ignored. The number of stored cells is output after the receipt of each “input” or “demand.” When a cell arrives and the number of stored cells equals MaxSize then the cell with the biggest virtual time stamp is discarded. This cell may or may not be the new arrival. If MaxSize is zero or negative, then infinitely many cells can be stored.

22.17.3 Lost cell recovery

The primitives in this subgroup implement a variety of mechanisms for replacing lost cells in a packet-switched network. They use a class called SeqATMCell that is designed to model packets in the proposed broadband integrated services digital network (BISDN). The class is derived from Message, but has added facilities for marking the packet with a sequence number, and setting and reading individual bits. The sequence number is used to determine when packets have been lost.

PCMVoiceRecover Input a stream of SeqATMCell objects. All the information bits in objects received with correct sequence numbers are sent to output. If a missing SeqATMCell object is detected, this primitive sends the most recent $8 \times tempSize$ received bits to the temp output, and the most recent $8 \times searchWindowSize + numInfoBits$ received bits to the window output. The bits output on the window and temp outputs can be used by the PatternMatch module to implement lost-speech recovery.

SeqATMSub Read a sequence of SeqATMCells. It will check sequence numbers, and if a SeqATMCell is found missing, the information bits of the previously arrived SeqATMCell will be output in its place. The information bits from each correctly received SeqATMCell are unloaded and sent to the output port.

SeqATMZero Read a sequence of SeqATMCell objects. For each object input correctly in sequence, headerLength bits are skipped over and the next numInfoBits bits in the cell are output. If this primitive finds, by checking sequence numbers, that a SeqATMCell is missing, it will substitute numInfoBits 0-bits for the missing bits.

22.17.4 Wireless network simulation

Ether (Not shown in the library.) This is the base class for transmitter and receiver primitives that communicate over a shared medium. Each transmitter can communicate with any or all receivers that have the same
value for the "medium" parameter. The communication is accomplished without graphical connections, and the communication topology can be continually changing. This base class implements the data structures that are shared between the transmitters and receivers.

**EtherRec**

Receive floating-point particles transmitted to it by an EtherSend primitive. The particle is produced at the output after some duration of transmission specified at the transmitter.

**EtherRecMes**

See the explanation for the EtherRec primitive. The only difference is that this primitive forces the output to be a message.

**EtherSend**

Transmit particles of any type to any or all receivers that have the same value for the medium parameter. The receiver address is given by the "address" input, and it must be an string. If the string begins with a dash "-", then it is interpreted as a broadcast request, and copies of the particle are sent to all receivers that use the same medium. The transmitter "occupies" the medium for the specified duration. A collision occurs if the medium is occupied when a transmission is requested. In this case, the data to be transmitted is sent to the "collision" output.

### 22.18 Miscellaneous primitives

#### 22.18.1 Hardware modeling

**Arbitrate**

Act as a non-preemptive arbitrator, granting requests for exclusive control. If simultaneous requests arrive, priority is given to port A. When control is released, any pending requests on the other port will be serviced. The requestOut and grantIn connections allow interconnection of multiple arbitration primitives for more intricate control structures.

**HandShake**

Cooperate with a possibly preemptive arbitrator through the request and grant controls. Input particles are passed to output, and an ackIn particle must be received before the next output can be sent. This response is made available on ackOut.

**handShakeQ**

Handshake with queued input events.

**TclScript**

Invoke a Tcl script. The script is executed at the start of the simulation, from within the primitive’s begin method. It may define a procedure to be executed each time the primitive fires, which can in turn read input events and produce output events.

#### 22.18.2 Statistics and monitoring

**Statistics**

Calculate the average and variance of the input values that have arrived since the last reset. An output is generated when a demand input is
22.19 Multi-Valued Logic in DE Domain

The MultiValuedLogic (MVL) Library in the DE domain contains various sub-libraries that perform logic functions on data. The primitives in this library are based on four element logic that includes the two states 0 and 1 as well as the states X and Z.

The Value of the ErrorHandle parameter determines the action(s) that are taken by the primitive:

1. The primitive outputs a bit vector with X elements
2. A warning message is generated and a bit vector with X elements is output
3. An error message is generated and the primitive outputs nothing
4. The primitive generates an error message and the simulation is terminated.

22.20 An overview of DE demos

The number of DE demos is considerably smaller than SDF. Many of the demos, however, are much more complex, often incorporating SDF subsystems to accomplish audio or video encoding.

22.21 Basic demos

These demos illustrate the use of certain primitives without necessarily performing functions that are particularly interesting. The individual demos are summarized below.

- caseDemo Demonstrates the Case primitive by de-constructing a Poisson counting process into three subprocesses.
- conditionals Demonstrate the use of the Test block in its various configurations to compare the values of input events with floating-point values. The input test signal is a pair of ramps, with each event repeated once after some delay. Since the ramps have different steps, they will cross.
- logic Demonstrate the use of the Logic primitive in its various instantiations as AND, NAND, OR, NOR, XOR, XNOR and inverter gates. The three test signals consist of square waves with periods 2, 4, and 6.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>merge</strong></td>
<td>Demonstrate the Merge primitive. The primitive is fed two streams of regular arrivals, one a ramp beginning at 10.0, and one a ramp beginning at 0.0. The two streams are merged into one, in chronological order, with simultaneous events appearing in arbitrary order.</td>
</tr>
<tr>
<td><strong>realTime</strong></td>
<td>Demonstrate the use of the Synchronize and Timer blocks. Input events from a Clock primitive are held by the Synchronize primitive until their time stamp, multiplied by the system parameter timeScale, is equal to or larger than the elapsed real time since the start of the simulation. The Timer primitive then measures the actual (real) time at which the Synchronize output is produced. The closer the resulting plot is to a straight line with a slope of one, the more precise the timing of the Synchronize outputs are.</td>
</tr>
<tr>
<td><strong>router</strong></td>
<td>Randomly route an irregular but monotonic signal (a Poisson counting process) through two channels with random delay, and merge the channel outputs.</td>
</tr>
<tr>
<td><strong>sampler</strong></td>
<td>Demonstrate the Sampler primitive. A counting process with regular arrivals at intervals of 5.0 is sampled at regular intervals of 1.0. As expected, this produces 5 samples for each level of the counting process.</td>
</tr>
<tr>
<td><strong>statistics</strong></td>
<td>Compute the mean and variance of a random process using the Statistics primitive. The mean and variance are sent to the standard output when the simulation stops. This action is triggered by an event produced by the StopTimer primitive.</td>
</tr>
<tr>
<td><strong>switch</strong></td>
<td>Demonstrate the use of the Switch primitive. A Poisson counting process is sent to one output of the switch for the first 10 time units, and to the other output of the switch for the remaining time.</td>
</tr>
<tr>
<td><strong>testPacket</strong></td>
<td>Construct packets consisting of five sequential values from a ramp, send these packets to a server with a random service time, and then deconstruct the packets by reading the items in the packet one by one.</td>
</tr>
<tr>
<td><strong>timeout</strong></td>
<td>Demonstrate the use of the Timeout primitive. Every time unit, a timer is set. If after another 0.5 time units have elapsed, the timer is not cleared, an output is produced to indicate that the timer has expired. The signal that clears the timer is a Poisson process with a mean inter-arrival time of one time unit.</td>
</tr>
<tr>
<td><strong>upDownCount</strong></td>
<td>Demonstrate the UDCounter primitive. Events are generated at two different rates to count up and down. The up rate is faster than the down rate, so the trend is upwards. The value of the count is displayed every time it changes.</td>
</tr>
<tr>
<td><strong>binaryCounter</strong></td>
<td>Demonstrate the FlipFlopJK primitive.</td>
</tr>
<tr>
<td><strong>4BitDownCounter</strong></td>
<td>Demonstrate the use of the other Flip Flop primitives.</td>
</tr>
</tbody>
</table>
22.22 Queues, servers, and delays

The library of demos illustrating queuing systems includes:

- **blockage** Demonstrate a blocking strategy in a queuing network. In a cascade of two queues and servers, when the second queue fills up, it prevents any further dequeuing of particles from the first queue until it once again has space.

- **delayVsServer** Illustrate the difference between the Delay and Server blocks. The Delay passes the input events to the output with a fixed time offset. The Server accepts inputs only after the previous inputs have been served, and then holds that input for a fixed offset.

- **measureDelay** Demonstrate the use of the MeasureDelay block to measure the sojourn time of particles in a simple queuing system with a single server with a random service time.

- **priority** Demonstrate the use of the PriorityQueue block together with a Server. The upper input to the PriorityQueue has priority over the lower input. Thus, when the queue overflows, data is lost from the lower input.

- **psServer** Demonstrate the processor-sharing server. Unlike other servers, this server accepts new inputs at any time, regardless of how busy it is. Accepting a new input, however, slows down the service to all particles currently being served.

- **qAndServer** Demonstrate the use of the FIFOQueue and Stack primitives together with Servers. A regular counting process is enqueued on both primitives. The particles are dequeued whenever the server is free. The Stack is set with a larger capacity than the FIFO-Queue, so it overflows second. Overflow events are displayed.

- **queue** Demonstrate the use of the FIFOQueue and Stack primitives. A Poisson counting process is enqueued on both primitives, and is dequeued at a regular rate, every 1.0 time units. The output of the FIFOQueue is always monotonically increasing, because of the FIFO policy, but the output of the Stack need not be. The Stack is set with a smaller capacity than the FIFOQueue, so it overflows first. Overflow events are displayed.

- **testServers** Demonstrate servers with random service times (uniform and exponential).

22.23 Networking demos

A major application of the DE domain is the simulation of communication networks. The demos are:
FlushNet: Simulate a queue with “input flushing” during overflow. If the queue reaches capacity, all new arrivals are discarded until all items in the queue have been served.

LBTest: Simulate leaky bucket network rate controllers. These controllers moderate the flow of packets to keep them within specified rate and burstiness bounds.

VClock: Model a network with four inputs and virtual clock buffer service.

wirelessNetwork: Demonstrate shared media communication without graphical connectivity, using EtherSend and EtherRec primitives. Two clusters on the left transmit to two clusters on the right over two distinct media, radio and infrared. The communication is implemented using shared data structures between the primitives.

### 22.24 Miscellaneous demos

This library shows miscellaneous demos. The first two of these model continuous-time random processes, although only discrete-time samples of these processes can be displayed.

- **shotNoise**: Generate a continuous-time shot-noise process and display regularly spaced samples of it. The shot noise is generated by feeding a Poisson process into a Filter primitive.

- **hdShotNoise**: Generate a high-density shot noise process and verify its approximately Gaussian distribution by displaying a histogram.

The following demos illustrate the use of the DE domain for high-level modeling of protocols for sharing hardware resources.

- **roundRobin**: Simulate shared memory with round-robin arbitration at a high level.

- **prioritized**: Simulate a shared memory with prioritized arbitration at a high level.

The following demos have sound output.

- **speechcode**: Perform speech compression with a combination of silence detection, adaptive quantization, and adaptive estimation. After speech samples are read from a file, they are encoded, packetized, unpacketized, decoded, and played on the workstation speaker.

- **shave**: Demonstrate the Synchronize primitive to generate a beeping sound with a real-time rhythm.

### 22.25 Wormhole demos

This library shows some simple demonstrations of multiple domain simulations. Each of these combines SDF with DE.
distortion

Show the effects on real-time signals of a highly simplified packet-switched network. Packets can arrive out of order, and they can also arrive too late to be useful. In this simplified system, a sinusoid is generated in the SDF domain, launched into a communication network implemented in the DE domain, and compared to the output of the communication network. Plots are given in the time and frequency domains of the sinusoid before and after the network.

distortionQ

Similar to the distortion demo. The only difference is in the reorderQ wormhole, which introduces queuing.

worm

Show how easy it is to use SDF primitives to perform computation on DE particles. A Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples.

four_level

A four level SDF/DE/SDF/DE system.

sources

Show how to use an SDF primitive as a source by using a dummy input into the SDF system. The SDF subsystem fires instantaneously from the perspective of DE. The schedulePeriod SDF target parameter has no effect.

block

The schedulePeriod parameter of the SDF target determines how the inside of the DE system interprets the timing of events arriving from SDF. When several samples are produced in one iteration, as here, the time stamps of the corresponding events are uniformly distributed over the schedule period.

22.26 Tcl/Tk Demos

buttons

Demonstrate TkButtons by having the buttons generate events asynchronously with the simulation.

displays

Demonstrate some of the interactive displays in the DE domain.

slider

Demonstrate TkSlider by having the slider produce events asynchronously. The asynchronous events are plotted together with a clock, which produces periodic outputs in simulated time. Notice that the behavior is roughly the same regardless of the interval of the clock.

sources

A Tcl script writes asynchronously to its output roughly periodically in real time (using the Tk ”after” command). The asynchronous events are plotted together with a clock, which produces periodic outputs in simulated time. Notice that the plot looks roughly the same regardless of the interval of the clock.

stripChart

Demonstrate the TkStripChart by plotting several different sources.

xyplot

Display queue size as a function of time with an exponential random server. Note that the TkPlot primitive overlays the plots as time progresses, which the TkXYPlot primitive does not. Thus, the
points on the TkXYPlot primitive go off the screen to the right. The TkStripChart primitive records the entire history.
Chapter 23

CTDE Domain

23.1 Purpose of the domain

The purpose of the Continuous-Time and Discrete-Event (CTDE) domain is to simulate continuous-time and mixed signal systems and to combine these systems with existing MLDesigner domains.

23.2 Introduction to the CTDE domain

As the name CTDE Continuous Time/Discrete Event suggests, two distinct models of computation are combined into one domain. The next two sections describe each of the models of computation. These descriptions are followed by a discussion of how the two models are linked in the CTDE domain.

23.3 Continuous-Time Computation Models

23.3.1 Computation model

The characteristic property of continuous-time formalisms is that the trajectories of input-, output- and state-values are continuous in time. Mathematically, this class of systems is modeled by systems of ordinary differential equations (ODE). The change of a state variable is described by its time derivative

\[
\dot{x}(t) = f(x(t), u(t), t) \quad (23.1)
\]

\[
y(t) = g(x(t), u(t), t) \quad (23.2)
\]

A system model is described by its differential equation (23.1) and an output equation (23.2), where \(u(t)\) is the input, \(x(t)\) is the state and \(y(t)\) is the systems output.

23.3.2 Signal Form

At every instant of the simulation interval, inputs, outputs and states have a distinct value. Therefore, signals in continuous-time models are continuous functions. The resulting signal form is shown in Figure 23.1.
23.3.3 Example: Spring-Mass system

Continuous-Time models are often used to model mechanical or electrical systems. As an example, see the simple mechanical model in figure 23.2: Two bodies are connected by springs and dampers and accelerated by an external Force $F$.

Mathematically, this system is described by the following set of ordinary differential equations
(abbr. ODE):

\[
\begin{align*}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -\frac{k_1 + k_2}{m_1} x_1(t) - \frac{b_1 + b_2}{m_1} x_2(t) + \frac{k_2/m_1}{x_3(t)} x_3(t) + \frac{b_2}{m_1} x_4(t) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= \frac{k_2}{m_2} x_1(t) + \frac{b_2}{m_2} x_2(t) - \frac{k_1}{m_2} x_3(t) - \frac{b_1}{m_2} x_4(t) + \frac{1}{m_2} F(t)
\end{align*}
\]

Here, \(m_1\) and \(m_2\) are the masses of the bodies, \(k_1\) and \(k_2\) the spring constants and \(b_1\) and \(b_2\) the damping constants of the springs and dampers, respectively.

The simulation results are shown in figure 23.3.

![State trajectory of a spring-mass system](image)

**Figure 23.3:** Spring-Mass Example: Simulation results

### 23.3.4 Modeling

The CTDE domain uses the MLDesigner graphically oriented hierarchical block style of modeling. Models are created and manipulated as interconnected blocks, defining the dynamics of the system. Complex models can be organized hierarchically by combining blocks into submodules, submodules into modules and modules into systems. This simplifies the structure of complex models and increases usability of existing components. Besides the easy-to-use interface, the visual representation reflects the structure of the modeled system.

The state derivatives are represented by integrator blocks, while the derivative equation and the output function are modeled by networks of primitives that perform arithmetic operations. In figure 23.4, the CTDE model of the spring-mass-system is shown.

### 23.3.5 Simulation

Simulation of continuous-time models uses numerical methods for solving systems of ordinary equations, more precisely initial value problems. These algorithms are often called numerical
Figure 23.4: Example: CTDE Model of the Spring-Mass system
integration methods or ODE solver.
A numeric integration method starts with an initial value of \( x_0 \) and approximates the state and output values at a finite number of time points in the simulation interval. Numerical methods for solving ODE systems are an area of extensive research. A large number of different algorithms are available, with different levels of accuracy, computational effort and suitability for distinct classes of problems. The CTDE domain supports multiple integration algorithms and a variety of configuration options to support them.

23.3.6 Limitations of purely continuous-time models

The numerical methods mentioned above generally require that the differential equation \( f(x(t), u(t), t) \) and input trajectory \( u(t) \) to be smooth. More precisely, these functions must be sufficiently differentiable (depending on the used ODE solver.) Real systems seldom meet these requirements. One cause is discontinuity in the state transition function. (Systems with friction or hysteresis effects show such behavior.) Input signals often change their values discontinuously, especially in systems where continuous components interact with digital devices. Most numerical algorithms fail or have significantly reduced accuracy when stepping over discontinuity points. Therefore, practical simulation software must offer facilities to manage discontinuity points and handle them appropriately. This is typically done via breakpoint handling.

23.4 The Combined Continuous Time/Discrete Event Model of Computation

To overcome the limitations described above, the continuous-time model of computation is extended by adding Discrete-Event semantics.
The Discrete-Event-part of the CTDE-domain is similar to the computational model implemented in the MLDesigner DE domain.
Model components typically transfer information between different blocks as real-value-timestamped messages. These messages are also called discrete events and, therefore, Discrete Event models depict signals as sequences of events as shown in Figure 23.5.
Combined continuous/Discrete-Event models of computation form a well-defined and especially expressive extension of pure continuous-time formalisms. This computational model is usually referred to as Discrete-Event/Differential Equation Specified System (DEV&DESS) or Mixed-Signal simulation. For more information, see references [Cel79, ZPK00].

23.4.1 The CTDE Computational Model

As depicted in Figure 23.6, a combined model consists of a continuous and a Discrete-Event component. Both model parts may contain inputs, outputs and states of their respective types. The CT/DE combined model of computation explicitly defines the way these components influence each other. These interactions are also called events and can occur as time events, external input events or state events.
Figure 23.5: A Discrete-Event signal

Figure 23.6: Structure of a combined model
23.4 The Combined Continuous Time/Discrete Event Model of Computation

Time events are related to events in pure Discrete-Event models. They are initiated in the discrete part, but could possibly change the state in the continuous part. External input events are input messages that occur at the input ports of the Discrete-Event partition. They are similar to time events and are handled in the same way. State events are initiated by the continuous model. A state event is triggered when a condition that depends on continuous state or input values, is satisfied. In general, this condition can be expressed by a state event equation:

\[ C(u(t), x(t), t) = 0 \]  \hspace{1cm} (23.3)

Note that it is only possible to determine if a state event occurred within a specific time period after an integration-step algorithm has been completed because the event condition depends on values that change continuously. The scheduler in the CTDE domain ensures that the exact time of a state event is accurately determined, and the discrete and the continuous parts of the model are synchronized properly.

23.4.2 Model Structure

CT/DE models, primitives as well as modules and systems, may contain both continuous and discrete-event elements, and may have an arbitrary combination of discrete and continuous input and output ports. As an example of a general hybrid block, the limited integrator block is shown in Figure 23.7. Beside the continuous input and state output of a pure continuous integrator, it contains a discrete input for resetting the state value and a saturation output which signals reaching the upper or lower limit as a discrete event.

![Figure 23.7: The LimitedIntegrator primitive as example of a general hybrid block](image)

Pure continuous or discrete blocks are special cases of the general hybrid and can be modeled using this atomic element. This allows a uniform representation of all blocks in a combined model. The CTDE-domain supports two distinct signal forms. Continuous waveforms with value-semantics are passed between continuous time ports; messages, transmitted as discrete events, are passed between discrete event ports. Conversion blocks connect ports of unlike ports and convert the signals from one form to the other.
23 CTDE Domain

23.5 Modeling in the CTDE domain

The CTDE domain supports all the modeling features that are found in other domains. Models are represented by block diagrams; the block diagrams are assembled, the domains supports all the modeling features and model-building commands that are available in other domains.

The domain supports both continuous and discrete signals, but a direct connection between ports of different types is not allowed. Primitives in the Event Interpreters library convert discrete signals into continuous signals; primitives in the Event Generators library convert continuous signals into discrete signals.

23.5.1 Vectorial continuous signals

Continuous signals are vectors of real numbers. Many primitives can operate on vectorial inputs. For example, the add primitive can add an arbitrary number of vectors. The vectors can be of arbitrary width, but must all be the same size. The system checks to ensure that vector widths are consistent before the simulation starts.

Usage of vectorial signals often simplifies models and speeds up simulation. For example, in figure 23.8 the spring-mass system is modeled using a StateSpace primitive that operates on vectorial signals.

![Figure 23.8: Spring-Mass model using vectorial signals and a StateSpace Block](image)

23.5.2 Simulation Algorithm

The simulator is divided into a Discrete-Event and a Continuous-Time component much like the model structure. These parts interact as follows:

At a given time in the simulation interval the simulator starts in Discrete Event mode. The Discrete-Event scheduler processes all events with the current time stamp. When all events have been processed, the simulator shifts to continuous mode. In this mode, the trajectory of the continuous states is approximated by a numerical ODE solver. At the time of the next scheduled discrete event, execution shifts back to the Discrete-Event scheduler.

Detection and processing of state events requires special treatment. In the graphical model representation used in the CTDE domain, state event conditions are modeled as Blocks, called state event generators. Since state event conditions depend on continuous variables, they must be handled in the continuous part of the simulator. After each step of the ODE solver, the state event generator blocks are queried to determine if a state event occurred during the current step. If this
is the case, this integration step has to be rejected. Integration will be repeated with adjusted step sizes until the event condition is met with sufficient accuracy. When this occurred, the simulator immediately changes into the Discrete-Event mode and executes the action triggered by the state event as an ordinary discrete event.

23.6 Example: Bouncing Ball-Model

Here, we show an example commonly used for demonstrating the handling of state-events. A ball bounces repeatedly on a surface. The impact is modeled by the block ZeroCrossingDetector state event generator, which ensures that the time of the hit is determined accurately. This CTDE (figure 23.9) model shows the combination of continuous and discrete dynamics clearly. Whereas the movement of the ball is modeled with continuous elements and signals; the signaling of impact, calculation of the reflected velocity and inverting the movement is modeled using discrete logic.

Figure 23.9: Bouncing Ball System Model

Figure 23.10: Bouncing Ball output
23.7 User-adjustable parameters

The CTDE domain has several unique simulation parameters that can be edited in the model properties window.

23.8 The ODE solver

The CTDE domain provides several different mathematical methods for solving the ordinary differential equations (ODEs) used to simulate continuous or mixed-signal systems. (These methods are commonly called ODE solvers.) The ODE solver choice is user-selectable at execution time. Currently MLDesigner provides the four ODE solvers seen in table 23.1. Additional solvers will be added in future versions.

<table>
<thead>
<tr>
<th>abbr.</th>
<th>name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOPRI5</td>
<td>Dormand Prince</td>
<td>A Dormand-Prince method with-order 5(4), embedded error estimation, and variable step size control</td>
</tr>
<tr>
<td>FE</td>
<td>Forward Euler</td>
<td>One-step Euler-Cauchy Method</td>
</tr>
<tr>
<td>RK2</td>
<td>Runge-Kutta 2</td>
<td>Second order Runge-Kutta method</td>
</tr>
<tr>
<td>RK4</td>
<td>Runge-Kutta 4</td>
<td>Classical Runge-Kutta method of order 4</td>
</tr>
</tbody>
</table>

Table 23.1: ODE solvers

At the moment, the ODE solver choice has to be entered as text into the Solver edit box of the model properties window. A multiple choice dialog will be provided in a future version.

23.8.1 Solver parameters

Several parameters can be used to control the operation of the ODE solver. These parameters are listed table 23.2. (We assume that the reader is familiar with basic terms of ODE solvers shown in this table.)

**NOTE:** The meaning of some of these parameters may be different for different ODE solvers, and some parameters are not supported in some ODE solvers (and have no influence there).

Several other ODE solver parameters currently appear in the CTDE model properties window. These are artifacts of a previous version. They are not useful in this release and likely to change or to be removed in subsequent releases.
One of the most distinguishing features of MLDesigner is its ability to simulate heterogeneous systems, i.e. systems composed using different domains. The CTDE domain is designed to fit well into this concept.

The following example illustrates the concept. The model is a simple control system consisting of a plant process and a closed loop control system. The continuous process plant is modeled in the CTDE domain. The controlling algorithm is formulated as a data flow graph formulated in the SDF domain. The plant process model is shown in Figure 23.11.

![Plant Process model](image)

**Figure 23.11: Plant Process model**
The control system is depicted in figure 23.12.

![Control System model](image)

Figure 23.12: Control System model

Normally, one would use a ZeroOrderHold primitive to convert a Discrete-Event signal from the wormhole boundary into a continuous waveform. The opposite conversion would usually be done with a sampler block, e.g. a PeriodicSampler or a TriggeredSampler primitive. An output from the Xgraph block of the control system is shown in figure 23.13.

![Plant process output graph](image)

Figure 23.13: Plant process output graph showing the effects of the control system

## 23.10 Current limitations

The CTDE domain is in the early-stages of development and currently has several limitations:

- The suite of ODE solvers is currently limited; we intend to add a suite of more sophisticated numerical methods in a future release.
23.10 Current limitations

- The primitive library is incomplete and insufficiently generic. We will add primitives in future releases.

- CTDE works only as the outer domain in multi-domain systems today. Future releases will provide the ability to use the CTDE domain in arbitrary model hierarchies (e.g., embedded in any other domain.)

- The application programmers interface for writing user-defined primitives is not complete at the moment and likely to change.
Chapter 24

FSM Domain

24.1 What is a Finite State Machine?

A finite state machine is a conceptual machine with a finite number of states. It can be in only one of the states at any specific time. A state transition is a change in state that is caused by an input event. In response to any input event, the finite state machine might transition to a different state. Alternatively, the event has no effect and the finite state machine remains in the same state. The next state depends on the current state as well as on the input event. Optionally, an output action may result from the state transition. ¹

The MLDesigner finite state machine domain includes a graphical editor and an action language for defining and managing states, transitions and interface elements. It supports the UML Statechart semantic, hierarchical states and special events, as well as key MLDesigner features such as data types and data structures, shared memory, and interaction with other design domains.

24.2 The MLDesigner FSM Domain

The new MLDesigner FSM Domain offers the following features:

- Input Events
- Special Events
- Internal Events
- Memory arguments
- Parameter arguments
- FSM action language
- Hierarchical States
- State Entry Action
- State Exit Action
- State Slave Process
- Transition Event Expression
- Transition Guard Condition
- Transition Action
- Transition Preemptive Flag

¹from: “Designing Concurrent, Distributed and Real-Time Applications with UML” by Hassan Gomaa
24 FSM Domain

- Transition Entry Type
- static / recursive History States
- Default Entrance Action
- Support of the UML Statechart semantic

24.3 MLDesigner FSM Semantic

The FSM semantic provided by MLDesigner supports synchronous and asynchronous behavior, additional events, variables and parameters for various runs of simulations.

24.3.1 Basic FSM Elements

The FSM mechanism provided by MLDesigner supports all the basic standard elements of a finite state machine.

24.3.1.1 Events

Events are used as triggers to cause a finite state machine to undergo state changes. In doing so, the presence of an event is interpreted as logical true and the absence of an event as logical false. The MLDesigner FSM supports 3 different kinds of events:

Input Ports

The basic events are represented by data parsed into an input port of the FSM model interface.

Special Events

If an FSM model is embedded into a discrete event (DE) environment, MLDesigner Special Event arguments can be used to trigger the FSM model.

Internal Events

In context with the FSM state slave process mechanism, internal events can be set or reset inside an FSM model, without influence of the outer environment of the finite state machine. These internal events are represented by a boolean flag associated with the name of the event. If a state slave model contains an input port with the name of an internal event, this input port gets the current flag value (true or false) of the associated internal event, before the slave model executes. If a state slave model contains an output port with the name of an internal event and this output port contains new data after execution of the slave model, the associated internal event flag is set to the integer cast (unequal zero = true, equal zero = false) of the new data. If the output port, associated with an internal event, contains no new data after execution of the slave model, the internal event flag is reset to false.
24.3 MLDesigner FSM Semantic

24.3.1.2 States

States represent conditions or periods characterized by the concepts of duration and stability. A finite state machine can have an arbitrary number of states but at any time of execution, the FSM must reside in only one state.

Name

In the scope of a single FSM, each state of this FSM must have a unique name. This name is centered at the top of the rounded rectangle in the graphical notation of a state.

Hierarchical States

Each state can have an arbitrary number of sub-states and the sub-states can also be hierarchical. All states on the same level of a hierarchy are sibling states. States up the hierarchy are called ancestor states and states down the hierarchy are called descendant states. Leaf states are states without sub-states. In the graphical representation of hierarchical states, the borders of a sub-state must reside completely inside the boundaries of all ancestor states.

Current State

At any time during simulation, a finite state machine must reside in only one state. This state is called the current state. If the current state is a hierarchical state, then only one of its sub-states must be the current sub-state. This rule goes down the hierarchy until a leaf state is the current sub-state of a level of the hierarchy.

State Actions

Together with each state, it is possible to define two sets of operations. The entry action is executed whenever the state is entered and the exit action is a set of operations performed whenever the state is exited. These actions are defined using the C/C++ like FSM action language.

Slave Process

The MLDesigner FSM provides a slave process associated with leaf states. An MLDesigner module or a different FSM model can be used as a slave process. The slave process of the current state executes if the FSM receives new events and no preemptive transition, possessed by the current state, fires.

The following conventions must be met.

- Every input port of the slave process model must have the same name and data type as an input port of the superordinate FSM model, or the name of a slave process model input port must be the same as the name of an Internal Event of the superordinate FSM model.
- Every Output port of the slave process model must have the same name and data type as an output port of the superordinate FSM model, or the name of a slave process model output port must be equal to the name of an Internal Event of the superordinate FSM model.
- Every external memory argument of the slave process model must have the same name and data type as a memory argument of the superordinate FSM model.
• The *external* memory arguments of the slave process are linked to the associated memory arguments of the superordinate FSM model.

• The slave process model **must** have neither *internal* nor *external* special event arguments, as the execution of the slave process depends only on the execution of the superordinate FSM model.

### 24.3.1.3 Transitions

State changes in finite state machines are described via transitions. Each transition specifies a *source state* and a *target state*. The graphical notation is a line or multiple line segments between the source and target state with an arrow on one end, pointing to the target state.

![Figure 24.1: Transition with Label](image)

**Self Transition**

If the source and target state of a transition is the same state, the transition is called a *self transition*.

![Figure 24.2: An FSM Self Transition](image)

**Inherited Transitions**

Each state possesses all transitions for which it is the transition’s source state as well as those possessed by its ancestor states. Latter transitions are called *inherited transitions*.

**Preemptive Transitions**

Associated with each transition is a boolean property called *preemptive*. All the preemptive transitions, possessed by the current state, are checked for firing, before the *slave process* of the current state is performed. In that way, the slave process executes only, if no preemptive transition fires.
### Entry Type

In context with the FSM state slave process mechanism, every transition contains a property called *Entry Type*. If a transition fires, the entry type of this transition specifies in which way the slave process of the next current state will be executed. The entry type can be either *Default* or *History*. For example, a leaf state containing another FSM model as slave process. If this state is entered via a *Default* entry type transition, the slave process FSM model will always be reset to its initial state, before execution. In the case, this state is entered via a *History* entry type transition, the slave process FSM model continues the execution in the last current state.

### Event Expression

Each transition can have an event expression $E$ described by the grammar:

$$
S_e := \epsilon | E \\
E := e | \neg E | E \land E | E \lor E | (E) \\
e \in \text{Events}
$$

Events = \{\text{Input Ports, Special Events, Internal Events}\}

In doing so, complex and nested event expressions can be defined using brackets and the C/C++ logical operators \(!\) (NOT), \(||\) (OR) and \(\&\&\) (AND). For instance, an event expression consisting of two events, combined by a logical AND (Event1 \(\&\&\) Event2), only evaluates to *true* when both events are present. If instead two events are combined by a logical OR (Event1 \(||\) Event2), just one of them needs to be present for satisfaction.

**NOTE:** The applicability of the different logical operators within transition event expressions depends on the outer domain of the associated FSM model. See section 24.11.2 for details.

MLDesigner provides a special *Event Expression Dialog* (section 24.9.2) for easy event expression composition.

If the event expression evaluates to *true*, while the current state is a transition’s source state or one of its sub-states, the transition is triggered and becomes a candidate for firing.

A transition without an event expression ($E = \epsilon$) is immediately triggered after its source state entry action is executed. These transitions are called *synchronous transitions*.

### Guard Condition

Optionally associated with each transition is a guard condition $C$, specified by an FSM Action expression. If the guard condition of a triggered transition evaluates to *true*, the transition fires and the finite state machine’s next state becomes the transition’s target state.

A triggered transition without a guard condition fires immediately.

**NOTE:** Triggering does not automatically cause transitions to fire, it merely enables firing. If a triggered transition cannot fire, because the guard condition evaluates to *false*, the transition must be triggered again by a satisfied event expression to become a candidate for firing.
Transition Action

Each transition can also have an action $A$, which is a set of FSM Action statements. Whenever a transition fires, its action is performed before the transition’s target state entry action is executed.

Transition Label

Unlike states, transitions have no unique name. They are specified by an automatically generated label. The syntax of the label is: $E [C] / A$. The label is also visible in the transition properties window.

Transition Priority

If more than one transition possessed by the current state is triggered, the inherited transitions up the hierarchy have a higher priority to fire.

Transition Conflict

A transition conflict occurs, when two or more transitions with the same priority are triggered and no transition with a higher priority is candidate for firing. In this case of nondeterminism, the FSM scheduler fires the first one, which guard condition evaluates to true.

24.3.1.4 Default Entrances

The Default entrance is a special state which indicates the point of entry to that level of the state hierarchy. Each level of the state hierarchy, including the FSM top level, has one default entrance, which is depicted by a small solid circle. A default entrance must not have any incoming transitions and must have only one outgoing transition to designate the default sub-state destination. The top level default entrance designates the initial state of the finite state machine. The Top Level Default Entrance may be linked to a set of FSM Action statements that initialize the state machine (e.g. initialize memories.) Note: Upon simulation startup, the entry action of the initial state is not performed.

Figure 24.3: Default Entrance
24.3 MLDdesigner FSM Semantic

24.3.1.5 Histories

Histories are special states, used to resume the last sub-state of a hierarchical state. An FSM can have an arbitrary number of histories, placed at any level of the state hierarchy. The graphical notation is a circle containing either an $H$ for static or $H^*$ for recursive histories.

![Recursive(*) and Static History symbols]

A history must have at least one incoming transition and must not have any outgoing transitions.

Static History

A static history memorizes only the previous sub-state of its hierarchy level. If this sub-state is a hierarchical state, then its default entrance destination becomes the current sub-sub-state and so on, until a leaf state becomes the current state.

Recursive History

Recursive histories apply to all descendant states and refer to the previous current state of their state hierarchy. So the state, memorized by a recursive history, is always a leaf state on the same or lower level of the state hierarchy.

NOTE: If a transition, pointing to a history fires, actions are performed as if the transition is pointing to the state stored in the history. Trying to enter a hierarchical state via an empty history, when the hierarchical state was never visited before, results in an error being displayed and the simulation aborts.

24.3.1.6 Arguments

In addition to the basic elements, the MLDdesigner FSM semantic supports typical MLDdesigner arguments associated with finite state machines.

Memory Arguments

Memories can be used to represent variables in a finite state machine. The MLDdesigner FSM supports memories of both scopes: internal and external, and all types and data structures as is the case elsewhere in an MLDdesigner block. FSM action statements have read as well as write access to the value of a memory argument.

Special Event Arguments

In addition to events, represented by the inputs of a finite state machine, the MLDdesigner FSM supports special event arguments of both scopes, internal and external, and of all types and data structures. These events can also be used in the event expression of transitions and can be accessed
via FSM action statements to schedule or cancel an event argument. Events of external scope can
generate events in other MLDesigner models, i.e. other FSM models.

NOTE: Using MLDesigner special event arguments in context with finite state machines makes
sense only if the FSM model is embedded into a discrete event (DE) environment.

Parameter Arguments

Like memories and special events, parameters are fully supported in MLDesigner finite state ma-
chines. FSM action statements have only read access to the value of a parameter.

24.3.2 FSM Action Language

24.3.2.1 Overview

The actions associated with states, transitions and default entrances as well as transition guard
conditions are defined using statements of the FSM action language.
Likewise the source code to define the functionality of MLDesigner primitive models, the FSM
action language is based on the C/C++ syntax. Additionally, the FSM action language includes a
set of built-in functions, mainly to provide a comfortable handling of FSM interface elements and
MLDesigner data structures.

In doing so, the FSM action language supports the following operations:

- read access to data of input ports
- write access to data of output ports
- read and write access to data of memory arguments
- read and write access to data of event arguments
- schedule and cancel special event arguments
- read access to data of parameter arguments
- read and write access to data structure member fields
- relational and logical expressions
- math operations
- control structures and loops (if, else, switch, for, while, etc.)
- built-in functions for number generators
- built-in functions for queue and vector operations
- built-in functions for simulation reports
### 24.3.2.2 Action Examples

This section exemplifies the usage of the different categories of built-in functions, given by the FSM action language. In doing so, these action examples are also used to show, how the different built-in functions can be nested to create complex action statements. Like other C/C++ functions, MLDesigner FSM built-in-functions can be nested in such a way, that a function parameter of a specific data type can be set by another function, which returns a value of that data type.

**NOTE:** In connection with data structures, nesting of FSM built-in functions is only working, if the data structure, returned by the inner-function, is compatible to the data structure expected by the parameter of the outer-function. For instance, it is not possible to set a data structure field of type `Root.Address.IPAddress` with a data structure of type `Root.FloatVector`. Such action statements are indeed compilable but lead to run-time errors!

The action examples are based on an FSM model with the following interface:

- Output port `IntOutput` of type `Root.Integer`
- Output port `FloatOutput` of type `Root.Float`
- Output port `StringOutput` of type `Root.String`
- Output port `EnumOutput` of type `Root.ENUM.Boolean`
- Memory `PacketMemory` of type `Root.NetworkProtocol.TCPProtocol`
- Memory `IntMemory` of type `Root.Integer`
- Memory `FloatMemory` of type `Root.Float`
- Memory `StringMemory` of type `Root.String`
- Memory `VectorOfPacketsMemory` of type `Root.Vector.VectorOfPackets`
- Memory `IntVectorMemory` of type `Root.IntVector`
- Memory `FloatVectorMemory` of type `Root.FloatVector`
- Memory `EnumMemory` of type `Root.ENUM.Boolean`
- Memory `ListOfPacketsMemory` of type `Root.List`
- Special Event `IntEvent` of type `Root.Integer`
- Parameter `IntParameter` of type `int`
- Parameter `FloatParameter` of type `float`
- Parameter `EnumParameter` of type `Root.ENUM.Boolean`
Base Structures

//for-loop structure
WriteMemory(StringMemory, "Byte");

for (IntMemory = 1; (int)IntMemory <= 4; IncMemory(IntMemory))
{
    InsertFieldDS(SelectFieldDS(PacketMemory, "SourceIP"),
                  ReadStringMemory(StringMemory) << (int)IntMemory,
                  IntParameter + (int)IntMemory);
    InsertFieldDS(SelectFieldDS(PacketMemory, "DestIP"),
                  ReadStringMemory(StringMemory) << (int)IntMemory, 0);
}

//while-loop structure
IntMemory = 0;

while ((int)IntMemory < (int)LengthOfVector(IntVectorMemory))
{
    SetElementIntVector(IntVectorMemory, (int)IntMemory,
                         (int)IntMemory * (int)IntMemory);
    IncMemory(IntMemory);
}

//do-while-loop structure
IntMemory = 0;

do
{
    Enqueue(ListOfPacketsMemory, PacketMemory, 0, (int)IntMemory);
    InsertFieldDS(PacketMemory, "DestPort", (int)IntMemory);
    IntMemory = (int)IntMemory + 1;
}
while ((int)IntMemory <= 3);

//if-else structure
if ((int)SelectFieldDS(PacketMemory, "SourcePort") < (int)IntMemory)
{
    InsertFieldDS(PacketMemory, "SourcePort", (int)IntMemory);
    IntToEnum(EnumMemory, 0);
}
else
{
    InsertFieldDS(PacketMemory, "SourcePort", IntParameter);
    IntToEnum(EnumMemory, 1);
}
//switch structure
switch (IntParameter) {
    case 0:
        InsertFieldDS(PacketMemory, "SourcePort", 10);
        break;
    case 1:
        InsertFieldDS(PacketMemory, "SourcePort", 20);
        break;
    default:
        InsertFieldDS(PacketMemory, "SourcePort", 0);
        break;
}

**Composite Data Structure Handling**

InsertFieldDS(PacketMemory, "SourcePort", DSToInt(IntMemory));
InsertFieldDS(PacketMemory, "DestPort", (int)IntMemory + 10);

InsertFieldDS(SelectFieldDS(PacketMemory, "SourceIP"), "Byte1", IntParameter);
InsertFieldDS(SelectFieldDS(PacketMemory, "DestIP"), "Byte2", 162);
InsertFieldDS(PacketMemory, "Name", DSToString(StringMemory));
WriteMemory(StringMemory, DSToString(SelectFieldDS(PacketMemory, "Name")));
IntMemory = SelectFieldDS(SelectFieldDS(PacketMemory, "DestIP"), "Byte4");

**Enumeration Data Structure Handling**

IntMemory = EnumToInt(EnumMemory);

InsertFieldDS(PacketMemory, "SourcePort", EnumToInt(EnumParameter));
WriteMemory(StringMemory, EnumToString(EnumMemory));
InsertFieldDS(PacketMemory, "Name", EnumToString(EnumParameter));
IntToEnum(EnumMemory, ((int)IntMemory + 1) % 2);
StringToEnum(EnumMemory, "TRUE");

**Vector Data Structure Handling**

ChangeLengthVector(VectorOfPacketsMemory, IntParameter);
WriteMemory(StringMemory, "Packet");

for (IntMemory = 0; (int)IntMemory < IntParameter; IncMemory(IntMemory))
{
  InsertFieldDS(PacketMemory, "Name",
    ReadStringMemory(StringMemory) << (int)IntMemory);
  SetElementVector(VectorOfPacketsMemory, (int)IntMemory, PacketMemory);
}

if (LengthOfVector(VectorOfPacketsMemory) > 10)
{
  PacketMemory = AccessElementVector(VectorOfPacketsMemory, 10);
  RemoveElementVector(VectorOfPacketsMemory, 10);
}

IntMemory = 0;

while ((int)IntMemory < (int)LengthOfVector(FloatVectorMemory) - 1)
{
  FloatMemory = AccessElementFloatVector(FloatVectorMemory,
    (int)IntMemory + 1);

  if ((double)FloatMemory < FloatParameter)
    FloatMemory = FloatParameter;

  SetElementFloatVector(FloatVectorMemory, (int)IntMemory,
    (double)FloatMemory);
  IncMemory(IntMemory);
}

RemoveElementFloatVector(FloatVectorMemory,
  LengthOfVector(FloatVectorMemory) - 1);

Memory Access

ResetMemory(IntMemory);

while ((int)IntMemory > 2)
  DecMemory(IntMemory);

do
  IncMemory(IntMemory);
while ((int)IntMemory < 4);

InsertFieldDS(PacketMemory, "SourcePort", ReadMemory(IntMemory));
InsertFieldDS(PacketMemory, "SourcePort", ReadIntMemory(IntMemory));
InsertFieldDS(PacketMemory, "SourcePort", (int)IntMemory);
InsertFieldDS(PacketMemory, "SourcePort", IntMemory);

WriteMemory(IntMemory, IntParameter + 10);

**Port Handling**

if (DataNew(PacketInput))
{
    PacketMemory = ReadNewInput(PacketInput);
    StringToEnum(EnumMemory, "TRUE");
} else
{
    PacketMemory = ReadBufferedInput(PacketInput);
    StringToEnum(EnumMemory, "FALSE");
}

WriteOutput(FloatOutput, e() + pi());
WriteOutput(IntOutput, (int)SelectFieldDS(PacketMemory, "SourcePort"));
WriteOutput(PacketOutput, PacketMemory);
WriteOutput(StringOutput, "Hello World!");
WriteOutput(EnumOutput, EnumMemory);

**Special Event Handling**

PacketMemory = ReadCurrentEvent(PacketEvent);
InsertFieldDS(PacketMemory, "Name", StringMemory);

ScheduleEvent(PacketEvent, FloatParameter, PacketMemory, 0);

if (EventIsScheduled(IntEvent, (int)IntMemory))
{
    FloatMemory = EventResidualTime(IntEvent, (int)IntMemory);
    CancelEvent(IntEvent, (int)IntMemory);
    IncMemory(IntMemory);
    ScheduleEvent(IntEvent, (double)FloatMemory, (int)IntMemory);
} else
{
    IntMemory = IntParameter;
    ScheduleEvent(IntEvent, ExpRangen(FloatParameter), (int)IntMemory);
}

**Queue Operations**

ClearQueue(ListOfPacketsMemory);
InitQueue(ListOfPacketsMemory, 10);

for (IntMemory = 0; (int)IntMemory < 2; IncMemory(IntMemory))
{
    InsertFieldDS(PacketMemory, "SourcePort", IntMemory);
    Enqueue(ListOfPacketsMemory, PacketMemory, 0, (int)IntMemory);
}

if (QueueCheck(ListOfPacketsMemory, 0))
{
    PacketMemory = Dequeue(ListOfPacketsMemory, 0);
    StringToEnum(EnumMemory, "TRUE");
}
else if (QueueLength(ListOfPacketsMemory) > 0)
{
    PacketMemory = PeekQueue(ListOfPacketsMemory, 0);
    InsertFieldDS(PacketMemory, "DestPort", 10);
    PokeQueue(ListOfPacketsMemory, PacketMemory, 0);
}

RemoveElement(ListOfPacketsMemory, 0);

**Number Generators**

FloatMemory = TNow() + ExpRangen(FloatParameter) + e() * pi();

InsertFieldDS(PacketMemory, "DestPort",
(int)BinomialRangen(IntParameter, 0.5));

**Simulation Report Handling**

InfoNumber("Current time stamp: ", TNow(), "");
InfoNumber("Current IntMemory value: ", (int)IntMemory, "");

if ((int)IntMemory > IntParameter)
{
    InfoString("Value is ok!");
    DecMemory(IntMemory);
}
else if ((int)IntMemory == IntParameter)
{
    WarningString("Value has reached threshold!");
    IncMemory(IntMemory);
}
else
24.4 FSM Execution Semantics

State diagrams are typically used in the development of finite state machines (FSMs). They provide a visual representation of the state transitions and the conditions under which those transitions occur. Each state in the diagram represents a condition that the system is in, and the transitions between states represent changes in that condition.

24.4.1 FSM States

States can be defined in a variety of ways, depending on the specific requirements of the system. In general, states can be defined based on the values of certain input or output variables, or based on the state of certain internal variables. For example, a state might be defined as the state in which the output variable is equal to zero, or as the state in which the internal variable is set to a specific value.

24.4.2 FSM Transition Guards

Transition guards are used to specify the conditions under which a particular transition will occur. These conditions are typically defined as boolean expressions that evaluate to true or false.

The example transition guard conditions are referring to the example FSM interface elements, introduced in section 24.3.2.2.

- EnumToString(EnumMemory) == "TRUE"
- EnumToInt(EnumMemory) == 0
- DSToInt(SelectFieldDS(ReadNewInput(PacketInput), "SourcePort")) > 0
- QueueLength(ListOfPacketsMemory) != 0 && (int)SelectFieldDS(ReadNewInput(PacketInput), "DestPort") == IntParameter
- (int)IntMemory >= 0 && (int)LengthOfVector(IntVectorMemory) > (int)IntMemory
- !((int)SelectFieldDS(SelectFieldDS(PacketMemory, "SourceIP"), "Byte1") == 192 && (int)SelectFieldDS(SelectFieldDS(PacketMemory, "SourceIP"), "Byte2") == 162) || ((int)SelectFieldDS(PacketMemory, "SourcePort") == IntParameter && (int)SelectFieldDS(PacketMemory, "DestPort") != (int)IntMemory && EnumToString(EnumParameter) == "TRUE")

24.4 FSM Execution Semantics

This section describes how the FSM scheduler responds to new events caused by arriving data on input ports or via internal or special events.

NOTE: Guard condition statements must not be terminated by a semicolon (;) character.

The example transition guard conditions are referring to the example FSM interface elements, introduced in section 24.3.2.2.
24 FSM Domain

24.4.1 Initialization

Upon simulation startup, any action associated with the top level default entrance is performed and the initial state becomes the finite state machine’s current state, without execution of the initial state’s entry action. While the initial state is a hierarchical state, its default entrance action is executed and its default sub-state becomes the initial state.

24.4.2 Execution Steps

After an FSM block received new events, the FSM scheduler passes through the following steps:

1. All transitions, possessed by the current state and triggered by the received events, are searched.

   NOTE: Synchronous transitions are triggered by any event.

2. In order of the priority of the triggered preemptive transitions, the guard conditions are evaluated. The first transition, whose guard condition evaluates to true, fires.

3. If a preemptive transition fires, goto step 7.

4. If the current state contains a slave process, this slave process is executed.

5. In order of the priority of the triggered non-preemptive transitions, the guard conditions are evaluated. The first transition, whose guard condition evaluates to true, fires.

6. If no non-preemptive transition fires, goto step 13.

7. The exit actions of all states, left by the firing transition, are performed in order up the hierarchy, starting with the current state. Additionally all histories of the leaving hierarchy are updated.

8. The action of the firing transition is executed.

9. If the firing transition points to a non-empty history, the state stored in the history becomes the new current state, otherwise the target state of the firing transition becomes the new current state.

10. While the new current state is a hierarchical state, its default entrance destination becomes the new current state.

11. The entry actions of all states, entered by the firing transition, are performed in order down the hierarchy, ending with the new current state.

   NOTE: In case a state hierarchy is entered by its default sub-state, the default entrance action is executed before the entry action of the default sub-state is performed.

12. If the new current state possesses synchronous transitions, goto step 2.

13. The current state memory is updated.
24.5 Elevator Example

In this example, a finite state machine is used to describe a simple elevator. The control panel of the elevator contains $N$ buttons, to select a target level. These buttons are labeled $0, \ldots, N-1$, where $N$ is the total number of levels. In addition there is a Stop and a Go button available to stop and run the elevator. There is also a display to show the current level the elevator is on.

Figure 24.5: FSM Example, Simple Elevator

24.5.1 Interface

The elevator FSM contains the following interface elements:

- Input port LevelIn of type Root.Integer
- Input port Stop of type ANYTYPE
- Input port Go of type ANYTYPE
- Output port DisplayLevel of type Root.Integer
- Internal Memory CurrentLevel of type Root.Integer
- Internal Memory TargetLevel of type Root.Integer
- Internal Special Event Timer of type Root

24.5.2 Execution

Upon simulation startup, the initial state S1 becomes the finite state machine’s current state. The FSM remains in state S1 until a LevelIn event occurs, to undergo a state change to S2. State S3,
as the default entrance destination of state S2, becomes the next current state and the entry action associated with state S3 is performed. In this case the TargetLevel memory is assigned with the integer value of the LevelIn input port. Possessed by state S3 are 4 transitions T2, T7, T3 and T4, whereas T2 and T7 are inherited transitions of a higher priority than T3 and T4. The transitions T2, T3 and T4 are synchronous transitions and triggered immediately after S3 is entered. One of the three synchronous transitions fires, dependent on the value of the TargetLevel memory. If the value of the TargetLevel memory is unequal to the value of the CurrentLevel memory, the finite state machine goes either to state S4 or S5. If the value of the TargetLevel memory is greater than the value of the CurrentLevel memory, state S4 becomes the next current state, and its entry action is performed. The entry action of state S4 schedules the Timer event to occur 5 time steps later than the current time. This time frame simulates the time the elevator needs to move from one level to the next. The integer value of the CurrentLevel memory is placed on the DisplayLevel output port. Possessed by state S4 are the 3 transitions T2, T7 and T5. The synchronous transition T2 can not fire immediately, since the values of the CurrentLevel and TargetLevel memories are unequal. The finite state machine remains now in state S4 until a Timer or Stop event occurs. In the case of a Timer event, the state S4 is exited. After the value of the CurrentLevel memory is incremented by the action associated with transition T5, state S4 is re-entered and its entry action is performed again, since transition T5 is a self transition of state S4. If now the value of the CurrentLevel memory is equal to the value of the TargetLevel memory, the synchronous transition T2 fires immediately and the states S4 and S2 are exited and state S1 is entered. In the case of a Stop event, while the elevator is in between two levels, so the FSM is either in S4 or S5, the current state and its ancestor state S2, are exited with the exit action of state S2 to cancel the current Timer event and state S6 is entered. After a Go event, the state S6 is exited and the state, stored in the history, either S4 or S5, becomes again the current state. A new Timer event is scheduled by the entry action of either state S4 or state S5. When this new Timer event occurs, the elevator moves on, in the same direction, as before the Stop event.

### 24.6 The FSM Model

An FSM model contains all the elements associated with a finite state machine. From the view of these model elements, an FSM model splits up into two different model types, the interface and the state transition diagram of a finite state machine.

#### 24.6.1 FSM Model Interface

The interface of an FSM model differs slightly from a typical MLDesigner model interface, specially of models of the discrete event (DE) domain. It contains the elements of a finite state machine to interact with an outer environment or to represent constants and variables. The following interface elements are available for FSM models:

- **single** Input Ports
- **single** Output Ports
- Internal / External Memory arguments
24.6 The FSM Model

- Internal / External Special Event arguments
- Parameter arguments

NOTE: These elements can be of any type or data structure.

24.6.2 FSM Model Design

The FSM model design represents the state transition diagram of a finite state machine. It contains all the state objects, like normal states, history states, or default entrance states, and their relations via transition objects. The graphical representation of the state transition diagram is completely embedded into the associated FSM model.

24.6.3 Current State Data Structure

Associated with each FSM model is a special data structure called current state data structure. This data structure is derived from the MLDesigner Root.ENUM data structure and named by default of the FSM model name followed by CS (e.g. SimpleElevatorCS). Each item of this enumeration data structure represents the logical name of a state of the associated state transition diagram. The current state data structure of an FSM model is generated automatically and updated, if the set of states or the logical name of a state has changed.

24.6.4 Current State Memory

Each FSM model must contain a memory argument called current state memory. This Memory argument is added automatically to a new FSM model and contains the default name CurrentState. In this context, the current state memory can not be deleted from the FSM model. The type of this memory argument is always set to the associated current state data structure. During execution, the value of the current state memory is always the logical name of the current state. If the scope of this memory argument is external, another block linked to this memory argument can change the finite state machine's current state.

NOTE: Changing the value of the current state memory from outside of the FSM block or via an FSM action statement causes no action execution.

The state, specified by the value of the current state memory, becomes the current state. If this state is a hierarchical state, its default entrance destination becomes the current state and so on, until the destination of a default entrance is a leaf state.

24.6.5 CurrentStateDS Property

The CurrentStateDS property can be used to change the name of the current state data structure associated with every FSM model. In this context, the old current state data structure is removed from the parent library model of the FSM model and a new generated current state data structure based on the new specified name is added to the parent library.

The type of the current state memory is automatically updated to the new current state data structure.
NOTE: Because the old current state data structure is deleted, any other elements (e.g. ports, memories), using this data structure will lose the reference to the old current state data structure and they are not automatically updated to the new current state data structure.

24.6.6 Internal Event Property

In addition to the basic properties of MLDesigner models, an FSM model contains a string property called Internal Events. The value of this property is a space separated list of the names of the internal events of the finite state machine. Each time the value of this property changes, every item on the list of internal events is checked to see if it is unique in the scope of the FSM Model and not equal to the name of any interface element.

24.6.7 Additional Code Property

The Additional Code property is a powerful feature for advanced usage of the MLDesigner FSM model. It provides the facility to define additional ptlang primitive source code in context with a finite state machine. Likewise for MLDesigner primitive models, it is possible to include additional header files or to define global variables and functions. These functions can be called in any action associated with states, transitions or default entrances. Global variables can be read and written in any action and used in expressions of transition guard conditions.

Example

//This code section can be used to define additional
//ptlang source code, e.g. global methods, variables
//or additional header includes, used in context with
//State Entry and Exit Actions, Transition Actions
//and Guard Conditions and the Action of the FSM top
//level Default Entrance.
//NOTE: The methods: constructor, destructor, go,
//begin, setup and wrapup are reserved and cannot
//be defined in this section !!!

hinclude {}
ccinclude {}

public
{
    double mArc;
    double mAngle;
}

method
{
    name { RadianToDegree }
}
The variables mArc and mAngle can be written and read in any action and used in transition guard conditions, e.g.

\[ m\text{Angle} \geq 90 \]

Furthermore, the functions \texttt{RadianToDegree} and \texttt{DegreeToRadian} can be called in any action, e.g.

\[ m\text{Angle} = \text{RadianToDegree}(m\text{Arc}); \]

**NOTE:** As described in the comment at the beginning of the additional code example, it is not possible to define reserved methods: \texttt{constructor}, \texttt{destructor}, \texttt{go}, \texttt{begin}, \texttt{setup} and \texttt{wrapup}. Furthermore it is highly recommended to check the additional code section for validity before editing any action or transition guard condition. Parse or compiling errors are often caused by source code from this section.

## 24.7 FSM Model Editor

The FSM model editor provides all the functionality needed to create the graphical representation of a finite state machine. Additional icons, specific to the FSM domain, become available on the toolbar when you create a new FSM or open an existing FSM model. Parameter arguments can also be added to an FSM model as elsewhere in MLDesigner.
24 FSM Domain

24.8 FSM Design Objects

This section introduces all the FSM design objects and their behavior inside the FSM model edit window.

**NOTE:** The interface elements of an FSM model are not introduced, since they can be used in context with finite state machines as elsewhere in MLDesigner.

The picture below shows an example with all kinds of FSM design objects.

![FSM Design Objects Diagram](image)

**Figure 24.7: FSM Design Objects**

24.8.1 States

States are depicted by a rounded rectangle. If a state is the only selected object, a *size grip*, located in every corner, is visible. The size grips can be used to change the size of the state view object. Each state **must** have a unique name, centered at the top of the graphical state object.
24.8.2 Transitions

Transitions between different state objects are depicted by a single line or multiple line segments, like connections between ports in other MLDesigner models. An arrow is located on one end of the transition, pointing to the transition’s target state. The start and the end point of a transition resides on the shape boundary of a state, history or default entrance view object. Self transitions are depicted by a semi ellipse on one side of the associated state.

24.8.3 Default Entrance Transitions

Default entrance transitions are only used to specify either the top level default entrance or the default entrance of a hierarchical state. In this context, a default entrance transition possesses neither a transition label nor any other transition property and is never active while running a finite state machine.

24.8.4 Transition Labels

Associated with each transition (exempted default entrance transitions) is a transition label, which is depicted like any other text label in MLDesigner, but with fixed font, color and size. The transition label is located near its associated transition in the FSM model edit window.

24.8.5 Default Entrances

A default entrance is depicted by a full circle with fixed radius. The radius of the default entrance view object is greater than the radius of transition points and less than the radius of the history view object.

24.8.6 Histories

Histories are depicted by a circle with fix radius and either a $H^*$ (for recursive history), or a $H$ (for non-recursive history) placed in the middle of the circle.

24.9 FSM Dialogs

24.9.1 Action Dialog

The FSM action dialog is a custom dialog for creating FSM action statements. The major reason for the FSM Built-In Functions and FSM Interface Elements sub-dialog is, to prevent syntax errors in FSM action statements. These sub-dialog can be used to include either an FSM built-in function or the name of an FSM interface element at the current text cursor position. The FSM action statements can be verified, using the corresponding tool button. If the statements contain syntax errors, the verification output window becomes visible and all errors with their associated line number are listed.
24.9.2 Event Expression Dialog

Like the FSM action dialog, the event expression dialog is another custom dialog, in this case used to create logical event expressions for transitions. The major part is the line edit at the top of the dialog, used to enter an event expression. All the elements below the line edit are helpful for creating an expression. In this context, the 3 combo boxes contain all the names of the input ports, special event arguments, and internal events of the associated FSM model. Closing the dialog by the Ok button, the expression is checked for validation and an error message occurs in case of parse errors are present.
### 24.9.3 Slave Model Dialog

The slave model dialog is a *custom dialog* to select a slave process model for leaf states. Since a slave process model must be either another FSM model or an MLDesigner module, only these two specific model types and their associated library models are listed in the tree view of the dialog. If the selected slave process model is the same FSM model as the superordinate finite state machine, an error message occurs and no new slave process model is set for the leaf state.

![Slave Model Dialog](image)

**Figure 24.10: Slave Model Dialog**

### 24.10 FSM Design Check

The *Check Design* tool button in the FSM model editor toolbar is used to check the FSM design for semantic errors. If the FSM design contains errors, a message occurs and all semantic errors are listed. This FSM design check is done automatically while saving a finite state machine model. The following semantic rules for states, default entrances, and histories are checked in association with the FSM design objects.

#### 24.10.1 States

- each state must be reached by at least one non-self transition
- each state must be left by at least one non-self transition
- each state hierarchy level and the FSM top level must have exactly one default entrance
- each state must not have any synchronous self transitions without a guard condition

**NOTE:** Hierarchical states can also be reached by transitions, whereas the source state is a state outside the hierarchical state and the target state is a descendant state of the hierarchical state. Hierarchical states can also be left by inherited transitions or by transitions,
whereas the source state is a descendant state of the hierarchical state and the target state is a state outside of the hierarchical state. States must not have synchronous self transitions without a guard condition, since such transitions are 100% candidates for infinite loops. There are a lot of ways to create infinite loops in a finite state machine, but they can not all be detected by the FSM semantic check.

24.10.2 Default Entrances

- each default entrance must have exactly one outgoing transition
- the target of the default entrance must reside at the same hierarchy level
- the target of the default entrance must not be a history

24.10.3 Histories

- each history must have at least one incoming transition

24.11 FSM and Concurrency Domains

In MLDesigner, a finite state machine is always combined with other MLDesigner models, since an FSM model is always embedded into a wormhole of a concurrency domain or different MLDesigner models can be used as a slave process inside an FSM model. This section describes, how FSM models interact with the Discrete Event (DE) domain, the Synchronous Data Flow (SDF) domain and the Finite State Machine (FSM) domain.

24.11.1 FSM and DE

The MLDesigner DE domain uses an event driven model of computation. Events occur at a point in time. A time stamp, possessed by every event, indicates the time, at which the associated event occurs.

24.11.2 FSM inside DE

An FSM model, embedded in a DE domain environment, behaves like any other DE model. New data on an FSM model input port represent the presence of a new event to trigger the FSM model and new data on an FSM model output port, generated during FSM execution, are interpreted as new events for the DE environment, whereas the FSM model reacts to the outer DE domain as a zero delayed system. In this context, output events, generated by the FSM model, get the same time stamp as the input event, which triggered the execution of the FSM model. A DE outer domain is the only case where an FSM model is able to use special event arguments to cause state changes inside the finite state machine, because all other domains do not support special events.

In case of an outer DE environment, the usage of the logical AND (\(\&\)) operator within transition event expressions is only practical for internal events in connection with FSM slave models. Event expressions of AND-combined FSM input ports or special events are only evaluating to true in the special case, the involved events are all present at the same time in the outer DE event queue and
scheduled for the same time stamp. But basically, FSM models inside DE react immediately on an incoming port event or dispatched special event. In doing so, all present events are consumed during FSM execution, independent if a transition has been able to fire on these events or not. In other words, FSM models do not queue events and are not waiting for missing events to satisfy event expressions including AND-combined input ports or special events. Therefore event expressions of non-synchronous transitions should only consist of a single event name, if the associated FSM model is embedded in a DE environment.

24.11.3 FSM outside DE

If a DE domain module is used as a slave process inside an FSM model, the supply of the input ports of the slave process, before execution, depends on the outer domain of the FSM model. In case, the outer domain is also a DE domain and there is currently no event on an input port of the FSM model, the appropriate input port of the DE slave process model gets also no event. In all other cases, each input port of the slave process gets a new event, before the slave process executes. All the input events of the slave process get the current time stamp of the superordinate FSM model. If the outer domain of the FSM model is also a DE domain, the current time stamp of the FSM model is the same, as the time stamp of the FSM model triggering event. Otherwise, the time stamp of the FSM model is zero. Output events, produced by the DE slave process model, are placed on the appropriate FSM model output ports. Again with the current, zero delayed time stamp of the FSM model.

24.11.4 FSM and SDF

An SDF system consists of a set of modules or primitives interconnected by directed arcs. MLDesigner SDF models represent computational functions that map input data into output data. Unlike the DE domain, the SDF domain is not event driven and there exist always data on each input and output port of the SDF model.

24.11.5 FSM inside SDF

An FSM model, embedded into an SDF domain environment, behaves like any other SDF model. To ensure this behavior, the FSM model needs an approach to differ between the presence and absence of an event, since there exist always data on each input port of the FSM model. In this context, the FSM model determines the presence and absence of an event via the integer cast of the appropriate input data. If this data cast returns zero, the associated event is interpreted absent and in the case of a non zero result, the associated event is interpreted present. If the FSM model execution produces no data for an associated output port, a zero valued data is placed on this output, to ensure that there are always data available on each FSM output port, as required by the semantic of the outer SDF domain.

24.11.6 FSM outside SDF

If an SDF domain module is used as a slave process inside an FSM model, each input port of the slave process model gets the same data as currently available on the appropriate FSM model input port, before the slave Process executes. Each execution of an SDF domain slave process model
is interpreted as one iteration. The output data, produced by the slave process, are placed on the
appropriate output ports of the superordinate FSM model.

24.11.7 FSM inside FSM

If an FSM model is used as a slave process inside another FSM model, each input port of the slave
FSM model gets the same data as currently available on the appropriate input port of the master
FSM model, before the slave Process executes. In the case, the master FSM model is embedded
into an DE domain environment and there is currently no event on an input port, the appropriate
input port of the slave FSM model gets a zero valued data, to specify the absence of the associated
event. Likewise an FSM model is embedded into an SDF domain environment, the slave FSM
model determines the presence and absence of an event via the integer cast of the associated input
data. output data, produced by the slave FSM model, are placed on the appropriate output ports of
the master FSM model.

24.12 Creating an FSM

A commonly used example for control-intensive software environments is the so-called “reflex
game” found in the Library view under MLD_Libraries/Demos/FSM_Demos/ReflexGame2/ReflexGame2.
In this example we demonstrate how to create the FSM instance game2_FSM#1 used in the Demo.
We then instantiate the FSM in a copy of this system (saved in your user library) and reference
slave processes. These are the fundamental principles of the FSM domain and following the ex-
ample here will demonstrate the MLDesigner FSM interface simply and effectively.

24.12.1 System Description

This version of the reflex game has two players. Each player has two buttons to press during the
game: player one has coin and go and player two has ready and stop. The normal game-play
should proceed as follow:

1. Player1 presses coin to begin. The status light turns blue.
2. When Player2 presses ready; the status light turns yellow.
3. Player1 presses go to start the game. When the status light turns green player2 has to press
stop as fast as possible.
4. The game ends after player2 presses stop. The status light turns red. The time between
Player 1 pressing go and Player2 pressing stop is indicated in the **Time elapsed** display of
the game control console.

The game times-out, indicated by the light display area of the Tcl display becoming light grey, if:

1. Player2 does not press ready within a certain time from the moment player1 pressed coin.
2. Player2 presses stop before or exactly when player1 presses go
3. After player1 presses go, player2 does not press stop within a certain amount of time

Another additional rule is that if player1 does not press go within an amount of time after player2
presses ready, then go will be asserted by the system and the game advances to the next step and
player2 is expected to press stop.
24.12 Creating an FSM

24.12.2 Example

The reflex game is a mixture of FSM machines, some modules in the DE domain, one module in the SDF domain and some Tcl scripts.

NOTE: The Tcl scripts and the implementation of the DE and SDF domains are not discussed extensively in this chapter. For more information on combining models refer to the Modeling Guide. This chapter only implements the FSM machines and the interface of each FSM to the other modules.

The ReflexGame2 is the topmost system, as shown in fig. 24.11. The reflex game is a real-time game and is therefore modeled in the DE domain. A brief explanation of the system is:

- The module clock generates a sequence of clock ticks which are synchronized by the module Synchronize.
- The modules Player1, Player2 and Display are Tcl scripts to create the buttons and output lights.
- The module Game2_FSM#1 models the main behavior of the game, this module consists of two state FSM (see fig. 24.12.)

Figure 24.11: ReflexGame2 System in the DE domain

To create this FSM module in your user-libraries proceed as follows:

1. Click the New Model icon on the upper toolbar.
2. Create a Library called MyFSM as a sub-library of MyLibrary.
3. Click OK.
4. Click the New Model icon on the upper toolbar.
5. Select FSM from the Type of Model drop-down menu.
6. Type MyReflex in the Logical Name input field.
7. Click OK.

The new FSM module is now open in the Model Editor Window and has a Memory model instance called CurrentState. In this context, the Current State Memory can not be deleted from the FSM model. The type of this Memory argument is always set to the as-
sociated Current State Data Structure. During execution, the value of the Current State Memory is always the logical name of the current state of the finite state machine. If the scope of this Memory argument is external, another block linked to this Memory argument can change the finite state machine’s current state.

NOTE: Changing the value of the Current State Memory from outside of the FSM block or via an FSM Action statement causes no action execution.

The state, specified by the value of the Current State Memory, becomes the current state. If this state is a hierarchical state, its default entrance destination becomes the current state and so on; until the destination of a default entrance is a leaf state.

We now need to create the two states called Game_On and Game_Off.

8. Click the Add State icon and place the cursor inside the Model Editor Window.
9. Click the mouse twice to create two states with the default names State0 and State1. A right click of the mouse returns the cursor to selection mode.
10. Arrange the states so you can see both clearly.
11. Select one state instance to activate the Model Properties window and change the Logical Name to Game_On. Repeat this step with the second state instance and call it Game_Off.

It is not possible to define expressions for transitions before you have created events and attempting to do so will result in an Invalid Event Expression warning being displayed. Events are generated by input ports, event arguments, or by internal events. Internal events are set by Slave Processes and are not connected to the interface of this FSM module. The first step here is to create the input/output ports and then the internal events. Proceed as follows:

12. Select the Add Input Port icon on the toolbar and click the mouse five times with the cursor placed over the left edge of the bounding box. Move the cursor slightly after each click to space the input ports evenly.
13. Repeat the previous step but select the Add Output Port icon and click six times over the right edge of the Model Editor Window.

The ports are all named Input1 to Input5 and Output1 to Output6. We need to rename the ports so it is easy to identify which port is which when we instantiate the FSM module into the system, and to define the event names for the transition expressions. See fig. 24.12 and rename the ports in the Port Properties window. Here the port name is the event identifier. At the same time change the Data Type of the ports to int except for the time input port which has type float.

14. Click on the module background to activate the Model Properties window.
15. In the Internal Events property type exit error to define the two events.

Now the FSM model need a Default Entrance. Click the Default Entrance icon on the toolbar to change the cursor mode. Click once in the Model Editor Window to place the default
entrance (a black dot) close to the Game_Off instance. A right click of the mouse returns the cursor to selection mode. You do not need to set any action for the default entrance but you do need to define which state is the starting point of the FSM module. Draw a transition from the default entrance to the Game_Off state to define the starting point. The next step is to create the transitions needed for the FSM to function.

16. Select the Add Transition icon on the toolbar and move the cursor over the Game_On state.
17. Click the left mouse button once to see the Self Transition appear as a loop back to itself. Another left mouse click anchors the transition on the state.
18. Click the left mouse twice on the edges of the Game_Off state to create another Self Transition.
19. Click on the Game_On state once and move the cursor towards the Game_Off state. Click the mouse once to complete the transition. Repeat the procedure from Game_On to Game_Off but first move the cursor slightly up and click the mouse once on the model background to create a node anchor so that the two transitions do not overlap. Now perform the same procedure in the opposite direction to create a transition from Game_Off to Game_On.

**NOTE:** The transition is directional so the direction the connections are made in plays an important role.

Return the cursor to selection mode with either a right mouse click or by selecting the Select Tool icon from the toolbar. The transitions may need to be arranged so they look neat and do not overlap one another. There are sliding nodes on the transitions that are visible as soon as a transition is highlighted and the cursor is moved with depressed mouse button. Arrange the transitions to look something like those in fig. 24.12.

![Figure 24.12: game2_FSM#1 FSM](image)

The next step is to define the **Conditions** and **Actions** for each transition.

20. Select the transition going from Game_Off to Game_On.
21. In the **Event Expression** input field type `coin`.
22. In the **Action** input field type `WriteOutput(blueLt, 1);`
23. Select the transition going from `Game_Off` to `Game_On`. Change the **Entry Type** in the Transition properties window from `History` to `Default`.

24. Select one of the transitions going from `Game_On` to `Game_Off`
25. In the **Event Expression** input field type `error`
26. In the **Action** input field type
   ```java
   WriteOutput(flashTilt, 1); WriteOutput(redLt, 1);
   ```
27. Select the other transitions going from `Game_On` to `Game_Off`
28. In the **Event Expression** input field type `exit`
29. In the **Action** input field type
   ```java
   WriteOutput(ringBell, 1); WriteOutput(redLt, 1);
   ```
30. Select the self transition of the `Game_Off` state. In the **Event Expression** input field type `!coin`. The same is entered in the **Label** input field automatically.
31. Select the self transition of the `Game_On` state. In the **Event Expression** input field type `!exit && !error`. The same is entered in the **Label** input field automatically.
32. Save the model.

   The next step would be to create the slave processes which are needed for this system to run. For this exercise, however, we will merely show how the slave processes are referenced by the top level FSM. When the system is running with your FSM you can look at all the instances and analyze how the slave processes work. It makes sense to read the rest of this chapter while going through the system.

   The simplest way to get this system running is to open the Demo system `MLD_Libraries/Demos/FSM Demo/ReflexGame2/ReflexGame2` and select **Save As** from the main **File** menu. Save the system in your **MyFSM** library. With the newly saved system open in the Model Editor Window, select the **MyReflex** FSM and click and drag it over the `Game2_FSM#1` instance. You will see the color of the model instance change indicating it is possible to replace the instance with your FSM. Release the mouse button to replace the model instance while keeping all connections between ports intact. If any connections are not intact, check your FSM port names and make sure they are the same as the demo FSM.

   The Slave Process must now be defined:

33. Double-click the **MyReflex** model instance.
34. Select the state `Game_On` to activate the State Properties window.
35. Click on the field **Slave Path** and click the **Module** icon to open the **Select Slave Model** dialog.
36. Select `MLD_Libraries/Demos/FSM Demo/Reflex_Game2/GameOn` as the slave process and click **OK**
37. Save the model.
The game can now be started using your FSM and the GameOn slave process.

**NOTE:** At the moment, MLDesigner is not able to determine internal events. Always check that the field is properly set (to use the events) and that these events are properly controlled by slave processes.

![Diagram](image)

**Figure 24.13:** the GameOn slave module, in DE domain

The GameOn module is a slave process consisting of the rules for the two players, as shown in fig. 24.13. This module is implemented in the DE domain and is nothing more than a divert to two other FSMs that implement two rules of the game.

If we analyze the GameOn module, we can see how the interfaces to FSMs are implemented. Once again, the events to be controlled are implemented as input and output ports (the names of these ports are, of course, identical to the events in the master FSM). The DE module consists of two concurrent FSMs to implement the rules of the game. These are interconnected using a zero-delay(!) loop and thus form an instantaneous dialog between the two players.

The two slave FSMs, as shown in fig. 24.14 and fig. 24.15 are created in exactly the same way as the MyReflex FSM explained above. They can be found at $MLD/MLD_Libraries/FSM/Demo/reflexGame2/rule1_FSM and $MLD/MLD_Libraries/FSM/Demo/reflexGame2/rule2_FSM

In several states, we need to count ticks from the clock (each clock is given as an event, in our system simply called time, and has been given to every slave process). The counting is a simple arithmetic computation that can be performed using the data flow graph shown in fig. 24.16. This graph, realized in the SDF domain, simply counts ticks, compares the count against a constant (this denotes the maximum time) and emits a timeout event when the threshold is exceeded.
24 FSM Domain

Figure 24.14: rule1_FSM for GameOn

Figure 24.15: rule2_FSM for GameOn
When time elapsed is larger than 1000, it will emit the error signal. (ie. error == 1)

Figure 24.16: The count module, in SDF domain

24.13 Backward Compatibility

The Create STD File property, in the model properties window, associated with MLDesigner FSM models, allows you to revert FSM models to the old format. The main purpose for this property is the further support of FSM code-generating tools which are based on the semantic of Ptolemy and previous versions of MLDesigner.

If the value of this property is set to Yes, the STD file with the name of the source FSM model and the extension .std is generated each time the associated FSM model is saved.

Only the top-level states of the FSM model and transitions, where the source state and the target state is a top-level state, are considered. If the FSM model contains hierarchical states a warning message occurs while generating the STD file. History states are ignored by the STD file generator. Not all the state and transition properties of the new MLDesigner FSM model can be mapped into the STD representation. The State Entry Action, State Exit Action and Transition Guard Condition property are not supported.

NOTE: The action property of a transition in the STD representation simply contains a list of names of output ports of the FSM model. The transition outputs an event on this ports if the transition fires. The STD file generator parses the contents of the action property of a MLDesigner FSM transition for the names of the FSM model output ports, to create the action for the associated transition in the STD representation.

A generated STD file is shown in the MLDesigner model file view as a sub-model of the associated FSM model but it is physically stored in the same directory. Changes made to STD files do not influence the source MLDesigner FSM model and may be edited using the FSM STD editor.


24.14  ANSI C Code Synthesis

24.14.1  Overview

The integrated ANSI C code generator enables generation of highly portable and run-time system independent C code for FSM based MLDesigner models (section 24.14.2). Major application area of this code generation option is to complete the model-based design flow for software control units of embedded and real-time systems. In other words, ANSI C code synthesis of FSM models offers the possibility to transform a designed and verified model into functional equivalent ANSI C source code. With respect to some limitations (section 24.14.4), the code generator supports all significant design elements of MLDesigner FSM models. The resulting source code can be customized for specific run-time systems (section 24.14.8), provided that the underlying C compiler is ANSI C conform.

NOTE: Most parts of this section are taken from [Rat03].

24.14.2  Generator Input

MLDesigner provides automated ANSI C code generation for two generator input model types: FSM modules and single FSM models.

Basically, an FSM module is a module of the discrete event (DE) domain. But the entire module functionality is represented by interconnected FSM instances or lower level FSM modules (fig. 24.17).

![Figure 24.17: FSM Module](image)

Likewise any other MLDesigner module, an FSM module may possess input and output interface ports and supports internal memories and parameter arguments.

In case automated code generation is used to produce ANSI C code for a single FSM model, the code generator internally creates a corresponding FSM module. Such a special FSM module just includes one instance of the selected FSM model and the same input/output interface. Accordingly, each FSM instance port is connected to its associated module interface port.
24.14 ANSI C Code Synthesis

24.14.3 Generator Output

The set of source files, produced by the code generator, is divided into three categories. First, the generator output includes a separate C source file for the implementation of each FSM model, which is instantiated in the generator input FSM module. The second category implements an associated DE run-time environment (section 24.14.6) to ensure an input model equivalent execution behavior. Finally, a set of configuration files enables a run-time system specific code customization (section 24.14.8).

The complete list of output source files and additional content details are given in section 24.14.7.

24.14.4 Limitations

Some design limitations and conditions are necessary to ensure widespread and efficient usage of the ANSI C source code, produced by the code generator.

24.14.4.1 FSM Module Limitations

A valid FSM module must only contain single interface ports and must not include any lower level FSM module instance with multi ports.
Module interface ports and instance ports of type anytype are considered as integer ports by the code generator.
Neither internal nor external event elements are directly supported for FSM modules.
Lastly, an FSM module, directly used as generator input model, must not contain external memory elements.

24.14.4.2 FSM Semantic Limitations

FSM state slave models and all related properties are not supported.
To avoid additional code overhead, the code generator does not implement the current state memory as well as the associated current state enumeration data structure of instantiated FSM models.
Only internal event elements are supported for FSM models, used to represent local and type-less asynchronous timer events.
In case of a single FSM model is directly used as generator input model, it must not contain external memory elements.
Finally, the set of supported action statements is restricted in terms of unsupported design elements and data types.
Table 24.1 lists all significant features of the MLDesigner FSM semantic in conjunction with ANSI C code synthesis.
FSM action built-in function limitations are listed in table 24.2 and table 24.3.

24.14.4.3 Data Type Limitations

ANSI C code synthesis of MLDesigner FSM models supports almost all data types, provided by the system design tool. Some limitations have to be made in terms of memory consumption and run-time efficiency.
Only well defined data types of a fixed memory size are allowed to avoid dynamic memory allocation. In this context, a maximum length must be defined for string characters. Furthermore,
<table>
<thead>
<tr>
<th>Feature</th>
<th>supported</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Ports</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Output Ports</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Internal Memories</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>External Memories</td>
<td>limited</td>
<td>not supported in case of single FSM input models</td>
</tr>
<tr>
<td>Internal Special Events</td>
<td>yes</td>
<td>considered as local and type-less asynchronous timer events</td>
</tr>
<tr>
<td>External Special Events</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Internal Events Property</td>
<td>no</td>
<td>related to state slave models</td>
</tr>
<tr>
<td>Parameters</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Current State Memory</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Current State Data Structure</td>
<td>no</td>
<td>related to current state memory</td>
</tr>
<tr>
<td>State XOR-Decomposition</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>State Slave Model</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>State Logical Name</td>
<td>yes</td>
<td>used for FSM debugging</td>
</tr>
<tr>
<td>State Entry Action</td>
<td>limited</td>
<td>depends on supported statements</td>
</tr>
<tr>
<td>State Exit Action</td>
<td>limited</td>
<td>depends on supported statements</td>
</tr>
<tr>
<td>Transition Event Expression</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Transition Guard Condition</td>
<td>limited</td>
<td>depends on supported statements</td>
</tr>
<tr>
<td>Transition Action</td>
<td>limited</td>
<td>depends on supported statements</td>
</tr>
<tr>
<td>Transition Label</td>
<td>no</td>
<td>not required</td>
</tr>
<tr>
<td>Transition Preemptive Flag</td>
<td>no</td>
<td>related to state slave models</td>
</tr>
<tr>
<td>Transition Entry Type</td>
<td>no</td>
<td>related to state slave models</td>
</tr>
<tr>
<td>Static History States</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Recursive History States</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Default Entrance Action</td>
<td>limited</td>
<td>depends on supported statements</td>
</tr>
<tr>
<td>Additional Code Property</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Table 24.1: FSM Semantic Limitations
<table>
<thead>
<tr>
<th>Built-In Function</th>
<th>supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSToFloat</td>
<td>yes</td>
</tr>
<tr>
<td>DSToInt</td>
<td>yes</td>
</tr>
<tr>
<td>InsertFieldDS</td>
<td>yes</td>
</tr>
<tr>
<td>SelectFieldDS</td>
<td>yes</td>
</tr>
<tr>
<td>EnumIsEqual</td>
<td>yes</td>
</tr>
<tr>
<td>EnumToInt</td>
<td>yes</td>
</tr>
<tr>
<td>EnumToString</td>
<td>yes</td>
</tr>
<tr>
<td>IntToEnum</td>
<td>yes</td>
</tr>
<tr>
<td>StringToEnum</td>
<td>yes</td>
</tr>
<tr>
<td>DecMemory</td>
<td>yes</td>
</tr>
<tr>
<td>IncMemory</td>
<td>yes</td>
</tr>
<tr>
<td>ReadFloatMemory</td>
<td>yes</td>
</tr>
<tr>
<td>ReadIntMemory</td>
<td>yes</td>
</tr>
<tr>
<td>ReadMemory</td>
<td>yes</td>
</tr>
<tr>
<td>ReadStringMemory</td>
<td>yes</td>
</tr>
<tr>
<td>ResetMemory</td>
<td>no</td>
</tr>
<tr>
<td>WriteMemory</td>
<td>yes</td>
</tr>
<tr>
<td>BinomialRangen</td>
<td>yes</td>
</tr>
<tr>
<td>ExpRangen</td>
<td>yes</td>
</tr>
<tr>
<td>GaussianRangen</td>
<td>yes</td>
</tr>
<tr>
<td>PoissonRangen</td>
<td>yes</td>
</tr>
<tr>
<td>TNow</td>
<td>yes</td>
</tr>
<tr>
<td>UniformRangen</td>
<td>yes</td>
</tr>
<tr>
<td>e</td>
<td>yes</td>
</tr>
<tr>
<td>pi</td>
<td>yes</td>
</tr>
<tr>
<td>DataNew</td>
<td>yes</td>
</tr>
<tr>
<td>ReadBufferedInput</td>
<td>yes</td>
</tr>
<tr>
<td>ReadNewInput</td>
<td>yes</td>
</tr>
<tr>
<td>WriteOutput</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 24.2: FSM Action Built-In Function Limitations (1)
<table>
<thead>
<tr>
<th>Built-In Function</th>
<th>supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClearQueue</td>
<td>no</td>
</tr>
<tr>
<td>Dequeue</td>
<td>no</td>
</tr>
<tr>
<td>Enqueue</td>
<td>no</td>
</tr>
<tr>
<td>InitQueue</td>
<td>no</td>
</tr>
<tr>
<td>PeekQueue</td>
<td>no</td>
</tr>
<tr>
<td>PokeQueue</td>
<td>no</td>
</tr>
<tr>
<td>QueueCheck</td>
<td>no</td>
</tr>
<tr>
<td>QueueLength</td>
<td>no</td>
</tr>
<tr>
<td>RemoveElement</td>
<td>no</td>
</tr>
<tr>
<td>ErrorNumber</td>
<td>no</td>
</tr>
<tr>
<td>ErrorString</td>
<td>yes</td>
</tr>
<tr>
<td>InfoNumber</td>
<td>no</td>
</tr>
<tr>
<td>InfoString</td>
<td>yes</td>
</tr>
<tr>
<td>WarningNumber</td>
<td>no</td>
</tr>
<tr>
<td>WarningString</td>
<td>yes</td>
</tr>
<tr>
<td>CancelEvent</td>
<td>only with integer event ID</td>
</tr>
<tr>
<td>EventIsScheduled</td>
<td>yes</td>
</tr>
<tr>
<td>EventResidualTime</td>
<td>yes</td>
</tr>
<tr>
<td>ReadCurrentEvent</td>
<td>no</td>
</tr>
<tr>
<td>ScheduleEvent</td>
<td>only with 3 arguments and integer event ID</td>
</tr>
<tr>
<td>AccessElementFloatVector</td>
<td>yes</td>
</tr>
<tr>
<td>SetElementFloatVector</td>
<td>yes</td>
</tr>
<tr>
<td>AccessElementVector</td>
<td>no</td>
</tr>
<tr>
<td>ChangeLengthVector</td>
<td>no</td>
</tr>
<tr>
<td>LengthOfVector</td>
<td>yes</td>
</tr>
<tr>
<td>SetElementVector</td>
<td>no</td>
</tr>
<tr>
<td>AccessElementIntVector</td>
<td>yes</td>
</tr>
<tr>
<td>SetElementIntVector</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 24.3: FSM Action Built-In Function Limitations (2)
the set of vector data structures is limited to integer vector and float vector, both with a settable maximum number of elements.
Lastly, connected ports and linked memory arguments must be of exact the same data type, because the code generator does not support data type inheritance.
Table 24.4 presents all data types, supported for ANSI C code synthesis of MLDesigner FSM models. Members of composite data structures can be of any of these types, for instance another composite data structure.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>fixed number of characters</td>
</tr>
<tr>
<td>Composite Data Structures</td>
<td></td>
</tr>
<tr>
<td>Enumerations</td>
<td></td>
</tr>
<tr>
<td>Integer Vector</td>
<td>fixed number of elements</td>
</tr>
<tr>
<td>Float Vector</td>
<td>fixed number of elements</td>
</tr>
</tbody>
</table>

Table 24.4: Data Type Limitations

24.14.5 Code Generation Process

ANSI C code synthesis for either an FSM module or single FSM model can be activated by the Generate ANSI C Code icon available in the appropriate model toolbar or model context menu. Directly after code generation has been activated, the given generator input model is first checked for design errors and inconsistencies. Additionally, the input model is hierarchically parsed for design elements, not supported by the code generator. Corresponding error and warning messages are shown in the Log Window of the Console View. In this context, code generation aborts, if error messages are present. Else, if just warning messages are present, the user is prompted by an additional dialog box to choose to continue or to abort code synthesis. Otherwise, in case a given input model satisfies all generator conditions and limitations, code generation automatically continues and the configuration dialog box is shown (fig. 24.18).
This dialog box is mainly used to select a directory name and path for the output source files. In case the selected directory already includes a set of generated source files, the configuration dialog box provides an option to specify how existing configuration source files are handled by the code generator.
Additionally, this dialog box provides general pre-customization options to:

- determine the size of numeric types
- determine the length of involved numeric vector types and strings
- determine either a random or fixed seed used by involved random number generator built-in functions
- enable specific code debug sections
Figure 24.18: Code Generator Configuration Dialog
After all settings are done, ANSI C code generation for the selected input model can be started by the Generate button.

In case of code generation failed, for instance, the output source files could not be saved in terms of not enough free disk space, a message box is prompted, showing internal generator problems. Otherwise, the user is informed about successful code generation and the output source files are available in the selected directory.

24.14.6 Run-Time Environment

To ensure an equivalent execution behavior between a generator input model and its corresponding ANSI C code implementation, the generator output includes an additional run-time environment. The principle task of this run-time environment is to control execution of interconnected FSM instances. Since FSM modules are designed and simulated under a discrete event (DE) model of computation, the run-time environment must ensure an equivalent run-time behavior for every FSM instance. Based on the fact that generated output code is mainly used as software control units in an embedded and real-time application field, additional real-time conditions have to be considered in conjunction with event dispatching time and FSM instance execution duration. In this context, the run-time environment can be considered as a real-time operating system (RTOS) and every state machine instance represents a system task.

24.14.6.1 Scheduler

The underlying RTOS scheduler is basically an ANSI C reimplementation of the integrated MLD-e
designer DE priority-free scheduler.

Event Queue Central scheduler element is an event queue (EventQ), represented by a single linked list of EventQ entries. An EventQ entry combines all information, required to perform a specific task.

Task Interaction Connections between FSM instances are implemented via emitter and deliver functions. The RTOS includes an emitter function for each individual instance output port and a deliver function for each non-terminated instance input port, respectively. During instance execution, an emitter function is called whenever the appropriate output port is triggered to send new events. Internally, this function schedules a new EventQ entry for each connected instance input port. Whenever the scheduler processes an input EventQ entry, the deliver function of the appropriate instance input port is called.

Interface Interaction Similar to interaction between FSM instances, the RTOS manages FSM module interface ports on the basis of customizable generic get and send functions (24.14.8.3). The get function of each individual interface input port returns a boolean value (either 1 for true or 0 for false) to determine presence of a new event on this input port. Directly after a clock tick has been occurred, the scheduler calls all present get functions to check the module interface for new input events. In case a new event is present on an interface input port, an EventQ entry is scheduled for each connected instance input port.
If an instance output port is connected to an interface output port, the associated send function is called inside the instance output emitter function to send new data to the module output interface. In the special case that an interface input port is directly connected to an interface output port, no EventQ entry is scheduled and the output send function us called directly after the scheduler grabbed a new event by the input get function.

**Timelines**  In reference to an underlying DE model of computation and additional real-time conditions, the RTOS includes two separate timelines. First, an external clock tick timeline of a configured clock tick interval (CTI) determines the global system clock pulse. A corresponding clock tick counter (CTC) is incremented each time a clock tick occurred.

Secondly, an internal system time counter (STC) is used as time base for EventQ entries. To ensure a quasi-zero-delayed execution behavior of FSM instances, the STC is not updated until all scheduled synchronous events for this time stamp are processed. While the scheduler is able to perform all tasks, related to a specific system time stamp, within one clock tick period, both timelines are running synchronously. Otherwise, in case processing of all events, scheduled for the current internal system time, exceeds the CTI limit, a clock tick overrun occurs and given real-time conditions might be violated. In a soft real-time system, a clock tick overrun might be ignorable, but in case of hard real-time conditions, the application usually has to be aborted and reconfigured with a longer clock tick interval. To define consequences of a real-time violation, the generator output provides a configurable handler function, which is called whenever the defined real-time level could not be met.

Figure 24.19 shows an example soft real-time scenario with the time response of both RTOS timelines. All RTOS timeline related configuration units are detailed described in section 24.14.8.2.

![RTOS Timelines](image)

**24.14.6.2 Custom Memory Management**

The RTOS includes a separate dynamic memory management, customized for a particular generator input model.

Within implementation of FSM modules, dynamic memory management is only essential in conjunction with EventQ entries and associated input event data, since the number of scheduled events is permanently changing. In this context, the RTOS memory handling is based on partitions of contiguous memory areas. Thereby, a particular FSM module requires as many partitions as different data types are used between input/output port connections. Additionally, one special partition is necessary in conjunction with EventQ entries.
Each partition is subdivided into a configurable number of fixed-sized memory blocks (fig. 24.20). In doing so, the block byte size is typically equivalent to the appropriate data type size. Within a particular partition, all free blocks are handled by a single linked list. In other words, if a specific data type representation requires $n$ bytes, the related memory partition is subdivided into blocks of size $n$.

But since all free blocks of a partition are connected on the basis of pointer links, at least as many bytes are used for a block as are needed by the run-time system to implement a pointer. This means that if, for instance, a data type requires two bytes, but a specific run-time system uses 4 bytes to implement a pointer, the appropriate memory partition is subdivided into blocks of 4 bytes.

![Dynamic Memory Partition](image)

Figure 24.20: Dynamic Memory Partition

The total number of reserved partition blocks is user configurable (24.14.8.1). But to ensure robust output code, all partitions are initially divided into as many blocks as are determined on the basis of a worst case execution scenario. This means that, for instance, the code generator counts the number of input/output connections inside a given input FSM module and reserves that much blocks for EventQ entries for the case that all input events are simultaneously scheduled. If this case can be excluded, the total number of EventQ entry blocks can be reduced to save system memory.

### 24.14.7 Output Source Files

The set of generator output files includes ANSI C declaration header (*.h) as well as definition source (*.c) files, which are partially dependent on each other. With respect to older C compilers, all source file names are limited to 8 characters.

Figure 24.21 demonstrates a complete project of code generator output files with appropriate dependencies.

#### 24.14.7.1 Configuration Files

Including different generic template functions and a set of boolean switches, the configuration files (*CFG_Base.h, CFG_OS.c, CFG_IO.c*) are used for code customization. Detailed described in section 24.14.8, this customization set provides all setup options for an optimized and successful execution on a specific run-time system.
In case the selected target code compiler supports strictly conform ANSI C code, only these files are meant to be modified after code generation.

### 24.14.7.2 Data Type Files

All global data types and structures are separately implemented (*Types.h, Types.c*), because these types are included in any other target code section. The data type implementation includes all C structures and functions, in conjunction with input model composite data structures and enumerations. Furthermore, multiple used internal RTOS types are defined within these files.

### 24.14.7.3 FSM Files

The complete state transition diagram representation of an instantiated FSM model and all associated instances are implemented by a separate FSM source file. Additionally, this set of state machine related source files includes an FSM base implementation (*FSM_Base.h, FSM_Base.c*) for commonly used code sections, like the ANSI C implementation of action built-in functions.

### 24.14.7.4 RTOS Files

Almost the complete RTOS functionality, as task and interface interaction, timelines and DE scheduler, is implemented by a single source file (*OS_Main.c*). Only EventQ handling and custom memory management are outsourced to an operating system base implementation (*OS_Base.h, OS_Base.c*). These parts are to be shared with FSMs in case of timer events are present.

### 24.14.7.5 Application Main File

Many embedded and real-time systems require that the C application main function is implemented at top of the appropriate source file, to setup the instruction pointer at the right position. Thus, a generated source code project includes a separate source file (*Main.c*) for this function. Likewise configuration files, this file is optionally backed up in terms of an additional main function customization, for instance, to handle application arguments.
24.14.7.6 Makefile

An additional makefile can be configured and used for comfortable compilation of the generated source files.

24.14.8 Code Customization

An accurate and efficient execution of a generated FSM module application on a specific run-time system is affected by several hardware/software architecture layers. Therefore, in reference to three existing configuration files, the output code customization is divided into three functional parts. Figure 24.22 shows the different customization layers within a hardware/software architecture.

Figure 24.22: Hardware/Software Architecture - Customization Layers

24.14.8.1 General Settings

Global and universal valid settings are handled separately by the CFG_Base.h configuration file. First of all, this customization unit includes a section for data type related settings. Especially important is the range and byte size definition of numeric integer and floating point data types. These settings are highly run-time system dependent.

Additionally, if an input model uses numeric vectors or strings, an appropriate maximum number of elements can be determined within this data type configuration section. The real-time level of the application can be defined by the maximum number of allowed clock ticks between the scheduled time stamp of an event and its real dispatching point in time. In case of hard real-time conditions, the real-time level has to be set to 0.

For all existing custom memory partitions an optimized number of reserved partition blocks can be set.

In conjunction with random number generator built-in functions, a constant global seed is definable or a random seed can be enabled. Latter one is based on the run-time system clock.

Finally, a set of boolean switches is present to enable miscellaneous debug code sections.
24 FSM Domain

24.14.8.2 Porting

MLDesigner FSM code synthesis produces high portable and strictly ANSI C conform code without any run-time system dependencies. In doing so, some processor specific code in C and/or assembly language is necessary to adapt a particular RTOS to a micro-processor or micro-controller. Clock tick handling is thereby the most critical configuration part for several reasons.

First, dependent on appropriate real-time conditions, an accurate clock tick interval has to be determined. This usually requires a certain trade-off, because a short CTI provides in fact exact timing, but on the other hand, the micro-processor must execute a clock tick handling routine frequently. Therefore, a short clock tick can decrease system throughput quite considerably by increasing the amount of micro-processor time spent in the clock tick handling routine.

Secondly, the setup has to ensure an independent and cyclical clock tick occurrence. In this context, an interrupt service routine or a periodic timer signal approach has to be used to call a clock tick handling routine.

In the third place, a code synthesis RTOS has to be configured in such a way that CPU is not permanently blocked by the scheduler, while it is waiting for the next clock tick. Especially in case of a generated application is used as a single task of a superordinate multitasking operating system, it has frequently to give up CPU control, when no more FSM instances have to be performed for current system time. Otherwise, the multitasking operating system is not able to perform concurrent tasks.

Finally, the real-time violation handler has to be configured according to either soft or hard real-time conditions.

Since the clock tick counter is represented by an unsigned integer variable of a finite range, a CTC overflow occurs when the counter exceeds range limit and is instead set back to zero. In case of such an integer range overflow is critical in a specific application area, the RTOS configuration file CFG_OS.c includes a CTC overflow handler function.

Table 24.5 lists all template functions of the CFG_OS.c configuration file.

<table>
<thead>
<tr>
<th>Template Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctSetup</td>
<td>used for clock tick setup, e.g. to start a timer ISR</td>
</tr>
<tr>
<td>ctWrapup</td>
<td>used for clock tick wrap-up, e.g. to stop a timer ISR</td>
</tr>
<tr>
<td>ctWaitNextTick</td>
<td>called whenever scheduler waits for next clock tick, e.g. to give up CPU control</td>
</tr>
<tr>
<td>ctCounterOverflowHandler</td>
<td>CTC overflow handler</td>
</tr>
<tr>
<td>osRealTimeViolationHandler</td>
<td>real-time violation handler</td>
</tr>
<tr>
<td>userSetup</td>
<td>additional user setup, called before ctSetup</td>
</tr>
<tr>
<td>userWrapup</td>
<td>additional user wrap-up, called after ctWrapup</td>
</tr>
</tbody>
</table>

Table 24.5: RTOS Porting Functions
### 24.14.8.3 Interface Configuration

The interface configuration file `CFG_IO.c` is used to define a hardware/software interface for a generated FSM module application within an embedded real-time system. In other words, to connect original FSM module input/output interface ports with appropriate actuator and sensor hardware components. Therefore, the customizable generic `get` and `send` functions, used by the RTOS for interface interaction, are defined within this configuration unit.

Thereby, `get` input functions inform the scheduler about active sensor components and `send` output functions are responsible to enable actuator components. Typically, such actuators and sensors are handled on the basis of specific driver software modules, which are available to configure a hardware/software interface.

Whenever a sensor is active, the associated boolean `get` input function has to be implemented in such a way that possibly input data are set and that it returns true when it is called next time by the RTOS scheduler, to specify presence of a new interface input event.

The `send` output function of a specific interface output port has to activate an associated actuator component, possibly dependent on given output data.

Additionally, an interface configuration unit includes three general template functions, which are described in table 24.6.

<table>
<thead>
<tr>
<th>Template Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ioSetup</code></td>
<td>called before <code>userSetup</code> and used for general interface setup, e.g. to initialize actuators and sensors</td>
</tr>
<tr>
<td><code>ioWrapup</code></td>
<td>called after <code>userWrapup</code> and used for general interface wrap-up, e.g. to reset actuators and sensors</td>
</tr>
<tr>
<td><code>ioUserGetInputs</code></td>
<td>called before scheduler checks interface inputs, e.g. to update sensor related variables</td>
</tr>
</tbody>
</table>

Table 24.6: General Interface Configuration Functions

An example interface configuration is shown in section 24.14.10.5.

### 24.14.9 Code Debugging

ANSI C code synthesis of MLDesigner FSM models supports monitoring of relevant information within several areas of generated FSM module applications.

The debug code sections can be enabled and disabled via the boolean switches defined in the `CFG_Base.h` configuration file.

Debug information are streamed via the `fprintf` C function to the `stderr` output.

#### 24.14.9.1 FSM Behavior

```c
#define FSMn_DEBUG 1
```

The configuration unit for debug settings includes a boolean switch for each involved state machine model, to enable monitoring of appropriate instance behavior. In case that debugging for a specific FSM model is enabled and an instance of this state machine is able to perform a state
change, all related execution steps are displayed. These execution steps include the firing transition, performed actions and the new current state.

### 24.14.9.2 Task Execution

```c
#define OS_DEBUG_TASK 1
```

To validate an accurate RTOS task execution order, especially in conjunction with given real-time conditions, task debugging enables monitoring of the currently executing FSM instance.

### 24.14.9.3 Data Transfer

```c
#define OS_DEBUG_DELIVER_DATA 1
#define OS_DEBUG_EMIT_DATA 1
#define IO_DEBUG_INPUT_DATA 1
#define IO_DEBUG_OUTPUT_DATA 1
```

Based on corresponding debug settings, the complete RTOS task interaction and interface data transfer can be displayed. This means that, whenever an instance output port emits new data to be scheduled for an appropriate instance input port, both procedures, data sending and delivery, can be independently debugged. Respectively, interface debugging can be enabled to monitor received input data and sent output data.

### 24.14.9.4 Dynamic Memory

```c
#define MEM_DEBUG 1
```

Concerning to an optimized memory consumption, memory partitions can be debugged in such a way that all associated partition information are displayed, whenever a memory block is allocated or deallocated.

Additionally, in case the application has been aborted after a specific run-time period, final partition snapshots provide information about how many blocks have been maximum used within this execution duration.

### 24.14.10 Example

The present section exemplifies, how ANSI C code synthesis for MLDesigner FSM models is used to develop a controller application for a Hitachi H8 micro-controller. The task of this controller application is to manage a LEGO® Mindstorms™ block sorter robot (fig. 24.23). In this context, the legOS custom firmware is used on the micro-controller to execute C written programs.

#### 24.14.10.1 Block Sorter Function

The block sorter robot represents a machine to sort different colored blocks in reference to their color lightness. Thereby, only two cases are considered: dark and bright colored blocks.

While the machine is running, blocks are moving one after another on a conveyor belt until a sensor has been reached and activated by the first block. In reaction, the conveyor belt stops and
color lightness of this block is determined. The measured value is compared to a threshold and, based on the result, a sorter unit unloads the block to either left or right side. Once the sorter unit is ready again, the conveyor belt moves on. Additionally, in case of interferences, a start/stop button is present to pause the procedure immediately and to continue it with a second press.

In reference to this block sorter functionality, the underlying LEGO® robot includes two motor bricks to run the conveyor belt and to move the sorter unit. Furthermore, one touch sensor is used to activate the sorter unit at the end of the conveyor belt and a second one represents the start/stop button. Block color lightness is measured by a light sensor brick.

### 24.14.10.2 Modeling

To use ANSI C code synthesis in conjunction with the controller application for the block sorter robot, the complete control function has to be developed on the basis of MLDesigner FSM models. Since the robot mainly consists of a conveyor belt and a sorter unit, the control function is decomposed into a conveyor belt controlling FSM (fig. 24.24) and an automaton, which handles the sorter unit (fig. 24.25). To recompose the control function, a Conveyor Belt FSM instance and a Sorter FSM instance are interconnected inside a higher level controller FSM module (fig. 24.26). This FSM module also defines the input/output interface of the controller application.

### 24.14.10.3 Simulation

After the function of the controller application has been developed in MLDesigner, the resulting controller FSM module can be validated on the basis of an appropriate system model, as shown in fig. 24.27.

Inside this system, an additional mission level environment model is interconnected with a controller FSM module instance. During simulation, this environment model is mainly used to pro-
Figure 24.24: Block Sorter Modeling: Conveyor Belt FSM

Figure 24.25: Block Sorter Modeling: Sorter FSM
24.14 ANSI C Code Synthesis

Figure 24.26: Block Sorter Modeling: Controller FSM Module

Figure 24.27: Block Sorter Simulation: System Model
duce significant signals for the controller FSM module and to evaluate and display corresponding outputs. In other words, it simulates actuators and sensors of the real block sorter robot. The main environment components are thereby an interactive control panel with two buttons (fig. 24.28) and a 3D visualization of the robot (fig. 24.29).

One control panel button is used to put a new random colored block on the conveyor belt of the 3D block sorter model and the second button represents the real start/stop button. Based on control panel activities, the 3D block sorter model visualizes all controller FSM module outputs and produces relevant sensor events, for instance, when a block reached the touch sensor or the light sensor is active. Such a qualified and realistic simulation model enables detailed testing of the controller FSM module functions.

24.14.10.4 Code Synthesis

To produce functional equivalent C code, the controller FSM module is used as input model for the ANSI C code generator. In this context, name and path of the target directory for the output source
files have to be determined by the configuration dialog. Furthermore, a minimum integer size of 8 bit can be selected, because integer values within the controller application will not exceed the corresponding range limit. After the configuration dialog has been accepted by the Generate button, the code generator produces the ANSI C implementation of the controller FSM module and creates all related source files inside the selected directory.

### 24.14.10.5 Code Customization

Before the generated code can be downloaded and executed on the micro-controller, it has to be customized and compiled on a host computer, which provides the legOS API and C compiler. In this context, the `CFG_Base.h` configuration file can be left unmodified for several reasons. First, no data type settings are necessary, because neither character strings nor numeric vector data types are used in the controller application and integer size has already been defined by the configuration dialog. Secondly, no random seed has to be configured in matters of involved random number generator built-in functions. In the third place, memory optimization in connection with the number of reserved partition blocks can be disregarded. Only an integer partition and the EventQ entry partition are present, each with 6 predefined blocks. Finally, all debug options must be left disabled in consideration of legOS is not strictly ANSI C conform and does not support output streams.

The RTOS clock tick configuration within the `CFG_OS.c` file can be limited to modifications of the `ctWaitNextTick` function, since the Hitachi H8 micro-controller is not a real-time system. In doing so, this method is implemented in such a way that after all tasks for the present system time have been processed, the RTOS scheduler gives up control to legOS for 100ms. Afterwards, the clock tick handler is called.

To setup the controller application input/output interface, all input `get` as well as output `send` functions of the `CFG_IO.c` file have to be configured using motor and sensor driver functions. These functions are provided by the legOS API. Additionally, three local boolean variables are used in connection with sensor activities and motor speeds are initialized by the `ioSetup` function. As a result of legOS is not strictly ANSI C conform and does not support standard input/output streams, the corresponding include statement has to be removed from the `Types.h` source file to satisfy the legOS C Compiler. Except this minor modification, all other non-configuration files can be left untouched.

Here are the customized configuration files of the block sorter controller application.

**CFG_Base.h** (general settings):

```c
#ifndef __CFG_Base_h
#define __CFG_Base_h

typedef char int8bit;
typedef unsigned char uint8bit;

typedef short int int16bit;
typedef unsigned short int uint16bit;

typedef long int int32bit;
```
typedef unsigned long int uint32bit;

typedef unsigned char IntegerT;
typedef double FloatT;

/* maximum number of allowed clock ticks between
the scheduled time stamp of an event and its real
dispatching point in time */
#define REAL_TIME_LEVEL 0

/* number of blocks in the IntegerT memory partition */
#define MEM_NUM_IntegerT_BLOCKS 6

/* number of blocks in the event queue memory partition */
#define MEM_NUM_EVENT_QUEUE_ENTRIES 6

#define SEED 123456789
#define USE_RANDOM_SEED 0

/* debug FSM "ConveyorBelt" */
#define FSM0_DEBUG 0

/* debug FSM "Sorter" */
#define FSM1_DEBUG 0

/* debug runtime errors */
#define DEBUG_RUNTIME_ERRORS 0

/* debug task execution */
#define OS_DEBUG_TASK 0

/* debug task input data */
#define OS_DEBUG_DELIVER_DATA 0

/* debug task output data */
#define OS_DEBUG_EMIT_DATA 0

/* debug interface input data */
#define IO_DEBUG_INPUT_DATA 0

/* debug interface output data */
#define IO_DEBUG_OUTPUT_DATA 0

/* debug dynamic memory allocation and de-allocation */
#define MEM_DEBUG 0
24.14 ANSI C Code Synthesis

#include <unistd.h>
#include "Types.h"

/* external clock tick counter declaration */
extern uint32bit ct_Counter;

/* external os abort flag declaration */
extern BooleanT os_Abort;

/* external clock tick handler declaration */
void ctHandler(void);

/* external error function declaration */
void osErrorAbortRun(const StringT* pMessage);

void ctSetup(void)
{
}

void ctWrapup(void)
{
}

void ctWaitNextTick(void)
{
    msleep(100);
    ctHandler();
}

void ctOverrunHandler(void)
{
}

void ctCounterOverflowHandler(void)
{
}
void

void tmrTimeStampOverflowHandler(void)
{
}

void

void userSetup(void)
{
}

void userWrapup(void)
{
}

CFG_IO.c (input/output interface configuration):

#include <dmotor.h>
#include <dsensor.h>
#include <rom/lcd.h>
#include <conio.h>

#include "IO_Ports.h"
#include "Types.h"

/* external error function declaration */
void osErrorAbortRun(const StringT* pMessage);


static BooleanT io_ButtonOnOff_Pressed = 0;
static BooleanT io_TouchSensor_Pressed = 0;
static BooleanT io_LightSensorActive = 0;

/* ==== USER I / O CONFIGURATION FUNCTIONS ==== */

void

void ioSetup(void)
{
    motor_a_speed(100);
    motor_c_speed(100);
}

void

void ioWrapup(void)
{
}

}
ioUserGetInputs(void)
{
}

/* ==== USER INPUT INTERFACE FUNCTIONS ======== */

static BooleanT
get_ButtonOnOff(IntegerT* pData)
{
    if (TOUCH_1)
    {
        if (io_ButtonOnOff_Pressed == 0)
        {
            io_ButtonOnOff_Pressed = 1;
            return 1;
        }
    }
    else
    {
        io_ButtonOnOff_Pressed = 0;
    }
    return 0;
}

static BooleanT
get_LightSensor(IntegerT* pData)
{
    int tLight;
    if (io_LightSensorActive)
    {
        tLight = LIGHT_2;
        *pData = tLight;
        cls();
        lcd_int(tLight);
        return 1;
    }
    return 0;
}

static BooleanT
get_TouchSensor(IntegerT* pData)
{
    if (TOUCH_3)
    {
        if (io_TouchSensor_Pressed == 0)
        {
            io_TouchSensor_Pressed = 0;
        }
    }
}
io_TouchSensor_Pressed = 1;
    return 1;
  }
  }
else
  io_TouchSensor_Pressed = 0;

  return 0;
}

/ * ===== USER OUTPUT INTERFACE FUNCTIONS ====== */

static void send_BandOn(IntegerT pData)
{
  motor_a_dir(fwd);
}

static void send_BandOff(IntegerT pData)
{
  motor_a_dir(brake);
}

static void send_SorterMotor(const EnumT* pData)
{
  if (pData->mIndex == 0)
    motor_c_dir(brake);
  else if (pData->mIndex == 1)
    motor_c_dir(fwd);
  else
    motor_c_dir(rev);
}

static void send_LightSensorOn(IntegerT pData)
{
  io_LightSensorActive = 1;
  ds_active(&SENSOR_2);
}

static void send_LightSensorOff(IntegerT pData)
{
  io_LightSensorActive = 0;
ds_passive(&SENSOR_2);
}
Chapter 25

NS2 Domain

25.1 Introduction

Many modern systems depend on a network environment. The complexity of protocols used in network environments leads to uncertainty in the design process. It is often difficult to size network equipment properly. For example, how much bandwidth has a customer with his laptop inside an airplane, if all the other 300 passengers want to use Internet as well? Or - what impact has a certain error probability on the throughput of the Reno TCP implementation - what is the impact if a New-reno, Vegas, Sack or Fack TCP implementation is used?

The system level design tool MLDesigner is a very good tool for building models of different scenarios, including aspects of architecture, performance and functionality. MLDesigner currently provides a limited set of specialized libraries for network protocols and network hardware. In contrast, NS2 - a free network simulator - has a variety of models for network simulations, but no graphical user interface and NS2 lacks the broad functionality MLDesigner provides within its framework of different domains working together in complex simulations.

The solution for solving complex problems would be a mixture of MLDesigner’s multi purpose and NS2’s network protocol and simulation engine. This mixture is implemented with the NS2 domain inside MLDesigner. The multi domain concept allows using different models of computation inside one simulation. The NS2 domain provides an interface to all NS2 features and models within the GUI of MLDesigner.

The next section describes MLDesigner and NS2 more in detail and then introduces the NS2 domain of MLDesigner and shows how it can be used with the initial components defined for the NS2 domain. It then shows how to create new MLDesigner NS2 primitives.

25.2 MLDesigner and NS2

The following sections give a short overview on MLDesigner and NS2, listing the advantages and disadvantages of both tools regarding network simulations in a design flow process.

25.2.1 Modeling Networks with MLDesigner

MLDesigner is a general-purpose modeling tool and has only a limited number of models for network hardware and protocols. MLDesigner provides, as an add-on, a model library called
“Network Building Set” [Zinb] that can be used as a framework to model network stacks including network protocols and hardware. This library currently contains models for UDP, a simplified version of TCP, a simple routing mechanism and some link layer models. Several wireless network models are also available for MLDesigner, including a very detailed [Ste] model and several more abstract [Zina] examples. The detailed 802.11 model is a direct translation of the formal 802.11 SDL description, with all known errors corrected. The abstract WLAN models use a simplified FSM model.

Experienced modelers could use the models provided in the Network Building Set as the starting point for developing additional network protocols. For those who want to use existing models, another solution is needed. One attractive solution is to take advantage of the rich collection of NS2 network protocol models.

### 25.2.2 About NS2

#### 25.2.2.1 General Introduction

NS2 is Network Simulator version 2 [NS2a], a discrete event network simulator that provides a rich collection of networking models. The simulator is written in C++ and it uses OTcl [OTc] as command and configuration interface. Currently NS2 does not provide any kind of graphical setting up network topology, or configuring model elements. In addition, it does not have the capabilities of MLDesigner (e.g., data structures, resources, etc) for modeling the hardware and software operations that generate and process networked messages.

#### 25.2.2.2 Existing Models inside NS2

NS2 provides models for:

- network nodes with multiple configuration possibilities.
- links for connecting the nodes together (e.g. simplex and duplex point-to-point links; links for multi-access LANs, wireless networks and broadcast media). Bandwidth, delay, queue type, error and loss models as well as dynamics, like “link goes down” and “link goes up”, are configurable.
- different kinds of queues, such as Drop Tail Queuing, Fair Queuing, Stochastic Fair Queuing, Deficit Round Robin Queuing, Random Early Detection Queuing, Class Based Queuing, and more. Each queuing mechanism offers its own parameters for configuration.
- routing protocols that route the packets from source to destination;
- Agents (models that are related to nodes) that are responsible for sending and receiving data. For example, NS2 provides agents for the following network protocols:
  - Several TCP (Transmission Control Protocol) implementations like Reno, New-reno, Vegas, Sack, Fack TCP, and more,
  - UDP (User Datagram Protocol),
  - SCTP (Stream Control Transmission Protocol) in several variations,
  - SRM (Scalable Reliable Multicast), and
some more

- timers inside the simulation;
- error models for packet losses at the link layer level;
- Local Area Networks, with channel properties like propagation delay, configurable Link Layer and Medium Access Layer and LAN routers.

A full description of all modeling features inside NS2 is given in [NS2a].

### 25.2.2.3 OTcl as User Interface

Modeling in NS2 is done by writing scripts for the OTcl interpreter\(^1\). OTcl [OTcl] is an object oriented extension to Tcl. A good tutorial is available at [ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/tutorial.html](ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/tutorial.html). The NS documentation and some tutorials guide the user how to write scripts for simulations.

Source code 25.1 shows a sample for a simulation with two nodes, two agents and a traffic source. For more details about the NS2 scripting interface, there are several tutorials available in the Internet [NS2b]. Furthermore [NS2a] is a good reference for all features and how they are used inside Tcl scripts.

### 25.2.2.4 C++ as simulation backend

Behind the OTcl user interface, there is a large C++ library that includes implementations for all the models that are inside NS2 and for the discrete event simulation engine itself. The usage of C++ allows the NS2 framework to achieve its fast simulation speeds - in comparison to its accuracy. The linkage between OTcl and C++ is done via the tclcl toolkit [tcl]. All important classes in OTcl have implementations inside the C++ space. In the first chapters of [NS2a], it is very well explained how OTcl and C++ work together forming the NS2 environment.

### 25.2.2.5 Summary of NS2

NS2 provides a huge amount of networking protocols. The use of OTcl as user interface provides a maximum of flexibility for the user. Advanced users are able to use Tcl command structures, like loops and iterations, to build complex scenarios. But, to use all the features, a lot of documentation has to be read. At the end, after building up the simulation, the designer has only a script, which makes difficult its presentation to customers or at conferences.

The following section shows the NS2 domain inside MLDesigner and how it is used from a modeler’s point of view.

### 25.2.3 Linking MLDesigner and NS2

MLDesigner 2.5 introduces a new modeling domain (currently MLD_Experimental/NS2) that can be used to create, launch, and control the execution of NS2 network traffic models from inside MLDesigner. These NS2 models can be run stand-alone or they can be integrated into MLDesigner models developed in other domains, such as Discrete Event.

\(^1\)Of course, advanced users will code in C++, too
#Create a simulator object
set ns [new Simulator]

#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Define a 'finish' procedure
proc finish {} {
    global ns nf
    $ns flush-trace
    #Close the trace file
    close $nf
    #Execute nam on the trace file
    exec nam out.nam &
    exit 0
}

#Create two nodes
set n0 [$ns node]
set n1 [$ns node]

#Create a duplex link between the nodes
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

#Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize 500
$cbr0 set interval 0.005
$cbr0 attach-agent $udp0

#Create a Null agent (a traffic sink) and attach it to node n1
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0

#Connect the traffic source with the traffic sink
$ns connect $udp0 $null0

#Schedule events for the CBR agent
$ns at 0.5 "$cbr0\_start"
$ns at 4.5 "$cbr0\_stop"
#Call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"

#Run the simulation
$ns run

Source Code 25.1: NS2 script for a two node scenario
25.3 Working with the MLDesigner NS2 Domain

All NS2 features can be encapsulated in primitives or modules and used together with MLDesigner DE, FSMs, etc. This is done by using NS2 as a co-simulator. The encapsulated models will communicate with the NS2 process that is started by the MLDesigner NS2 domain during simulation startup. This is a very loose binding of MLDesigner and NS2. This allows much flexibility on both sides. New versions, or versions of NS2 that have been enriched by custom features can be used.

25.3 Working with the MLDesigner NS2 Domain

The rest of this chapter discusses how to link MLDesigner and NS2 so they work together. It starts with configuring the two programs to work together. Next we describe the NS2 domain from the modeler perspective. This is followed by a description of some example systems (see MLD_Experimentals/NS2/Demos). The chapter closes with an introduction to the primitive API and how new features of NS2 can be used within MLDesigner.

25.3.1 Getting Started

As already mentioned in the introduction of this section, MLDesigner uses NS2 as a co-simulator, that is MLDesigner starts, controls and stops the execution of NS2 models using a standalone NS2 process. Before that can be done, it is necessary to perform the following steps:

- Get and install an NS2 simulator
- Build the NS2 binary with \texttt{-rdynamic} (only on non Solaris systems)
- Set environment variable \texttt{NS2} to the NS2 Source Directory
- Add the \texttt{bin} directory of the NS2 distribution to the \texttt{PATH} variable

In the following paragraphs details for each step are provided.

25.3.1.1 Get and Install NS2

The NS2 home page is \url{http://www.isi.edu/nsnam/ns}. Under \url{http://www.isi.edu/nsnam/ns/ns-build.html} a detailed building instruction is provided. Like mentioned on the site, the most comfortable way to get and install NS2 is to use the all-in-one packet. The following instructions are based on an installation of the all-in-one packet. If NS2 was installed from pieces, the instructions should be applied analogously. NS2 should be compiled successfully before proceeding with the following steps.

25.3.1.2 Build the NS2 Binary with \texttt{-rdynamic}

To build up a communication between the NS2 process and MLDesigner, some additional implementations have to be included into the NS2 process. To be compatible with coming NS2 versions, these implementations are not part of the built NS2 binary, but they are compiled and loaded into the NS2 process in the setup phase of the MLDesigner simulation. To be able to load these compiled implementations into the NS2 process, the NS2 binary has to be linked with the \texttt{-rdynamic}
linker option on several operating systems (excluding Solaris systems). If NS2 is compiled with the standard makefiles, it is enough to remove the NS2 binary and recompile it with `make LDFLAGS=-rdynamic`. Assuming NS2 was compiled in `$HOME/ns-allinone-2.27/` using the all-in-one packet, the following steps should be performed:

- change to the NS2 source directory (`cd $HOME/ns-allinone-2.27/ns-2.27`)
- remove ns binary (`rm -f ./ns`)
- rebuild the ns binary with `-rdynamic` (`make LDFLAGS=-rdynamic`)

### 25.3.1.3 Set NS2 Environment Variable

MLDesigner needs to know, where NS2 was compiled and where to find the NS2 executable. This can be specified using the Target parameters `NS2SourceDir` and `SimulatorCommand`. As default values `NS2SourceDir` is set to `$NS2` and `SimulatorCommand` is set to `$NS2SourceDir/ns`. The best way of telling MLDesigner where to find the NS2 program, is to set the environment variable `$NS2` to the directory where NS2 was installed to before starting MLDesigner. If NS2 was installed to `$HOME/ns-allinone-2.27/` using the all-in-one packet, `$NS2` should be set to `$HOME/ns-allinone-2.27/ns-2.27/`. On Linux distributions using bash as shell this can be done with an export command:

```
export NS2=$HOME/ns-allinone-2.27/ns-2.27
```

For adding these environment variable permanently, the export command shown above should be copied into your `.bashrc` file or equivalents.

### 25.3.1.4 Add the `bin` directory of the NS2 distribution to the `PATH` variable

You can use NS2 postprocessing tools like `nam` and `xgraph` from within MLDesigner. To have all these necessary tools together, it is recommended to use the all-in-one package of the NS2. After installing this package, the created `bin` directory contains all executables and must be added to the `PATH` environment variable in order to use them within the MLDesigner simulation environment. If NS2 was compiled in `$HOME/ns-allinone-2.27/` using the all-in-one packet, `$PATH` should be extended by `$HOME/ns-allinone-2.27/bin`. On Linux distributions using bash as shell, this can be done via:

```
export PATH=$PATH:$HOME/ns-allinone-2.27/bin
```

Again, we recommend you to copy the export command to the `.bashrc` file or equivalents.

### 25.3.2 Assembling NS2 Models to Build Simulations

The next section provides an overview about modeling inside the NS2 domain in general, and shows examples how the model primitives can be used to build sample scenarios.

---

2 If `tcsh` is used, the appropriate command is: `setenv  NS2 $HOME/ns-allinone-2.27/ns-2.27`  
3 The `.bashrc` file should be stored in your home directory  
4 If `tcsh` is used, the appropriate command is: `setenv  PATH $HOME/ns-allinone-2.27/bin`  
5 Again, the NS2 domain is located in the experimental domains of MLDesigner `MLD_Experiments/NS2`
25.3 Working with the MLDesigner NS2 Domain

25.3.2.1 General Modeling Guidelines

People who are already familiar with NS2 should have little trouble adapting to the NS2 domain. People who are unfamiliar with NS2 can find many NS2 tutorials on the Internet; we recommend they take advantage of them to learn NS2 basics.

The basic elements for modeling inside NS2 are nodes, links, agents and applications. Nodes represent computers or other network equipment. They are connected together via Links. Agents are connected to Nodes and they can send and receive data over them. In addition agents can perform protocol actions on the data. For example transport protocols like UDP and TCP are modeled as agents. The last basic components are applications. They are associated with agents and control the amount of data that is sent by the sending agent. Applications are for example a ftp process or a periodic sender.

Other blocks with different purposes exist, and they are described step by step in the modeling example below.

In general, the NS2 domain uses its models to define a network topology. Each block in the MLDesigner model is mapped to one block in the network topology inside the NS2 co-simulator. These blocks are connected together via MLDesigner relations to define the connectivity between these blocks. The relations in this case do not transport any data. Therefore, the ports of the blocks are set to bi-directional ports. For that reason, usually InOut ports are used. An application is connected via its Agent single InOut port to an agent’s Applications multi InOut port. A single InOut port should be connected to one another single InOut port, whereas a multi InOut port can be connected to more then one single InOut ports. Since Applications is a multi InOut port, agents can be associated to several applications.

The next section is a small tutorial for the first steps within the NS2 domain. To introduce all types of blocks including a full description of their parameters is not intended, rather some basic guidelines for the first modeling steps are shown in the sample systems.

25.3.2.2 A Hello World Scenario

Hello World in terms of networking, can be defined as a traffic source connected to a sink. Figure 25.1 shows such a simple NS2 system. This example is found at MLD Experimentals/NS2/Demo/HelloWorld. The blocks and their main parameters are described below.

Traffic Source - CBR is an application that generates traffic according to a deterministic rate. The parameters used in this scenario are:

PacketSize specifies the size of the packets to send (set to 500)

Interval specifies the time in between two packets (set to 2.0)

StartTime specifies when the traffic source starts sending; if it is set to a negative value, the application does not start automatically⁶ (set to 0)

StopTime specifies when the application stops sending; a negative value causes no stopping (set to -1)

This source starts at time 0 and sends packets of 500 bytes every two seconds.

⁶Other ways to start applications are possible and are described later
Agents - UDP and MLDSimpleSink  The CBR application is connected to an UDP agent. Parameter DestinationAgent should be set to the agent that is supposed to receive the data of the UDP sending agent. The full-qualified instance name of the receiving agent can be put here. Beside the full-qualified instance name, it’s also possible to put in an alias for the receiving agent. Aliases are user-defined names that refer to model elements. They can be specified via the NS2Name parameter.

MLDSimpleSink’s parameter NS2Name and DestinationAgent of UDP are set to mldsink to “connect” the source agent with the sink agent.

Nodes and Links  The agents are each connected to a Node. These two nodes are connected via a DuplexLink model in between them. DuplexLink can be configured by:

Bandwidth that defines the bandwidth of the link. It is specified as a floating point value, optionally suffixed by a ‘k’ or ‘K’ to mean kilo-quantities, or ‘m’ or ‘M’ to mean mega-quantities. A final optional suffix of ‘B’ indicates that the quantity expressed is in Bytes per second. The default is bandwidth expressed in bits per second. For example all of the following values are equivalent:

1.0m
1.0Mb
1000k
1.0e6
125kB

The bandwidth in the example is set to 1Mb - 1 Mega bit per second.

7That is the name of the system and the module hierarchy down to the receiving agent; in the example it is “HelloWorld.MLDSink#1”
25.3 Working with the MLDdesigner NS2 Domain

**Delay** defines the propagation delay of the channel. It’s value is measured in seconds. For convenience, this floating point value can be suffixed by ‘m’, ‘n’ or ‘p’ to express time in milli-seconds, nano-seconds or pico-seconds. It is safe to add a ‘s’ to reflect the time unit in seconds. In the example, the value is set to 10 ms.

**Accessing Simulation Results - Dump** MLDSimpleSink modules pass an integer particle through their **Size** Port whenever a packet is received by this sink agent. The value of this particle is set to the size of the received packet. The port is connected to a **Dump** block, that simply dumps its identification together with the current simulation time and the received particle to standard output.

**Simulation Results** The simulation results are shown below; **RunLength** was set to 10. The source is sending every 2 seconds. A packet sized 500 Bytes takes 0.004 seconds over a 1 Mb medium. The channel propagation delay of 10ms is added to packet delivery time.

<table>
<thead>
<tr>
<th>Simulation 448 Iteration 1 started Thu Feb 19 12:31:10 2004 on host 'greer'</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld.Dump#1: at: 2.014: 500</td>
</tr>
<tr>
<td>HelloWorld.Dump#1: at: 4.014: 500</td>
</tr>
<tr>
<td>HelloWorld.Dump#1: at: 6.014: 500</td>
</tr>
<tr>
<td>HelloWorld.Dump#1: at: 8.014: 500</td>
</tr>
</tbody>
</table>

**Simulation 448 Iteration 1 finished Thu Feb 19 12:31:10 2004 CPU time 0.000000 seconds**

Simulation results for the Hello World scenario

### 25.3.2.3 Eval, AtJob and Script

As mentioned in sec. 25.2.2, the NS2 simulator has an OTcl interface for simulation set up and control. The NS2 domain of MLDdesigner provides primitives that allow the access to this OTcl interface. The **Eval** primitive allows the command given in parameter **Command** to be executed inside the NS2 co-simulator. As an example, the command

```otcl
globals::puts "Agent/TCP baseclass is [Agent/TCP info superclass]"}
```

used in the **Command** parameter of an **Eval** primitive will print the string

`Agent/TCP baseclass is Agent`

to standard out.

To execute commands during simulation, the **AtJob** block can be used. The time when the command is executed can be defined as parameter as well as the command itself.

**Script** executes an OTcl script, specified in parameter **Script** inside the NS2 co-simulator. It is possible to execute scripts originally written for the standalone NS2 simulator. To do this, some modifications to the script have to be done:

- Because MLDdesigner creates its own instance of an NS2 Simulator, this script (modified for use in the NS2 domain) should be used. That means, inside the script there should no command like `set ns [new Simulator]`. As simulators inside NS2 scripts are often called ns, a good way to use MLDdesigner’s Simulator instance is to set ns as alias to $NS2_simulator instead of creating your own instance. The common simulator instance allows MLDdesigner modules to interact with the scene you have defined inside your script.
• Do not run the simulator manually. MLDesigner controls the execution of it’s slave NS2 process. The run method of the simulator is called by MLDesigner.

• Do not call exit inside the script. MLDesigner controls startup and exit of its slave NS2 process. MLDesigner kills the slave if the RunLength has been reached - you don’t need to take care about it.

The MLDesigner NS2 domain can execute anything that NS2 can execute. Figure 25.2 shows a system that only contains instances of the Eval (see above) and the Script primitive. The contents of the script executed by the instance of the Script primitive is set as it is instance label. It is an original script from an NS2 tutorial. The necessary changes for MLDesigner are marked by red comments. Note, that even the call for nam (the network animator [nam]) is possible. Figure 25.3 shows a screen shot of the nam executed inside the script.

![Script](image)

Figure 25.2: MLDesigner executing an NS2 Script

### 25.3.2.4 Using the Network Animator (NAM)

The network animator [nam] can display network topologies and simulation results that are recorded by NS2. The flow of the packets exchanged during simulation can be monitored, fur-
25.3 Working with the MLDesigner NS2 Domain

Figure 25.3: Nam analyzing the Script-System

thermore packet losses can be animated using nam. To start nam at simulation end for analyzing simulation results, an instance of the Nam Tracer primitive should be placed into the scenario. The Nam Tracer writes trace information into the via parameter NamTraceFilename specified file. If the parameter is empty, a temporary file is used. The nam program using this file is started after simulation if parameter NamStartOnWrapup is set to TRUE. Figure 25.4 shows a small scenario with 9 nodes, a deterministic data source sending over UDP and a ftp application sending over a TCP connection. The links are modeled as duplex links. After simulation is finished, nam is started and packet flows can be analyzed. Figure 25.5 shows a screen shot of the nam instance started. It shows the 9 nodes, as well as their connecting links and some packets flowing over the links.

25.3.2.5 Embedding an NS2 Network into a DE System

A big advantage of MLDesigner is its ability to combine different models of computations inside one simulation. An NS2 model can be embedded into a discrete event system model. In future releases, it will be possible to include DE Models into NS2 models too and to mix them with models of other domains, like SDF. For example the system DE Wormhole (fig. 25.6) shows an NS2 model that is embedded inside a DE system. A clock triggers every 100 ms the CBR application inside the NS2 module to start sending packets. After a delay of 50 ms the CBR model is stopped. Inside NS2Module, the topology is a very simple one, two nodes connected by a duplex link. Figure 25.8 shows three sending periods out of the Xgraph plot.

---

8The NamTracer scenario is part of the Tutorials that can be found in the Demo library of the NS2 domain
9The DEWormhole scenario is part of the Tutorials that can be found in the Demo library of the NS2 domain
Figure 25.4: The “Using Nam” system

Figure 25.5: Nam output for the demo system
25.3 Working with the MLDesigner NS2 Domain

Figure 25.6: DE Wormhole System

Figure 25.7: NS2 Module inside the DE Wormhole system

Figure 25.8: XGraph results for the DE Wormhole system
25.3.3 Writing New NS2 Primitives

The previous section provided an overview of existing NS2 domain primitives. This section describes basic concepts for writing new primitives. First, an introduction is given before the API is described in general, followed by example code pieces. Knowledge about the NS2 OTcl command interface is usually needed for design and implementation of NS2 primitives.

25.3.3.1 General Introduction

The NS2 domain’s primitives are written in the Ptolemy Language (ptlang). The common interface for creating, modifying and compiling primitives inside the NS2 domain is similar to other MLDesigner domains. For details about how to write primitives in MLDesigner please consult ch. 13.

This section is not an introduction or full reference on how primitives are written in MLDesigner, but, rather, a discussion of how to create NS2-specific primitives.

On the other hand, this document is not a replacement of the NS2 documentation. Because writing primitives for the NS2 domain is often something like writing OTcl scripts for the NS2 simulator, it is necessary to know the basic concepts behind the NS2 OTcl API. The NS2 documentation [NS2a] is a very good reference guide of NS2. Furthermore there are a lot of tutorials for NS2 available on the Internet.

This section describes what is necessary to know for writing NS2 primitives, assuming that basic concepts of writing primitives using NS2 are known.

25.3.3.2 Communication Architecture: MLDesigner and NS2

Figure 25.9 shows the communication architecture of the MLDesigner NS2 domain and how it is embedded into MLDesigner’s domain framework.

![Architecture of MLDesigner and NS2](image)

**Figure 25.9: Architecture of MLDesigner and NS2**

The arrows in the diagram represent the possible ways of communication:

**Arrow 1** A primitive inside the NS2 domain controls the NS2 simulator via the simulator’s OTcl interface. Everything that can be done inside OTcl NS2 scripts, can be done by primitive code. This provides an easy way for accessing all features that NS2 provides. Sec. 25.3.3.4 shows how this interface is used.
Simulations inside the NS2 slave are performed as C++ code. This allows very high simulation speeds. Simulation results, or events in the NS2 slave process (for example the arrival of a network package) can be forwarded from the NS2 core to MLDesigner’s primitives. A common interface is provided and its usage is shown in examples.

Like primitives in other domains, NS2 primitives can have input ports. NS2 primitives are able to receive particles from other MLDesigner domains, e.g. the Discrete Event domain. The wormhole example in the modeling guide shows how NS2 primitives can react on inputs from outside.

Simulation results (usually produced by the NS2 simulator) can be passed to other domains or other primitives inside the NS2 domain. This is done over ports.

It is possible to use all ways of communication together inside one primitive. The arrows in the figure are only some examples and do not show all possible communication.

For writing new primitives, it is necessary to have a closer look at the simulation phases. After pressing the Go button, simulations are divided into two major phases: simulation setup and simulation run. In the setup phase, the topology is defined and general simulation parameters are set. Defining the topology includes creating nodes, links, agents, applications, etc. During a simulation run, packets are forwarded between nodes, agents perform their protocol logic and data is produced, sent, received and processed. During setup phase, NS2 primitives can advise the NS2 simulator to create the topology by using the OTcl interface. In this phase no particles flow through any port of any NS2 primitives.

In NS2 scripts, this phase includes all commands given before the NS2 simulator is started (via $ns run). At this stage the `setup()` and `begin()` are called.

Inside the `begin` method, primitives can execute commands in the NS2 simulator’s OTcl environment. This can be done very easily using the `EVAL( ... )` macro. Every construct that can be streamed into a String stream can be used as parameter. Examples for correct `EVAL` statements are shown in source code 25.2. If the OTcl interpreter of NS2 throws an error, the error message is shown in the MLDesigner graphical user interface and the simulation is aborted.

In source code 25.3 the `AtJob`’s `begin` method is shown. The primitive has two parameters: `Time` (float) and `Command` (string). The example shows how the parameters are used in the primitive’s code. As explained in sec. 25.3.2.3, the `Simulator` instance inside the NS2 context is called `$NS2_simulator`.

While it is possible to define everything, including topology, during simulation run, but usually it is defined in advance.
The Begin Schedule  In MLDesigner domains, the begin method of primitives is normally called exactly once in a non-predictable order. Inside the NS2 domain, pre-defined begin phases are passed through in ascending order. Each primitive is responsible to add itself to the phase or phases in which its begin method should be called. This is done using the method addToBeginPhase(int phase) inside the primitive’s constructor. This method can be called with one of the following predefined identifiers as argument:

- NS2Star::SIMULATOR
- NS2Star::SIMULATORCONFIG
- NS2Star::NODE
- NS2Star::AGENT
- NS2Star::AGENTCONNECT
- NS2Star::APPLICATION
- NS2Star::NODELINK
- NS2Star::USER

For illustration, source code 25.4 shows the hand-written pthlang source code of Eval. In the constructor, addToBeginPhase(NS2Star::USER) is called to schedule this primitive for the USER phase. Therefore, the command given by parameter is evaluated after all nodes, agents, applications and links are already initialized.

In source code 25.5, constructor and begin of the UDP agent are shown. Data sending agents like UDP have to be connected to a destination agent using the connect command. But both source and destination agent must be created before they can be connected to each other. If begin is called in AGENT phase, it’s possible that the destination agent has not yet been set up (because it is scheduled in AGENT phase, too). The solution is to schedule the UDP agent primitive twice, once in AGENT phase to set up the agent, and again in AGENTCONNECT phase to connect it to its destination. This is done in the 3rd and 4th lines. In the begin method, getBeginPhase() is used to find the correct phase to schedule the command.

11. It is even possible to re-schedule a primitive for a later phase during execution of begin.
12. Actually, this is a simplified version of Eval, that executes its command during USER begin phase. The built-in Eval supports parameters for specifying in which begin phase the command should be executed.
13. This is a simplified version, too. The built-in version uses methods of base classes and does a better error handling if the destination agent is not known inside the OTcl interface.
25.3 Working with the MLDesigner NS2 Domain

```cpp
constructor
{
    addToBeginPhase(NS2Star::USER);
} begin
{
    EVAL((const char *)Command);
}
```


```cpp
constructor
{
    addToBeginPhase(NS2Star::AGENT);
    addToBeginPhase(NS2Star::AGENTCONNECT);
} begin
{
    switch (getBeginPhase()) {
    case NS2Star::AGENT :
        // Create Agent
        EVAL("set\" << fullName() << \"[new Agent/UDP]\";
        attachToNode();
        setAlias();
        break;
    case NS2Star::AGENTCONNECT :
        if (((const char *)DestinationAgent)[0])
            EVAL("$NS2_simulator_connect\" << fullName() << \"}\";
        break;
    } // of switch
}
```

Source Code 25.5: Hand-written Ptlang Code of “UDP Agent”
class NS2Star : public Star
{
    // ...
    public:
        static const int RESOLUTION = 1000;
        static const int SIMULATOR = 1 * RESOLUTION;
        static const int SIMULATORCONFIG = 2 * RESOLUTION;
        static const int NODE = 3 * RESOLUTION;
        static const int AGENT = 4 * RESOLUTION;
        static const int AGENTCONNECT = 5 * RESOLUTION;
        static const int APPLICATION = 6 * RESOLUTION;
        static const int NODELINK = 7 * RESOLUTION;
        static const int USER = 100 * RESOLUTION;
    // ...
}

Source Code 25.6: Declaration of pre-defined begin phases

to determine the actual begin phase (refer to line 8). Statements in lines 10-15 are called the first time begin is executed to create the agent (line 11), lines 17-19 is executed after all other agents have been created.

**Naming Conventions** Many primitives in NS2 are place holders for instances inside the NS2 simulator (e.g., agents, nodes, applications etc.). This implies that during begin phase the corresponding instance in the NS2 simulator has to be created. As instance name, the full name of the primitive is used. Agent A#1 in system S is stored in the S.A#1 variable. You can access NS2 objects by using the fully qualified name of their corresponding MLDesigner primitives. Because the full name contains '#', the variable name has to be embedded in curly braces if it is used in OTcl commands. Lines 18 and 19 of source code 25.5 show how this is done.

**Future and Custom Begin Phases** More begin phases will be added in future MLDesigner versions. Of course, it is possible to schedule primitives between phases and set the order of execution relative to one-another because the phase is an integer value that is not limited to the pre-defined phases. Any other positive integer value will work as well. The primitives are executed in ascending order depending on their scheduled phases. The pre-defined phases are set to constant values with difference of 1000 in between. It is possible to schedule own primitives to be executed between AGENTCONNECT and APPLICATION by calling addToBeginPhase(NS2Star::AGENTCONNECT + 500) in their constructors. The mapping of well-known phase names to integers may change in future releases, so it is a good idea to use the pre-defined values as often as possible - for example use NS2::AGENTCONNECT + 500 instead of 5500. Source code 25.6 shows the declarations for the well-known phases as it can be found in $MLD/MLD_Experimental/NS2/kernel/NS2Star.h.

**Using an Own Simulator Instance** The NS2 domain creates an instance of the NS2 Simulator class called $NS2_simulator. Many primitives use this Simulator instance and they bank on that naming convention. To give modelers the possibility to create the Simulator instance themselves rather than use the standard creation routine, a primitive can be scheduled at NS2Star::SIMULATOR phase. This primitive has to create a Simulator instance called $NS2_simulator. One of these primitives can be placed into an NS2 scenario. If there is no such
primitive, the standard command "set NS2\_simulator [new Simulator]" is executed before any begin method is called. If there is exactly one primitive scheduled for the SIMULATOR phase, this is executed in SIMULATOR phase. More then one primitive in SIMULATOR phase aborts simulation.

Using Ports for Defining Relationship Connections between ports can be used to define relationships between primitives. An application has to be connected to its sending agent. This relationship between application and agent primitive is modeled via a connection from the application’s Agent port to the agent’s TrafficSources port. Because applications are scheduled after agents, it is safe to connect applications to their agents immediately after creating them. To follow the connection to the agent, the port’s method far()→parent()→fullName() should be used to get the full qualified name of the connected agent. This name corresponds to the agent’s name in NS2 namespace.

Source code 25.7 shows a possible source code for the CBR application. It is scheduled for APPLICATION phase (line 3). During begin, first the application itself is created (line 8), some parameters are set (lines 11 and 12), its default start and stop time is set (lines 14-20) and the application is attached to the agent connected at the Agent port in lines 23 and 24. setAlias() is an inherited method of NamedObject - this is described in the next section together with the primitive’s class hierarchy.

14The current code is different, because CBR inherits from Application. The shown code is meant as an example
This diagram does not show all NS2 Primitives. Rather it should give an idea about the base class hierarchy.

Figure 25.10: The Primitive Class Hierarchy
25.3 Working with the MLD designer NS2 Domain

**Primitive Class Hierarchy** Figure 25.10 shows an extract out of the class hierarchy of NS2 primitives. This hierarchy has been established to avoid code duplication and to put often used parameters, ports or methods at one distinct location. Following are descriptions about the individual classes:

**NamedObject** The abstract\(^\text{15}\) primitive NamedObject provides a Parameter called `NS2Name` together with a method called `setAlias()`. If `NS2Name` is specified, `setAlias()` sets this name as alias for the NS2 object that was created as `fullName()` of the primitive. After this, the alias name can be used to refer to the NS2 object - aliases are often used for defining the `DestinationAgent` parameter in agent primitives. Refer to the Hello World Example in sec. 25.3.2.2. Another parameter supplied by the NamedObject primitive is called `NS2ParameterSettings`. This parameter can be used to set a number of parameters of the NS2 object after it has been created. Therefore, the parameter names and their values have to be specified in a list separated by whitespaces. If a value contains whitespaces, it must be double-quoted. The setting of the parameters is done via the method `setParameters()`. As a third parameter, the NamedObject primitive contains `NS2FunctionCalls`. This parameter specifies additional functions that should be called on the NS2 object after it has been generated. Therefore, the parameter should contain a list of function calls. If they contain whitespaces (this is the case as soon as the function has an argument), each function call must be double-quoted. The execution of the function calls is done via the method `callFunction()`. As the last parameter, `NS2ConstructorArgs` contains a list of arguments that are passed to the function creating the NS2 object. This is used to specify additional parameters such as specification of the address of a node.

**ImplementedObject** The abstract primitive ImplementedObject provides all necessary functionality to compile and load additional code into the NS2 process. It has a parameter called `NS2Implementation` that contains a list of source files that should be compiled and loaded into the NS2 process. If the implementation is already precompiled, it is possible to specify a list of object files in the parameter `NS2ObjectFiles`. It is possible to use only one or both parameters to specify the implementation. For compilation and linking of the implementation files, the compile and link options of the NS2 installation are used. To specify additional options, the parameters `NS2CompileOptions` and `NS2LinkOptions` can be used. To be able to load the created shared object into the NS2 process, it must contain an init function that is called by the NS2 after the library was loaded. If this init function is not contained in the implementation, the parameter `NS2UseEmptyInitFunction` must be set to `TRUE`. If this function is contained inside the implementation, it must be defined as follows:

```c
extern "C" int Library_Init(Tcl_Interp *interp)
```

All these actions described within the description of the ImplementedObject primitive are done by the function `compileAndLoadToNS()`.

**BaseAgent** This abstract primitive provides the basic interface of an Agent: `TrafficSources` port and `Node` port. In addition, the method `attachToNode()` is defined. It connects the agent to its related node. Source code 25.8 shows the source code for that method.

---

\(^\text{15}\)There is no explicit code to make these primitives abstract. Here, abstract should illustrate that the primitive should not be instantiated.
inline method
{
    name { attachToNode }
    access { protected }
    type { void }
    code
    {
        NS2Node* node = !Node::far() ? 0 :
        dynamic_cast<NS2Node*>(Node::far()->parent());
        if (node)
            EVAL("$NS2_simulator_attach-agent$\{" << Node::far()->parent()->fullName()
                << "}.$(" << fullName() << ")[;]
        }
    }
}

Source Code 25.8: NS2BaseAgent::attachToNode()

go
{
    std::cout << (Label::null() ? (const char*)(fullName())
        : (const char*)Label )
        << ":\omega\at\omega\leq\ TNow() \leq\" \omega "' << Input::get().print() << std::endl;
}

Source Code 25.9: go method of Dump primitive

BaseApplication This is an abstract primitive, too. It defines an Agent port and the StartTime and StopTime parameters. The methods scheduleStartAndStopTime() and connectToAgent() are defined to perform the corresponding actions.

Node The node primitive defines ports for links and agents to be connected to the node.

Script, Eval, . . . Beside the primitives derived from NamedObject, there are many primitives that don't have any primitive parent - Script and Eval are examples of such primitives.

CBR, UDP, . . . Primitives without any further members or methods listed in fig. 25.10 are given examplary for implemented primitives inside the NS2 domain. Providing a detailed description of all of them is not the intention of this document.

25.3.3.5 Simulation Run

After all requested begin methods have been called, the simulation starts. Particles and events inside the NS2 simulator are processed. Hence the NS2 domain is a specific discrete event domain, there is a continuous time. Events occur at a concrete time. The following section gives an overview about time, event and data handling inside the NS2 domain during the simulation phase.

Accessing Current Simulation Time Current time can be accessed via the primitive member function TNow(). This function is used for example in Dump to print the time when the input integer arrives. Source code 25.9 shows its go method, a description how ports are handle follows in the next paragraphs.
25.3 Working with the MLDesigner NS2 Domain

```c
// Source Code 25.10: go method of CBRTriggerOnOff primitive

go
{
    if (On.dataNew())
    {
        EVAL("$\{ \langle \text{fullName}() \rangle \} \_\text{start}\)
        On.clearDataNew();
    }
    if (Off.dataNew())
    {
        EVAL("$\{ \langle \text{fullName}() \rangle \} \_\text{stop}\)
        Off.clearDataNew();
    }
}
```

Receiving Data through Ports  As in the DE domain, a primitive’s `go` method is called whenever a particle is received at one of its input ports. Additionally the `dataNew` flag of the receiving port is set to true. This flag can be accessed read-only via port’s `dataNew()` member function. The flag is reset to false after port’s `get()` method or `clearDataNew()` is called. Source code 25.10 shows the flag handling in `CBRTriggerOnOff`. Via `get()` the data that has arrived at the port can be accessed read-only. Return type of `get()` is a particle reference that can be handled like inside other MLDesigner domains. Source code 25.9 shows how the data of `Input` is accessed in `Dump`.

Sending Data through Ports  Data can be sent through OutPorts using the stream operator: `outport << 10` sends the integer value 10 through the port. An arbitrary amount of particles can be sent through output ports at simulation run phase. All particles are buffered and will arrive one after another at the receiving port. The time when particles are received corresponds to the sending time of them.

The put method can be used to add a delay for particle sending: `outport.put(TNow() + 1.0, 10)` sends an integer particle 10 that will arrive 1.0 time unit later at the receiving port.

Note, that in contrast to the DE domain, each particle causes a call of `go`. That implies that particles arriving at the same point in simulated time on different ports of a primitive will trigger the `go` method once for each arriving particle.

Sending and Receiving data by MLDesigner NS2 primitives is symbolized by arrow 5, 3 and 4 in fig. 25.9 at page 25-14.

Controlling the NS2 Simulator  Whenever `go` is called, it is possible to send commands to the NS2 simulator via the EVAL macro (refer to source code 25.10). So, it’s even possible to send data to the simulator. This is symbolized by arrow 1 in fig. 25.9 at page 25-14.

Receiving Data from NS2 Simulator  The last important step for using the NS2 simulator in self-built NS2 primitives is to receive data from the NS2 simulator (arrow 2 in fig. 25.9 on page 25-14). Two different procedures are provided by the MLDesigner - NS2 framework: One procedure for pre-defined primitive data types like integer, double or strings and the other for user-defined complex data structures.
The basic architecture for sending small data from the NS2 simulator to MLDesigner is shown in fig. 25.11. It is possible to send integers, doubles and strings; in future releases, more primitive data types might be supported. There are two principle ways to send data to MLDesigner: using the TCL interface or using C++ code. The following paragraphs describe in detail how this is done.

**Simple Data Transfer using TCL**  
Inside the Tcl namespace of NS2 there are three new classes available: MLDIntPort, MLDFloatPort and MLDStringPort. After correct instantiation the corresponding port inside MLDesigner emits an Integer/Float or String whenever a data structure is sent through these Tcl classes. During instantiation, the constructor of MLDXXXPort expects the address of an MLDesigner port as second parameter. Source code 25.11 shows how an MLDIntPort is instantiated and stored under the name $fullname(). The port is linked to the primitive’s integer output port “Out”.

```
begin
{
  EVAL("set " << fullName () << ",\[ new,MLDIntPort," << \( void *,Out " << "]"));
  setAlias ();
  // ...
}
```

Source Code 25.11: instanciation of MLDIntPort (is linked to ’Out’)

Once this is done, the MLDIntPort can be accessed inside the whole Tcl namespace of NS2. So the port can be used in receiving routines of Agents to transfer simple data to MLDesigner. As an example how to use the MLDIntPort instance, an Eval primitive executes the command $NS2_simulator at 5.0 "$IntPort send 42". As result, 42 is emitted through the output port of the MLDesigner primitive and dumped to the console via the Dump primitive.

In the example scenario Port data transfer using Tcl shown in fig. 25.12, all three types (MLDIntPort, MLDFloatPort and MLDStringPort) are used to illustrate how this mechanism works. A simulation run generates the output shown below.

```
Simulation 924 Iteration 1 started Thu Jul 15 13:30:07 2004 on host 'everett'
Integer Received: at 5: 42
String Received: at 6: Hello World
Float Received: at 7: 23.23
```
25.3 Working with the MLDesigner NS2 Domain

Figure 25.12: System: Port data transfer using Tcl

**Simple Data Transfer using C++** The OTcl classes MLDIntPort, MLDFloatPort and MLDStringPort are shadow classes for the template C++ class MLDPort (declaration is shown in source code 25.12). Like the corresponding Tcl classes, MLDPort must be constructed with a string parameter that represents the address of the MLDesigner port. Then the method `send` can be used to send data.

System **Port data transfer using C++** that uses the primitive `IntEmitter`, shows how the mechanism can be used in C++. `IntEmitter` is an `ImplementedObject`. It defines the new OTcl class `IntEmitter` inside `NS2Implementation`. Source code 25.13 shows the implementation. In lines 3-20 the `IntEmitter` class is defined: line 19 declares `port` as `MLDPort<int>`, in line 7 the constructor of `port` is provided with the string parameter for binding to MLDesigner and in line 13 the data is sent to MLDesigner. Lines 23-25 defines the `TclClass` for binding `IntEmitter` into the OTcl namespace.

**Building a new Agent Class** The next paragraph describes how to create a new Tcp Sink Agent primitive. This primitive should emit the size of the received data to MLDesigner whenever a packet arrives. To do this the following steps were done:

**Creating a new NS2 Sink Agent** Because standard NS2 classes do not know anything about MLDesigner, an NS2 class that should be able to communicate with MLDesigner has to be enriched by specific functionality. As the new TcpSink should be able to send its received data through MLDesigner’s NS2 primitive’s ports, a new class MLDTcpSink derived from TcpSink has been created to implement the desired behavior.
```cpp
template <typename Type>
class MLDPort : public TclObject
{
  public:
    MLDPort(const std::string &str);
    MLDPort(const char *const &str);

    void send(const Type &i) const;

    inline const void *const getMLD_EmitterPort() const { return MLD_EmitterPort; }

  protected:
    MLDPort<Type> &operator=(const std::string &portaddr);
    void *MLD_EmitterPort;

    // Tcl command Linkage
    int command(int argc, const char *const *argv);
};
```

Source Code 25.12: Class declaration for MLDPort

```cpp
#include "MLDesigner.h"
#include "MLDPort.h"
#include "config.h"

class IntEmitter : public TclObject {
  public:
    IntEmitter(const char * const port): port(port) {}
    int command(int argc, const char * const *argv) {
      if (argc == 2) {
        if (strcmp(argv[1], "send") == 0) {
          port.send(42);
        }
      }
      return TCL_OK;
    }

  protected:
    MLDPort<int> port;
};

static class IntEmitterClass : public TclClass {
  public:
    IntEmitterClass() : TclClass("IntEmitter") {}
    TclObject *create(int argc, const char * const *argv) {
      if (argc != 5) {
        MLDesigner::error("You should specify the Address of the MLD_Port as a Parameter");
        return NULL;
      }
      else
        return new IntEmitter(argv[4]);
    }
};
```

Source Code 25.13: Source Code for the new OTcl class IntEmitter
25.3 Working with the MLDesigner NS2 Domain

```cpp
class MLDTcpSink : public TcpSink {
public:
    MLDTcpSink(Acker* a, const char* const port) : TcpSink(a), port(port) {}
    void recv(Packet* pkt, Handler* h) {
        port.send(hdr_cmn::access(pkt)->size());
        TcpSink::recv(pkt, h);
    }
protected:
    MLDPort<int> port;
};
```

Source Code 25.14: Class declaration for MLDTcpSink ($NS2Addon/MLDSink.h)

Source code 25.14 shows the declaration of this new class inside NS2 (the code is located in $MLD/MLD_Experimentals/NS2/kernel/NS2Addons/MLDTcpSink.h). The MLDTcpSink class contains a `MLDPort<int>` member variable called `port` (line 11). As we want to send integers, the int specialization of the MLDPort template is used. Via this variable the communication to MLDesigner ports is done. For its initialization a string is necessary. For that reason, the constructor of MLDTcpSink needs an additional `const char* const` parameter that is passed to the `port`’s constructor (in line 3).

The `recv(...)` method of TcpSink is overwritten to perform the actual sending to MLDesigner whenever a packet arrives (line 4-8 in source code 25.14). First, the size of the received packet is sent to MLDesigner. Then, `recv(...)` of the base class is called. The types for the arbitrary packet headers are declared in `$NS2/common/packet.h`. A more detailed description about NS2 packet headers and the access mechanisms for those headers can be found in [NS2a].

Next step at NS2 side is to declare the created MLDTcpSink class in NS2’ s OTcl namespace. Source code 25.15 shows, how this is done. A new class called `MLDTcpSinkClass` has been derived from `TclClass`. The name in OTcl is set to “Agent/TCPSink/MLD” following OTcl’s naming convention for class derivation (line 3). In lines 4 - 12, the creation of a new `MLDTcpSink` instance is specified. The OTcl syntax for creating a new instance ’a’ is “set a [new Agent/TCPSink/MLD 0x12345678]”. 0x12345678 stands examplary for any address specified in that way. Line 4 in the source code tests if the Tcl creation command was invoked with this address parameter. If so, the new `MLDTcpSink` instance is created with this parameter passed to the constructor of `MLDTcpSink` (line 11), otherwise the `FAIL` macro throws an exception in the MLDesigner primitive that has called the malformed command (line 7).

A more detailed description on how to create new agents is given in [NS2a].

**Writing the MLDesigner Primitive** The easier part of the data communication is to write the MLDesigner primitive. Because there is some new code that has to be added to the NS2 process, the new `MLDTCPSink` primitive should be derived from `ImplementedObject`. The `NS2Implementation` parameter should be set to the source code file(s) containing our NS2 code.

---

16 In the following paper, $NS2Addons refers to $MLD/MLD_Experimentals/NS2/kernel/NS2Addons
17 The string contains the address of the MLDesigner’s port. Details are following.
```java
static class MLDTcpSinkClass : public TclClass {
    public:
        MLDTcpSinkClass() : TclClass("Agent/TCPSink/MLD") {}
        TclObject *create(int argc, const char *const *argv) {
            if (argc != 5) {
                FAIL("You should specify the Address of the MLD Port as Parameter");
                return NULL;
            } else {
                return new MLDTcpSink(new Acker, argv[4]);
            }
    }
} class_mld_tcp_sink;
```

Source Code 25.15: OTcl linkage for MLDTcpSink ($NS2Addons/MLDSink.cc)

```java
begin {
    switch (getBeginPhase()) {
        case NS2Star::SIMULATORCONFIG:
            if (compileAndLoadToNS())
                EVAL("Agent/TCPSink/MLD instproc init args {eval $self next $args}"");
            break;
        case NS2Star::AGENT :
            EVAL("set " << fullName() << "[new Agent/TCPSink/MLD]" << (void *)&Size << "[" << NS2ConstructorArgs << "]\n\nsetAlias();
setParameters();
callFunctions();
attachToNode();
break;
    }
}
```

Source Code 25.16: begin of MLDTcpSink

Furthermore, it is necessary to advise the primitive to compile and load the additional sources to the NS2 process during begin phase. This is done via `compileAndLoadToNS()`. If this function returns false, all sources have already been loaded by other instances of this primitive. If the function returns true some initialization can be done.

Here, the code "Agent/TCPSink/MLD instproc init args { eval $self next $args }" is executed to set up the OTcl constructor for "Agent/TCPSink/MLD" after `compileAndLoadToNS()` returns true (refer to source code 25.16 lines 6 and 7).

The created MLDTcpSink can now be used like any other Agent in NS2. The only difference is, that it needs the address of the port where it should send its integer particles through as parameter during construction. Source code 25.16 shows the `begin` code of the MLD TCP Sink primitive. In line 10 and 11, the MLDTcpSink instance is created with the address of the Size port as the argument.

Now MLDTCPSink’s Size port will emit an integer each time the MLDTcpSink receives a network packet. Since the size of the packet is sent automatically through the linked port, MLDTCPSink’s go method is not called when the data is put through the port - of course go of the particle receiving primitive is called as usual.

25-28 MLDesigner Version 2.8
25.3 Working with the MLDesigner NS2 Domain

Complex Data Transfer (MLDataTransfer)  In the previous section it was shown how simple pre-defined data can be transferred from NS2 to MLDesigner. The next paragraphs show how user-defined data structures can be transferred from NS2 to MLDesigner. To understand the procedure, it is necessary to know all concepts described in the last section. Figure 25.13 shows the basic architecture for transferring user data to MLDesigner.

![Diagram of Architecture: sending user-defined data to MLDesigner]

The procedure is introduced using MLDSink as an example. This primitive emits several data particles whenever a packet arrives.

Defining Transfer Data Structure  First the data structure that should be transferred has to be defined. It is possible to transfer any data structure that can be copied by a flat copy. Valid structures are structs that contain only primitive data types, structs or unions of primitive types or structs or unions of other valid structs. Valid structures must not have any virtual methods, virtual base classes or base classes containing virtual methods. The definition of this structure has to be included into the MLDesigner’s primitive and into the NS2 class header or cc file. Source code 25.17 shows how the data structure, that is transferred to MLD Sink is defined.

NS2 Simulator Site  Like for the simple data transfer, a new NS2 class must be defined to talk to MLDesigner. Here, we derive MLDSinkAgent directly from Agent. Like in the simple data transfer introduced above, constructor and recv() method are overwritten (lines 3 and 4 in source code 25.18). Instead of an instance of MLDPort<T>, a member typed MLDDataTransfer is needed (line 6). Further, a member containing the data structure that is supposed to be sent is necessary (line 7).

---

18All structures that can safely be copied via memcpy(Struct *, sizeof(Struct)) can be used.
struct CmnPacketHeaderPart
{
    int size;       // simulated packet size
    int uid;        // unique id
    int error;      // error flag
    int errbitcnt;  // # of corrupted bits jahn
    int fecsize;    //
    double ts;      // timestamp: for q−delay measurement
};

Source Code 25.17: Transfer struct ($NS2Addon/MLDSinkAgent-TransferStruct.h)

class MLDSinkAgent : public Agent {
public:
    MLDSinkAgent(const char *str);
    void recv(Packet*, Handler*);
protected:
    MLDDataTransfer dataToMLD;
    CmnPacketHeaderPart headerPart;
};

Source Code 25.18: Declaration of MLDSinkAgent ($NS2Addon/MLDSinkAgent.h)

During initialization of the MLDSinkAgent, the MLDDataTransfer instance has to be initialized by a string parameter that is provided by the MLDesigner primitive. This is done analogous to the procedure described above: MLDSinkAgent’s constructor passes the string to MLDDataTransfer’s constructor. The create() method in the OTcl class uses argv[4] as parameter to construct a new MLDSinkAgent instance. In addition to a simple data transfer, a MLDDataTransfer instance should be “linked” to the member variable that should be transferred. This is done using the linkToData(const DataRef & ) member function of MLDDataTransfer. The DataRef parameter should be created temporary out of the address and size of the data that should be transferred. Source code 25.19 shows how all this is done inside MLDSinkAgent’s constructor. Source code 25.15 shows how the Tcl linkage should be done.

The recv( . . . ) function fills the headerPart data structure with data out of the received packet’s header and calls the send( ) method of the MLDDataTransfer class. Then, the packet structure is released because a receiving NS2 Agent has to do this. Source code 25.20 shows the implementation of MLDSinkAgent::recv(. . .).

MLDesigner’s MLDSink primitive Inside the MLDesigner primitive, a member variable as instance of the transfer structure is needed ( ph ). This instance will be updated automatically by the NS2 MLDDataTransfer class whenever a send( ) was called. In contrast to the simple data transport through ports, the primitive’s go method is called after data has been

MLDesigner Version 2.8
25.3 Working with the MLDdesigner NS2 Domain

void MLDSinkAgent::recv(Packet* pkt, Handler*)
{
    headerPart.size = hdr_cmn::access(pkt)->size();
    headerPart.uid = hdr_cmn::access(pkt)->uid();
    headerPart.error = hdr_cmn::access(pkt)->error();
    headerPart.errbitcnt = hdr_cmn::access(pkt)->errbitcnt();
    headerPart.fecsize = hdr_cmn::access(pkt)->fecsize();
    headerPart.ts = hdr_cmn::access(pkt)->timestamp();

    dataToMLD.send();
    Packet::free(pkt);
}

Source Code 25.20: MLDSinkAgent::recv(...) ($NS2Addon/MLDSinkAgent.cc)

Source Code 25.21: MLD Sink’s go method

received. Inside go the received data can be processed or passed through output ports. Source code 25.21 shows the go method of MLD Sink.

For initializing the MLDSinkAgent a string must be sent to the NS2 process that is created using a temporary instance of NS2::DataSink. For constructing such a temporary NS2::DataSink instance variable, a reference of the current primitive (this), the address of the receiving structure (&ph) and the size of this structure (sizeof(ph)) have to be provided as parameters. Source code 25.22 shows the begin method of MLDSink. The parameter for constructing the Sink is colored red.

Error handling inside the NS2 process You can send information, warnings or critical errors to MLDdesigner using TCL or C++ commands. In Tcl, the MLDdesigner implements the methods message, warn and error. Because $mldesigner is the singleton instance of this class, messages can be sent via $mldesigner message "Hello World". GUI warnings, messages and errors shown in fig. 25.14 generates the messages inside the Log window of MLDdesigner shown in fig. 25.15. These display features can be used in the C++ context in NS2, too. Here it is necessary to include <MLDesign.h>. The C++ class MLDdesigner implements message, warn and error as static member functions. So, everywhere inside C++ code, MLDdesigner::error("Abort due to a critical error"); can be called to sent an error to MLDdesigner and abort simulation.

19Sent errors abort the current simulation
begin
{
switch (getBeginPhase ())
{
    case NS2Star::SIMULATORCONFIG:
        if (compileAndLoadToNS ())
            EVAL("Agent/MLDSink_insproc_init_args\{_eval_$self_next$_args\}\); break;
    case NS2Star::AGENT:
        EVAL("set\_\" << full\_Name ()
             \"\" << "\{new\ Agent/MLDSink\}" << \
             NS2::Data\_Sink (this, &p\_h, sizeof(p\_h))
             \"\" << NS2\_Constructor\_Args << \
             set\_Alias ();
        set\_Parameters ();
        call\_Functions ();
        attach\_To\_Node ();
        break;
}
}

Source Code 25.22: MLD Sink’s begin method

This system demonstrates how messages, warnings and errors inside the ns2 simulator process can be sent to MLDesigner. If the system is executed using the MLDesigner GUI these messages, warnings and errors are displayed inside the command window in the 'Log' tab.

Figure 25.14: System: GUI warnings, messages and errors

Figure 25.15: The commands window 'Log' tab window after "GUI warnings, ..." was run
Chapter 26
Unsupported Domains

MLDesigner supports a number of domains. Some of them are well developed, while others are still experimental. The supported domains are documented in ch. 23. This chapter discusses only the experimental (unsupported) domains. The documentation and MLDesigner libraries are directly taken from Ptolemy and have not been tested.

For details about the CTDE domain which is experimental but supported see ch. 23

26.1 SR Domain

26.1.1 Introduction
The Synchronous Reactive domain is a statically-scheduled simulation domain in MLDesigner designed for concurrent, control-dominated systems. To allow precise control over timing, it adopts the synchronous model of time, which is logically equivalent to assuming that computation is instantaneous.

26.1.2 SR concepts
Time in the SR domain is a sequence of instants. In each instant, the system observes its inputs and computes its reaction to them. Each instant is assumed to take no time at all. All computation is treated as being instantaneous.

Communication in the SR domain takes place through unbuffered single driver, multiple receiver channels. In each instant, each channel may have a single event with a value, have no event, or be undefined, corresponding to the case where the system could not decide whether the channel had an event or not. Communication is instantaneous, meaning that if an event is emitted on a channel in a certain instant, every primitive connected to the channel will see the event in the same instant.

26.1.3 SR compared to other domains
SR is similar to existing MLDesigner domains, but differs from them in important ways. Like Synchronous Data Flow (SDF), it is statically scheduled and deterministic, but it does not have
buffered communication or multi-rate behavior. SR is better for control-dominated systems that need control over when things happen relative to each other; SDF is better for data-dominated systems, especially those with multi-rate behavior.

SR also resembles the Discrete Event (DE) domain. Like DE, its communication channels transmit events, but unlike DE, it is deterministic, statically scheduled, and allows zero-delay feedback loops. DE is better for modeling the behavior of systems (i.e., to better understand their behavior), whereas SR is better for specifying a system’s behavior (i.e., as a way to actually build it).

### 26.1.4 The semantics of SR

An SR primitive must be well-behaved in the following mathematical sense to make SR systems deterministic. It must compute a monotonic function of its inputs, meaning that when it is presented with more-defined inputs, it must produce more-defined outputs. In particular, an output may only switch from undefined to either present or absent when one or more inputs do, but it may not change its value or become undefined.

The semantics of SR are defined as the least fixed point of the system, meaning the least-defined set of values on the communication channels that is consistent with all the primitives’ functions. That is, if any primitive were evaluated, it will not want to change its output — the value is already correct. The monotonicity constraint on the primitives ensures that there is always exactly one least-defined set, and this is what the SR schedulers calculate.

There are two schedulers for the SR domain, default-SR and dynamic-SR. The dynamic scheduler is the easiest to understand. In each instant, it first initializes all the communication channels to ”undefined” and then executes all the primitives in the system until none of them try to change their outputs. The default scheduler is more shrewd. It uses the communication structure of the system to determine an execution order for the primitives that will make them converge. This is based on a topological sort of the primitives, but is made more complicated when there are feedback loops.

### 26.1.5 Overview of SR primitives

#### 26.1.5.1 General primitives

- **Const**
  - Output a constantly-present integer output given by the `level` parameter.

- **Pre**
  - Emit the value of the integer input from the most recent instant in which it was present.

- **And**
  - Emits the logical AND of the two integer inputs, or absent if both inputs are absent.

- **Add**
  - Emit the sum of the two integer inputs, or the value of the present input if the other is absent.

- **Printer**
  - Print the value of each input to the file specified by the `fileName` parameter. All inputs are printed on a single line with the prefix in the
26.1 SR Domain

prefix parameter.

IntToString Convert the integer input to a string.

StringToInt Convert the string input to an integer.

Sub Emits the different of the two integer inputs, or absent if both inputs are absent.

When When the clock is true, emit the input on the output, otherwise, leave the clock absent.

Mux Depending on the value of the select input, copy either the true input or the false input to the output.

10.5.2 Itcl primitives

By default, all of these primitives have no behavior. They provide an interface for user-written Itcl scripts that specify their behavior. Each has the following states: itclClassName, the class name of the itcl object associated with the primitive, itclSourceFile, the path name of the itcl script containing the definition of this class, and itclObjectName, the name of the itcl object to create. If this field is left blank, the object is given the name of the primitive instance.

ItclOut Single output Itcl primitive.

ItclIn Single input Itcl primitive.

ItclInOut One input, one output Itcl primitive.

ItclCounter Itcl incrementer/decrementer.

ItclModeSelect Itcl primitive used in the Rolodex demo.

ItclDatabase Simple sorted database.

ItclDisplay Display for the rolodex.

ItclEditor Editor for the rolodex

10.5.3 MIDI primitives

The MIDI primitives below are used in the MIDISynthesizer demo described later in this chapter. We include these primitives only as an example of what can be done with the Synchronous Reactive domain.

SerialIn Emit the character waiting on the serial port, or leave the output absent if there isn’t one. The deviceName parameter specifies the port, and baudRate specifies the speed.

MIDIin An interpreter for the MIDI protocol. It takes an incoming MIDI stream and fans it out to Note On and Note Off commands.

SynthControl A polyphonic synthesizer control.

EnvelopeGen An envelope generator for FM sound synthesis.

10.6 An overview of SR demos There are currently three SR demos.

NOTE: Since the SR domain is not supported, the demos mentioned here are not directly ac-
supported through demo palettes, although they are included in the MLDesigner package suite. To access the demos, use the file view tree and look in $MLD/MLD_Experimentals/SR/demo. Please remember that the demos are not tested and may be unable to run properly!

ramp
Prints a sequence of increasing integers. Essentially a "hello world" for the SR domain, this demonstrates how Pre can interact with an adder.

rolodex
A digital address book implemented in SR. This demonstrates how itcl primitives can be used to prototype user-interface-dominated systems at a high level. The system is divided into keyboard, database, editor, and display blocks.

MIDIsynthesizer
A music synthesizer written in SR. This is a polyphonic sound synthesizer written using custom SR primitives for the control portion. Waveform synthesis is done using an FM algorithm implemented in CGC. This requires a MIDI keyboard to be attached to /dev/ttya, and functioning CGC audio drivers for your platform.
Chapter 27

Code Generation Domains - unsupported

27.1 VHDL Domain

27.1.1 Introduction

The VHDL domain generates code in the VHDL (VHSIC Hardware Description Language) programming language. This domain supports the synchronous data flow model of computation. This is in contrast to the VHDLB domain, which supports the general discrete event model of computation of the full VHDL language.

Since the VHDL domain is based on the SDF model, it is independent of any notion of time. The VHDL domain is intended for modeling systems at the functional block level, as in DSP functions for filtering and transforms, or in digital logic functions, independent of implementation issues.

The VHDL domain replaces the VHDLF domain. It is not, however, meant to be used in the same way as the VHDLF domain: the VHDL domain is for generating code from functional block diagrams with SDF semantics, while the VHDLF domain was intended to contrast with the VHDLB domain. It supported structural code generation using VHDL blocks with no execution delay or timing behavior, just functionality. The semantics for the VHDLF domain were not strictly defined, and quite a lot depended on how the underlying VHDL code blocks associated with each VHDLF primitive were written.

Within the VHDL domain, there are a number of different Targets to choose from. The default target, default-VHDL, generates sequential VHDL code in a single process within a single entity, following the execution order from the SDF scheduler. This code is suitable for efficient simulation, since it does not generate events on signals. The SimVSS-VHDL target is derived from default-VHDL, and provides facilities for simulation using the Synopsys VSS VHDL simulator. Communication actors and facilities in the SimVSS-VHDL target support code synthesis and co-simulation of heterogeneous CG systems under the CompileCGSubsystems target developed by Jose Pino. There is also a SimMT-VHDL target for use with the Model Technology VHDL simulator. The struct-VHDL target generates VHDL code in which individual actor firings are encapsulated in separate entities connected by VHDL signals. This target generates code which is intended for circuit synthesis. The SynthVHDL target, derived from struct-VHDL, provides facilities for synthesizing circuit representations from the structural code using the Synopsys De-
sign Analyzer tool set. Each of these targets is discussed in more detail in the next section.

Because the VHDL domain uses SDF semantics, it supports retargeting from other domains with SDF semantics (SDF, CGC, etc.) provided that the primitives in the original graph are available in the VHDL domain. As this experimental domain evolves, more options for VHDL code generation from data flow graphs will be provided. These options will include varying degrees of user control and automation depending on the target and the optimization goals of the code generation, particularly in VHDL circuit synthesis.

**27.1.1.1 Setting Environment Variables**

In order to have the Synopsys simulation target work correctly, you should make sure that the following environment variables and paths are set correctly. The `SYNOPSYS` and `SIM_ARCH` shell environment variables are settable within the Synopsys simulation target, `SimVSS-VHDL`.

Also, you may need to permanently add the following lines to your `.cshrc` file and uncomment the ones you wish to take effect:

```bash
# For VHDL Synopsys demos, uncomment the following:
# setenv SYNOPSYS /usr/tools/synopsys
# setenv SIM_ARCH sparcOS5
# You need the last one of these (.../sge/bin) to run vhdl.dbx
# since vhdl.dbx looks for "msgsvr":
# set path = ( $path $SYNOPSYS/$SIM_ARCH/syn/bin
# $SYNOPSYS/$SIM_ARCH/
# sim/bin $SYNOPSYS/$SIM_ARCH/sge/bin)
# You need this to run vhdl.sim, and since vhdl.dbx calls
# vhdl.sim, you need this to run vhdl.dbx also:
# setenv MLD_LIBRARY_PATH ${MLD_LIBRARY_PATH}
# ${SYNOPSYS}/${SIM_ARCH}/sim/lib
## For Motorola S56x card demos on the Sparc, you will
## need something like:
# setenv S56DSP /users/ptdesign/vendors/s56dsp
# setenv QCKMON qckMon5
# setenv MLD_LIBRARY_PATH ${MLD_LIBRARY_PATH}:${S56DSP}/lib
```

You will need to have a `.synopsys_vss.setup` file with the right library directive in it in order to use the communication vhdl modules needed for the `CompileCGSubsystems` target. This file in the root MLDdesigner directory has the correct directive defining the location of the `PTVHDLSIM` library. Synopsys simulation only sees the file if it is in one of three places: the current directory in which simulation is invoked, the configuration directory within the Synopsys installation tree, or the user’s home directory. Since working directories are frequently created and destroyed, and since the Synopsys installation will vary from site to site, the user’s home directory is the best place to put this file, but each user must do this if the root of their personal MLDdesigner tree is anything other than their home directory.

Here is the text in `MLD/./.synopsys_vss.setup`:
27.1 VHDL Domain

-- This is so communication code can be
-- compiled into the PTVHDLSIM library:
PTVHDLSIM: $MLD/obj.$PTARCH/utils/ptvhdlsim

NOTE: If you build your own tree and it includes your own $MLD/src/utils/ptvhdlsim directory, then you will need to modify your .synopsys.vss.setup file to point to this directory prior to building the new tree. During the build process, this file is needed so that the ptvhdlsim executable can be correctly linked. If it is pointing to some other directory, then you may experience problems linking ptvhdlsim.

27.1.2 VHDL Targets

The targets of the VHDL domain generate VHDL code from SDF graphs. The targets differ from one another in the styles of VHDL code which they produce, or in the facilities they provide for passing the generated code to VHDL simulation or circuit synthesis tools. The graphs of VHDL actors in MLDesigner are meant to be retargetable in that one graph can be used with multiple VHDL targets, depending on the circumstances. The available targets in the VHDL domain are: default-VHDL, struct-VHDL, SimVSS-VHDL, SimMT-VHDL, and Synth-VHDL. There is also support for using SimVSS-VHDL as a child target of CompileCGSubsystems for heterogeneous code generation and co-simulation.

All of the VHDL targets share the following parameters, which are inherited from the base class HLLTarget:

- **directory** (STRING) Default = $HOME/MLD_SYSTEMS
  The name of the directory into which generated code files and supporting files are written. In derived targets, this is also the directory in which compilation for simulation and synthesis are performed.

- **Looping Level** (INT) Default = 0
  The control for selecting the looping complexity of the SDF scheduler which is used. Note that looping of code is not supported in the current implementation, except at the main iteration loop on the outside. Therefore a looping level of zero should be used with all loop schedulers or incorrect code may result. In future releases, higher looping levels will be supported.

- **display?** (INT) Default = TRUE
  Option to display generated code files to the screen.

- **write schedule?** (INT) Default = FALSE
  Option to write the schedule to a file. The name of the file will be <module name>_<module name>_sched.

### 27.1.2.1 The default-VHDL Target

The default-VHDL target generates VHDL code in a simple and straightforward style which is designed to preserve the SDF scheduling order while incurring minimum VHDL simulation overhead. The code is generated as a single VHDL entity containing a single process of sequential
states. The sequential process reflects the order of execution determined by the SDF scheduler. All data values are stored and communicated through internal variables so that the simulation overhead of VHDL signals and the VHDL discrete-event scheduler can be avoided. No actual simulation is performed by the default-VHDL target. It is left to derived targets to support VHDL simulation.

To generate the code, the default-VHDL target first invokes the SDF scheduler, and then goes through the resulting schedule in order, firing each VHDL primitive in sequence. As each VHDL primitive is fired, a block of VHDL sequential statements is generated. Porthole and State references and values are resolved and any necessary VHDL variables are created and placed in the list of declared variables. One VHDL primitive may be fired multiple times and each firing will cause a new codeblock with new variables to be generated. The target manages the communication of data from one VHDL primitive to the next through VHDL variables. The target also manages state propagation from one firing to the next of the same VHDL primitive through VHDL variables. State values and tokens remaining on arcs at the end of the schedule iteration are also fed back through the correct variables so that the process can be looped repeatedly and function identically to the original SDF graph.

27.1.2.2 The struct-VHDL Target

The struct-VHDL target generates VHDL code in a structural style, in which firings of VHDL primitives are individually encapsulated in VHDL entities. The entities are connected to one another through VHDL signals, and the flow of data and state from one firing entity to the next enforces the precedence relationships inherent in the data flow graph and the resulting schedule. The overall structure of the completed code description parallels the precedence directed acyclic graph (DAG).

The procedure used by the struct-VHDL target to generate the code begins similarly to that of the default-VHDL target. First, the SDF scheduler is invoked and a valid schedule is computed. Then the schedule is run, and as each VHDL primitive is fired, the target generates an individual VHDL entity for each firing while keeping track of input and output references to portholes and states. The target manages the references so that it can correctly instantiate each VHDL entity and create VHDL signals to map to the VHDL ports for carrying data and state from one firing to the next. Only firings which have actual dependencies will be connected in the VHDL code representation. In this way, the code generated represents the maximum parallelism in the graph computation outside the granularity level of an individual firing.

The current version of the struct-VHDL target also generates registers for latching the values of states and remaining tokens at the end of an iteration. It feeds back the outputs of these registers to the correct inputs at the beginning of the graph so that the structure can be "clocked" by an input clock signal common to all such registers. This clock, on a positive transition, represents the tick of one completed iteration of the data flow graph. This clock becomes an input to the entire top-level VHDL entity, and will presumably be supplied by an outside source or signal driver during simulation. Similarly, there is an input created for a control signal which selects between the initial values of states or initial tokens and the succeeding values which are passed from one iteration to the next.
27.1.2.3 The SimVSS-VHDL Target

The SimVSS-VHDL target is derived from the default-VHDL target. It generates code in the same single-entity, single-process, sequential style as the default-VHDL target, but it also provides facilities for simulation using the Synopsys VSS VHDL simulator. Depending on the target parameters set when running this target, following the code generation phase this target can compile, elaborate, and execute interactively or non-interactively the design specified by the generated VHDL code.

Communication actors and facilities in the SimVSS-VHDL target support code synthesis and co-simulation of heterogeneous CG systems under the CompileCGSubsystems target developed by Jose Pino. This allows a user to manually partition a graph using hierarchy so that multiple code files of different code generation domains can be generated. They are then executable if run on host machines which provide all the needed simulators and supporting hardware resources that the individual child targets require. The communication between the different code generation subsystems is automatically generated and correct synchronization and deadlock avoidance are guaranteed. This capability is demonstrated with VHDL in a number of demos included through the main VHDL demo palette.

The additional parameters of the SimVSS-VHDL target are as follows:

- **$SYNOPSYS** (STRING) Default = /usr/tools/synopsys
  
  Value of the SYNOPSYS environment variable. It points to the root of the Synopsys tools installation on the host machine.

- **$ARCH** (STRING) Default = sparcOS5

  Value of the ARCH environment variable. It indicates which architecture/operating system the Synopsys tools will be run on.

- **$SIM_ARCH** (STRING) Default = sparcOS5

  Value of the SIM_ARCH environment variable. It indicates which architecture/operating system the Synopsys VSS simulator will be run on.

- **analyze** (INT) Default = TRUE

  If TRUE then attempt to analyze the VHDL code using the gvan tool, checking for syntax errors.

- **startup** (INT) Default = TRUE

  If TRUE then attempt to startup the VHDL simulator (vhdldbx if interactive = TRUE, else ptvhdl2sim).

- **simulate** (INT) Default = TRUE

  Currently unused. If interactive = FALSE, simulation under ptvhdl2sim will begin automatically following startup.

- **report** (INT) Default = TRUE

  Currently unused.

- **interactive** (INT) Default = FALSE

  If TRUE then when simulating, run vhdldbx. Otherwise, run ptvhdl-
### 27.1.2.4 The SimMT-VHDL Target

The SimMT-VHDL target is derived from the default-VHDL target. It generates code in the same single-entity, single-process, sequential style as the default-VHDL target, and also provides facilities for simulation using the Model Technology VHDL simulator. Depending on the target parameters set when running this target, following the code generation phase this target can compile, elaborate, and execute interactively or non-interactively the design specified by the generated VHDL code.

The additional parameters of the SimMT-VHDL target are as follows:

- **analyze** (INT) Default = TRUE
  If TRUE then attempt to analyze the VHDL code using the vcom tool, checking for syntax errors.

- **startup** (INT) Default = TRUE
  If TRUE then attempt to startup the vsim VHDL simulator

- **simulate** (INT) Default = TRUE
  Currently unused. If startup = TRUE and interactive = FALSE, simulation under vsim will begin automatically following startup. If startup = TRUE and interactive = TRUE, vsim will startup but wait for user input.

- **report** (INT) Default = TRUE
  Currently unused.

- **interactive** (INT) Default = FALSE
  If TRUE, then when simulating, start up vsim and wait for user input.
  If FALSE, then when simulating, run vsim in the background.

### 27.1.2.5 The Synth-VHDL Target

The Synth-VHDL target is derived from the struct-VHDL target. It generates code in the same structural style as the struct-VHDL target, but it also provides facilities for synthesis and optimization using the Synopsys Design Analyzer tool set.

Not every design which can be specified as an SDF graph using the VHDL primitives available in the main primitive palettes will be synthesizable. Some primitives generate code which is not synthesizable under the rules required by the Synopsys Design Analyzer.

There is conceptually more than one way to generate synthesizable VHDL for a given data flow graph. Just as the sequential VHDL of the default-VHDL target differs from the structural VHDL of the struct-VHDL target, so there are also multiple ways in which the structural VHDL could be generated. The struct-VHDL target as is only generates one particular style. A programmer with some experience could modify this target or create a new or derived target to generate the code in a different structural style to suit different needs.
The additional parameters of the Synth-VHDL target are as follows:

- **analyze** (INT) Default = TRUE
  If TRUE then attempt to analyze the VHDL code using the design_analyzer tool, checking for syntax errors.

- **elaborate** (INT) Default = TRUE
  If TRUE then attempt to elaborate the analyzed design into a netlist form.

- **compile** (INT) Default = TRUE
  If TRUE then attempt to compile the elaborated design into an optimized netlist.

- **report** (INT) Default = TRUE
  If TRUE then generate reports on the compile-optimized designs for area and timing.

### 27.1.3 An Overview of VHDL Primitives

This library is divided into the following categories:

- Arithmetic
- Control
- DSP - signal processing.
- Demo
- Nonlinear
- Sinks
- Sources

Most of the primitives in the VHDL domain have equivalent counterparts in the SDF domain.

#### 27.1.3.1 Source Primitives

Source primitives have no inputs and produce data on their outputs. All of these primitives are equivalent to the SDF primitives of the same name.

#### 27.1.3.2 Sink Primitives

Sink primitives have no outputs and consume data on their inputs. All of these primitives are equivalent to the SDF primitives of the same name.

#### 27.1.3.3 Arithmetic Primitives

Arithmetic primitives perform simple functions such as addition and multiplication. All of the primitives are equivalent to the SDF primitives of the same name.
27.1.3.4 Nonlinear Primitives

Nonlinear primitives perform simple functions. All of these are equivalent to the SDF primitives of the same name.

27.1.3.5 Control Primitives

Control primitives are used for routing data and other control functions. All of these are equivalent to the SDF primitives of the same name.

27.1.3.6 DSP (Digital Signal Processing)

All of the primitives are equivalent to the SDF primitives of the same name.

27.1.4 An Overview of VHDL Demos

NOTE: Since the VHDL domain is not supported, the demos mentioned here are not directly accessible through demo palettes, although they are included in the MLDesigner package suite. To access the demos, use the Library view and look in $MLD/MLD_Experimental/VHDL Domain/Demo. Please remember that the demos are not tested and may not run properly!

The demos are divided into categories: code generation, simulation, synthesis, and co-simulation. Some of the demos in the VHDL domain have equivalent counterparts in the SDF or CGC domains. In this chapter, only brief descriptions are given to explain the using of the VHDL domain.

27.1.4.1 Code Generation Demos

These demos do nothing but to generate code.

The sequential demos use the default-VHDL target. The structural demos use the struct-VHDL target. They are essentially the same systems being run, but with two different targets producing two different styles of VHDL code. These demos provide a direct comparison of these two basic styles of VHDL code generation.

27.1.4.2 Simulation Demos

These demos use the SimVSS-VHDL target. Each one generates VHDL code which is functionally equivalent to the SDF graph specification, and then the code is executed on the Synopsys VSS Simulator. Graphical monitoring blocks provide output analysis of the results of running these systems.

27.1.4.3 Synthesis Demos

These demos use the Synth-VHDL target. Each one generates structural VHDL code which is equivalent to the SDF specification. One difference is that the data types are converted to simple 4-bit integers to speed up the synthesis process. Once the code is generated, the netlist is synthesized...
through the Synopsys Design Analyzer. Following that, the netlist is optimized and then control of the Design Analyzer is returned to the user for further exploration and inspection.

### 27.1.4.4 Co-simulation Demos

These demos use the `CompileCGSubsystems` target which uses the SimVSS-VHDL target as a child target for the VHDL portions of the systems. The first three demos generate stand-alone heterogeneous programs which run in C, Motorola DSP56000 assembly, and VHDL. They produce analysis and synthesis filterbanks for perfect reconstruction using progressively more complex structures. The fourth demo also generates a Tcl/Tk user interface for selecting one of three waveform inputs to the system. The fifth and final demo generates the filterbank system, but instead of doing it as a standalone program, it incorporates the system into a wormhole inside a top-level SDF system. This way the subsystem can be executed in code which is potentially faster than SDF simulation, and it can be reused without having to recompile the subsystem each time the top-level system is executed.

### 27.2 CG Domain

#### 27.2.1 Introduction

The Code Generation (CG) domain and its derivative domains, such as the CG56 domain (Motorola DSP56000) and the C language (CGC) domain, are used to generate code rather than to run simulations. Only the derivative domains are of practical use for generating code. The primitives in the CG domain itself can be thought of as "comment generators"; they are useful for testing and debugging schedulers and for little else. The CG domain is intended as a model and a collection of base classes for derivative domains. This section documents the common features and general structure of all code generation domains.

All the code generation domains that are derived from the CG domain in this release obey SDF semantics and can thus be scheduled at compile time. Internally, however, the CG only assumes that primitives obey data flow semantics. Currently, we have implemented two approaches for data-dependent execution, CGDDF, which recognizes and implements certain commonly used programming constructs, and BDF ("Boolean data flow" or the token-flow model) [Buc93]. Even when these are implemented, the vast majority of primitives in any given application should obey the SDF rules to permit efficient multiprocessor code generation.

A key feature of code generation domains is the notion of a target architecture. Every application must have a user-specified target architecture, selected from a set of targets supported by the user-selected domain. Every target architecture is derived from the base class `Target`, and controls such operations as scheduling, compiling, assembling, and downloading code. Since the target controls scheduling, multiprocessor architectures can be supported with automated task partitioning and synchronization.

Another feature of the code generation domains is the ability to use different schedulers. A key idea in MLDesigner is that there is no single scheduler that is expected to handle all situations.
We have designed a suite of specialized schedulers that can be mixed and matched for specific applications. Some targets in the CG domain, in addition to serving as base classes for derived domains, allow the user to experiment with these various schedulers.

27.2.2 Targets

A code generation Domain is specific to the language generated, such as C (CGC) and DSP56000 assembly code (CG56). Each code generation domain has a default target which defines routines generic to the target language. These targets are derived from targets defined in the CG domain.

A Target object has methods for generating a schedule, compiling the code, and running the code (which may involve downloading code to the target hardware and beginning its execution). There also may be child targets (for representing multiprocessor targets) together with methods for scheduling the communication between them. Targets also have parameters that are user specified. There are four targets in the CG domain; these are described below.

27.2.2.1 default-CG

This target is the default target for the CG domain. It allows the user to experiment with the various uniprocessor schedulers. Currently, there is a suite of schedulers that generate schedules of various forms of optimality. For instance, the default SDF scheduler generates schedules that try to minimize the amount of buffering required on arcs, while the loop schedulers try to minimize the amount of code that is generated. Refer to the schedulers section in this chapter for a discussion on these schedulers. There are only two parameters for this target:

- directory (STRING) Default = $HOME/MLD_SYSTEMS
  This is the directory to which all generated files will be written to.

- looping Level (STRING) Default = ACYLOOP
  The choices are DEF, CLUST, SJS, or ACYLOOP. Case does not matter; ACYLOOP is the same as AcyLoop. If the value is DEF, no attempt will be made to construct a looped schedule. This can result in very large programs for multirate systems, since inline code generation is used, where a codeblock is inserted for each appearance of an actor in the schedule. Setting the level to CLUST invokes a quick and simple loop scheduler that may not always give single appearance schedules. Setting it to SJS invokes the more sophisticated SJS loop scheduler, which can take more time to execute, but is guaranteed to find single appearance schedules whenever they exist. Setting it to ACYLOOP invokes a scheduler that generates single appearance schedules optimized for buffer memory usage, as long as the graph is acyclic. If the graph is not acyclic, and ACYLOOP has been chosen, then the target automatically reverts to the SJS scheduler. For backward compatibility, "0" or "NO", "1", and "2" or "YES" are also recognized, with "0" or "NO" being DEF, "1" being CLUST, and "2" or "YES" being SJS. NOTE: Loop scheduling only applies to uniprocessor targets; hence, this parameter does not appear in the FullyConnected target.
In addition to these parameters, there are a number of parameters that are in this target that are not visible to the user. These parameters may be made visible to the user by derived targets. The complete list of these parameters follows:

- **host** (STRING) Default = 
The default is the empty string. This is the host machine to compile or assemble code on. All code is written to and compiled and run on the computer specified by this parameter. If a remote computer is specified here then rsh commands are used to place files on that computer and to invoke the compiler. You should verify that your `.rhosts` file is properly configured so that rsh will work.

- **file** (STRING) Default = 
The default is the empty string. This represents the prefix for filenames for all generated files.

- **display?** (INT) Default = YES
  If this flag is set to YES, then the generated code will be displayed on the screen.

- **compile?** (INT) Default = YES
  If this flag is set to YES, then the generated code will be compiled (or assembled).

- **load?** (INT) Default = YES
  If this flag is set to YES, then the compiled code will be loaded onto a chip.

- **run?** (INT) Default = YES
  If this flag is set to YES, then the generated code is run.

### 27.2.2.2 bdf-CG

This target demonstrates the use of BDF semantics in code generation. It uses the BDF scheduler to generate code. See the BDF domain documentation for more information on BDF scheduling. There is only one target parameter available to the user; the directory parameter above. This parameter has the same functionality as above.

### 27.2.2.3 FullyConnected

This target models a fully connected multiprocessor architecture. It forms the base class for all multiprocessor targets with the fully connected topology. Its parameters are mostly to do with multiprocessor scheduling.

The parameters for FullyConnected are:

- **nprocs** (INT) Default = 2
  Number of processors in the target architecture.

- **sendTime** (INT) Default = 1
  This is the time required, in processor cycles, to send or receive one datum in the multiprocessor architecture. Sending and receiving are assumed to take the same amount of time.
onePrimitiveOneProc  (INT) Default = NO  
If this is YES, then all invocations of a primitive are scheduled onto 
the same processor.

manualAssignment (INT) Default = NO  
If this is YES, then the processor assignment is done manually by the 
user by setting the procId parameter in each primitive.

adjustSchedule (INT) Default = NO  
If this is YES, then the automatically generated schedule is overridden 
by manual assignment. This feature requires improvements in the user 
interface before it can be implemented; hence, the default is NO.

childType (STRINGARRAY) Default = default-CG  
This parameter specifies the names of the child targets, separated by 
spaces. If the number of strings is fewer than the number of processors 
specified by the nprocs parameter, the remaining processors are of type 
given by the last string. For example, if there are four processors, and 
childType is set to default-CG56[2] default-CGC, then 
the first two child targets will be of type default-CG56, and the 
next two of type default-CGC.

resources (STRINGARRAY) Default = 
The default is the empty string. This parameter defines the specific 
resources that child targets have, separated by ";". For example, if the 
first processor has I/O capabilities, this would be specified as STDIO. 
Then, primitives that request STDIO would be scheduled onto the first 
processor.

relTimeScales (INTARRAY) Default = 1  
This defines the relative time scales of the processors corresponding to 
child targets. This information is needed by the scheduler in order to 
compute scheduling costs. The number of entries here should be the 
same as the number of processors; if not, then the last entry is used 
for the remaining processors. The entries reflect the relative comput-
ing speeds of different processors, and are expressed as relative cycle 
times. For example, if there is a DSP96000 (32Mhz) and a DSP56000 
(20Mhz), the relative cycle times are 1 and 1.6. The default is 1 (mean-
ing that all processors have the same computing speed).

ganttChart (INT) Default = YES  
If this is YES, then the Gantt chart containing the generated schedule 
is displayed.

logFile (STRING) Default = 
This is the name of the file to which a log will be written of the schedul-
ing process. This is useful for debugging schedulers. If no filename is 
specified, no log is generated.

amortizedComm (INT) Default = NO  
If this is YES, the scheduler will try to reduce the communication over-
head by sending multiple samples per send. This has not really been implemented yet.

`schedName(DL,HU,DC,HIER,CGDDF)` (STRING) Default = DL

Using the `schedName` parameter, a user can select which parallel scheduling algorithm to use. There are three basic SDF parallel scheduling algorithms. The first two can be used for heterogeneous processors, while the last can only be used for homogeneous processors.

**HU** selects a scheduling algorithm based on the classical work by T. C. Hu [Hu61]. This scheduler ignores the interprocessor communication cost (IPC) during scheduling and thus may result in unrealistic schedules. The next two scheduling algorithms take into IPC.

**DL** selects Gil Sih’s dynamic level scheduler [SL93a] (default).

**DC** selects Gil Sih’s declustering algorithm [SL93b]. This scheduler only supports homogeneous multiprocessor targets. It is more expensive than the DL and HU schedulers, so should be used only if the DL and HU schedulers produce poor schedules.

**HIER** selects a preliminary version of Jose Luis Pino’s hierarchical scheduler [PBL95]. With this scheduler, the user can specify a top-level parallel scheduler from the three listed above and also specify uniprocessor schedulers for individual galaxies. The default top-level scheduler is DL; to specify another use the following syntax: HIER(HU) or HIER(DC). To specify a uniprocessor scheduler for a module, add a new module string parameter named `Scheduler` and set it to either Cluster (looping level 1), Loop (looping level 2) or SDFScheduler (looping level 0).

**CGDDF1** selects Soonhoi Ha’s dynamic construct scheduler [Ha92]. A dynamic construct, clustered as a primitive instance, can be assigned to multiple processors. In the future, we may want to schedule a primitive exploiting data-parallelism. A primitive instance that can be assigned to multiple processors is called a ”macro” actor. MACRO scheduler is expected to allow the macro actors. For now, however, MACRO scheduler is not implemented.

### 27.2.2.4 SharedBus

This third target, also a multiprocessor target, models a shared-bus architecture. In this case, the scheduler computes the cost of the schedule by imposing the constraint that more than one send or receive cannot occur at the same time (since the communication bus is shared).
27.2.3 Schedulers

Given a System of functional blocks to be scheduled and a Target describing the topology and characteristics of the single- or multiple-processor system for which code is to be generated, it is the responsibility of the Scheduler object to perform some or all of the following functions:

- Determine which processor a given invocation of a given Block is executed on (for multi-processor systems).
- Determine the order in which actors are to be executed on a processor.
- Arrange the execution of actors into standard control structures, like nested loops.

If the program graph follows SDF semantics, all of the above steps are done statically (i.e. at compile time). A data flow graph with dynamic constructs uses the minimal runtime decision making to determine the execution order of actors.

27.2.3.1 Single-Processor Schedulers

For targets consisting of a single processor, we provide three different scheduling techniques. The user can select the most appropriate scheduler for a given application by setting the loopingLevel target parameter.

In the first approach (loopingLevel = DEF), which is the default SDF scheduler, we conceptually construct the acyclic precedence graph (APG) corresponding to the system, and generate a schedule that is consistent with that precedence graph. Note that the precedence graph is not physically constructed. There are many possible schedules for all but the most trivial graphs; the schedule chosen takes resource costs, such as the necessity of flushing registers and the amount of buffering required, into account. The target then generates code by executing the actors in the sequence defined by this schedule. This is a quick and efficient approach when the SDF graph does not have large sample-rate changes. If there are large sample rate changes, the size of the generated code can be huge because the codeblock for an actor might occur many times (if the number of repetitions for the actor is greater than one); in this case, it is better to use some form of loop scheduling.

The second approach we call Joe’s scheduling. In this approach (loopingLevel = CLUST), actors that have the same sample rate are merged (wherever this will not cause deadlock) and loops are introduced to match the sample rates. The result is a hierarchical clustering; within each cluster, the techniques described above can be used to generate a schedule. The code then contains nested loop constructs together with sequences of code from the actors.

Since the second approach is a heuristic solution, there are cases where some looping possibilities go undetected. By setting the loopingLevel to SJS, we can choose the third approach, called SJS (Shuvra-Joe-Soonhoi) scheduling after the inventor’s first names. After performing Joe’s scheduling at the front end, it attacks the remaining graph with an algorithm that is guaranteed to find the maximum amount of looping available in the graph. That is, it generates a single appearance schedule whenever one exists.

A fourth approach, obtained by setting loopingLevel to ACYLOOP, we choose a scheduler
that generates single appearance schedules optimized for buffer memory usage. This scheduler was developed by Praveen Murthy and Shuvra Bhattacharyya [Mur96, BML96]. This scheduler only tackles acyclic SDF graphs, and if it finds that the system is not acyclic, it automatically resets the loopingLevel target parameter to SJS. Basically, for a given SDF graph, there could be many different single appearance schedules. These are all optimally compact in terms of schedule length (or program memory in inline code generation). However, they will, in general, require differing amounts of buffering memory; the difference in the buffer memory requirement of an arbitrary single appearance schedule versus a single appearance schedule optimized for buffer memory usage can be dramatic. In code generation, it is essential that the memory consumption be minimal, especially when generating code for embedded DSP processors since these chips have very limited amounts of on-chip memory. Note that acyclic SDF graphs always have single appearance schedules; hence, this scheduler will always give single appearance schedules. If the file target parameter is set, then a summary of internal scheduling steps will be written to that file. Essentially, two different heuristics are used by the ACYLOOP scheduler, called APGAN and RPMC, and the better one of the two is selected. The generated file will contain the schedule generated by each algorithm, the resulting buffer memory requirement, and a lower bound on the buffer memory requirement (called BMLB) over all possible single appearance schedules.

If the second, third, or fourth approaches are taken, the code size is drastically reduced when there are large sample rate changes in the application. On the other hand, we sacrifice some efficient buffer management schemes. For example, suppose that primitive A produces 5 samples to primitive B which consumes 1 sample at a time. If we take the first approach, we schedule this graph as ABBBBB and assign a buffer of size 5 between primitive A and B. Since each invocation of primitive B knows the exact location in the allocated buffer from which to read its sample, each B invocation can read the sample directly from the buffer. If we choose the second or third approach, the scheduling result will be A5(B). Since the body of primitive B is included inside a loop of factor 5, we have to use indirect addressing for primitive B to read a sample from the buffer. Therefore, we need an additional buffer pointer for primitive B (memory overhead), and one more level of memory access (run-time overhead) for indirect addressing.

### 27.2.3.2 Multiple-Processor Schedulers

The first step in multiprocessor scheduling, or parallel scheduling, is to translate a given SDF graph to an acyclic precedence expanded graph (APEG). The APEG describes the dependency between invocations of blocks in the SDF graph during execution of one iteration. Refer to the SDF domain documentation for the meaning of one iteration. Hence, a block in a multirate SDF graph may correspond to several APEG nodes. Parallel schedulers schedule the APEG nodes onto processors. Unfortunately, the APEG may have a substantially greater (at times exponential) number of nodes compared to the original SDF graph. For this a hierarchical scheduler is being developed that only partially expands the APEG [PL95].

We have implemented three basic scheduling techniques that map SDF graphs onto multiprocessors with various interconnection topologies: Hu’s level-based list scheduling, Sih’s dynamic level scheduling [SL93a], and Sih’s declustering scheduling [SL93b]. The target architecture is described by its Target object. The Target class provides the scheduler with the necessary information on the number of processors, interprocessor communication etc., to enable both
27 Code Generation Domains - unsupported

scheduling and code synthesis.

The hierarchical scheduler can use any one of the three basic parallel schedulers as the top-level scheduler. The current implementation supports user-specified clustering at module boundaries. These galaxies are assumed to compose into valid SDF primitives in which the SDF parameters are derived from the internal schedule of the module. During APEG expansion, these compositions are opaque; thus, the entire module is treated as a single SDF primitive. Using hierarchical scheduling techniques, we have realized multiple orders of magnitude speedup in scheduling time and multiple orders of magnitude reduction of memory usage. See \[PBL95\] for more details.

The previous scheduling algorithms could schedule SDF graphs, the CGDDF scheduler can also handle graphs with dynamic constructs. See ch. 27.2.5 for more details.

Whichever scheduler is used, we schedule communication nodes in the generated code. For example, if we use Hu’s level-based list scheduler, we ignore communication overhead when assigning primitives to processors. Hence, the generated code is likely to contain more communication code than with the other schedulers that do not ignore the IPC overhead.

There are other target parameters that direct the scheduling procedure. If the parameter manualAssignment is set to YES, then the default parallel scheduler does not perform primitive assignment. Instead, it checks the processor assignment of all primitives (set using the procId parameter of CG and derived primitives). By default, the procId parameter is set to -1, which is an illegal assignment since the child target is numbered from 0. If there is any primitive, except the Fork primitive, that has an illegal procId parameter, an error is generated saying that manual scheduling has failed. Otherwise, we invoke a list scheduler that determines the order of execution of blocks on each processor based on the manual assignment. We do not support the case where a block might require more than one processor. The manualAssignment target parameter automatically sets the onePrimitiveOneProc parameter to YES; this is discussed next.

If there are sample rate changes, a primitive in the program graph may be invoked multiple times in each iteration. These invocations may be assigned to multiple processors by default. We can prevent this by setting the onePrimitiveOneProc parameter to YES. Then, all invocations of a primitive are assigned to the same processor, regardless of whether they are parallelizable or not. The advantage of doing this is the simplicity in code generation since we do not need to splice in Spread/Collect primitives, which will be discussed later. Also, it provides us another possible scheduling option, adjustSchedule; this is described below. The main disadvantage of setting onePrimitiveOneProc to YES is the performance loss of not exploiting parallelism. It is most severe if Sih’s declustering algorithm is used. Therefore, Sih’s declustering algorithm is not recommended with this option.

In this paragraph, we describe a future scheduling option that this release does not support yet. Once automatic scheduling (with onePrimitiveOneProc option set) is performed, the processor assignment of each primitive is determined. After examining the assignment, the user may want to override the scheduling decision manually. It can be done by setting the adjustSchedule parameter. If that parameter is set, after the automatic scheduling is performed, the procId parameter of each primitive is automatically updated with the assigned processor. The programmer can override the scheduling decision by changing the value of the procId parameter. The
adjustSchedule parameter cannot be YES before any scheduling decision has been made previously. Again, this option is not supported in this release.

Regardless of which scheduling options are chosen, the final stage of the scheduling is to decide the execution order of primitives including send/receive primitives. This is done by a simple list scheduling algorithm in each child target. The final scheduling results are displayed on a Gantt chart.

### The Gantt Chart Display

Demos that use targets derived from CGMultiTarget can produce an interactive Gantt chart display for viewing the parallel schedule.

The Gantt chart display involves a single window for displaying the Gantt chart, which provides scroll bars and zoom buttons for controlling how much of the Gantt chart is shown in the display canvas.

The display canvas represents each primitive schedule as a box drawn through the time interval over which it is scheduled. If the name of a primitive can fit in its box, it is printed inside. A vertical bar inside the canvas identifies primitives which cannot be labeled. The names of the primitives which this bar passes through are printed alongside their respective processor numbers. The bar can be moved horizontally by pressing the left mouse button while on the primitive to be identified. The primitives which the bar passes through are identified by having their icons highlighted in the vem window.

Here is a summary of commands that can be used while the Gantt chart display is active:

- **To change the area of the Gantt chart inside the display canvas:** Use the scroll bars to move along the Gantt chart in the direction desired. Click on the zoom buttons to increase or decrease the size of the Gantt chart.

- **To move the vertical bar to the mouse inside the display window:** Depress and drag the left mouse button inside the display window. The left and right cursor keys move the bar by one time interval; shift-left and shift-right move the bar by ten time intervals.

- **To exit the Gantt chart display program:** Type control-D inside the display window or click on the dismiss button.

A number of limitations exists in the Gantt chart display widget. There are a fixed (hard-coded) number of colors available for coloring processors and highlighting icons. The print function does not work because the font chosen by the font manager is not guaranteed to be Postscript convertible.

### 27.2.4 Interfacing Issues

A framework for interfacing code generation targets with other targets (simulation or code generation) has been developed. The concepts behind this new infrastructure are detailed in [PPL96]. Currently, only a few of our code generation targets support this new infrastructure including:
CGCTarget (CGC Domain), S56XTarget (CG56 Domain), SimVSPrimitive (VHDL Domain).

The code generation targets that support this infrastructure can be mixed arbitrarily in an application specification, and can also be embedded within simulation wormholes (i.e. a CG domain module embedded within a simulation-SDF module).

This infrastructure requires that each target provide CGC communication primitives that can be targeted to the MLDesigner host workstation. The current implementation does not support specialized communication links between two individual code generation targets, but rather builds the customized links from the communication primitives written in C.

### 27.2.4.1 Interface Synthesis between Code Generation Targets

To interface multiple code generation targets, you must set the target parameter for the top-level module to `CompileCGSubsystems`. The target parameters for `CompileCGSubsystems` are identical to those of the `FullyConnected` target. You must declare each individual target in the `childType CompileCGSubsystems` target parameter list. The first of these child targets must be a CGC target whose code will be run on the MLDesigner host workstation. The processor mapping of each primitive is user-specified by setting either the procId primitive parameter or setting the domain for the current module. The interconnect between the primitives to be mapped onto different targets can be totally arbitrary.

### 27.2.4.2 Interface Synthesis between Code Generation and Simulation Domains

The interfacing of code generation targets with simulation targets is more restricted than interfacing only code generation targets. Unlike the previous case, where the primitive interconnect could be arbitrary, we require that the simulation targets be used at a higher level in the user-specification than all of the code generation targets. This restriction enables us to create simulation SDF primitive wrappers for each of the code generation subsystems. This generated primitive can then be added to the user primitive palette by creating an icon for it.

The top-level module for each code generation subsystem should have its target set to either `CompileCGSubsystems` or `CreateSDFPrimitive`. The `CompileCGSubsystems` target should be used if more than one code generation target is used. The `childType` target parameter should list the child targets to use. The first child target listed must be the `CreateSDFPrimitive` target. The `CreateSDFPrimitive` is actually a CGC target that generates ptlang code for all of the communication between the various targets and MLDesigner.

If only CGC primitives are being used in a code generated subsystem, we have no need for the multiprocessor target `CompileCGSubsystems`, but rather can use the uniprocessor CGC target `CreateSDFPrimitive`.

### 27.2.5 Dynamic constructs in CG domain

All multiprocessor code generation domains included in previous releases assumed that the data flow graph is synchronous (or SDF)-that is, the number of tokens consumed and produced by
each primitive does not vary at run time. We also assumed that the relative execution times of
blocks was specified, and did not allow blocks with dynamic behavior, such as the case construct,
data-dependent iteration, and recursion. In simulation, however, data-dependent behavior was
supported by the DDF (Dynamic Data Flow) domain. The current release allows data-dependent
constructs in the code generation domains by a clustering technique and a scheduler called the
CGDDF scheduler.

27.2.5.1 Dynamic constructs as a cluster

Dynamic constructs are specified using predefined graph topologies. For example, an *if-then-else*
construct is represented as a module that consists of two DDF primitives, *Case* and *End-Case*,
and two SDF modules to represent the bodies of the TRUE or FALSE branches. The dynamic
constructs supported by the CGDDF scheduler are *case*, *for*, *do-while*, and *recursion*.
The case construct is a generalization of the more familiar *if-then-else* construct. The topology
of the module is matched against a set of pre-determined topologies representing these dynamic
constructs.

A module is a hierarchical block for structural representation of the program graph. When an
APEG is generated from an SDF graph for parallel scheduling, galaxies are flat-tened. To handle
dynamic constructs as a unit of parallel scheduling, we make a cluster, called a *module cluster*, for
each dynamic construct. The programmer should indicate the systems to be clustered by creating
a module parameter *asFunc* and setting its value to YES. For example, the systems associated
with the TRUE and the FALSE branch of a case construct will have the asFunc parameter as well
as the module of the construct itself.

27.2.5.2 Quasi-static scheduling of dynamic constructs

We treat each dynamic construct as a special SDF primitive and use a static scheduling algorithm.
This SDF primitive is special in the sense that it may need to be mapped onto more than one
processor, and the execution time on the assigned processor may vary at runtime (we assume it is
fixed when we compute the schedule). The scheduling results decide the assignment to and order-
ing of blocks on the processors. At run time, we will not achieve the performance expected from
the compile time schedule, because the dynamic constructs behave differently to the compile-time
assumptions. The goal of the CGDDF scheduler is to minimize the expected makespan of the
program graph at run time.

The type of the dynamic construct and the scheduling information related to the dynamic con-
structs are defined as module parameters. We assume that the run-time behavior of each dynamic
construct is known or can be approximated with a certain probability distribution. For example,
the number of iterations of a for or do-while construct is such a variable; similarly, the depth of
recursion is a variable of the recursion construct. The parameters to be defined are as follows:

```
constructType
(STRING) Default =
There is no default, the initial value is the value of the module param-
ter.
Type of the dynamic construct. Must be one of case, for, doWhile, or recur (case insensitive!).
```
paramType (STRING) Default = geometric
Type of the distribution. Currently, we support geometric distribution, uniform distribution, and a general distribution specified by a table.

paramGeo (FLOAT) Default = 0.5
Geometric constant of a geometric distribution. Its value is effective only if the geometric distribution is selected by paramType. If constructType is case, this parameter indicates the probability of branch 1 (the TRUE branch) being taken. If there are more than two branches, use paramFile to specify the probabilities of taking each branch.

paramMin (INT) default = 1
Minimum value of the uniform distribution, effective only when the uniform distribution is chosen.

paramMax (INT) default = 10
Maximum value of the uniform distribution, effective only when the uniform distribution is chosen.

paramFile (STRING) default = defParams
The name of a file that contains the information on the general distribution. If the construct is a case construct, each line contains the probability of taking a branch (numbered from 0). Otherwise, each line contains the integer index value and the probability for that index. The indices should be in increasing order.

Based on the specified run-time behavior distribution, we determine the compile-time profile of each dynamic construct. The profile consists of the number of processors assigned to the construct and the (assumed) execution times of the construct on the assigned processors. Suppose we have a for construct. If the loop body is scheduled on one processor, it takes 6 time units. With two processors, the loop body takes 3 and 4 time units respectively. Moreover, each iteration cycle can be paralleled if skewed by 1 time unit. Suppose there are four processors: then, we have to determine how many processors to assign to the construct and how many times the loop body will be scheduled at compile time. Should we assign two processors to the loop body and parallelize two iteration cycles, thus taking all 4 processors? Or should we assign one processor to the loop body and parallelize three iteration cycles, thus taking 3 processors as a whole? The CGDDF scheduler uses a systematic approach based on the distribution to answer these tricky scheduling problems [Ha92]. We can manually determine the number of assigned processors by defining a fixedNum module parameter. Note that we still have to decide how to schedule the dynamic construct with the given number of processors. The Gantt chart display will show the profile of the dynamic construct.

### 27.2.5.3 DDF-type Primitives for dynamic constructs

A code generation domain should have DDF primitives to support dynamic constructs with the CGDDF scheduler. For example, the Case and EndCase primitives are used in the case, do-while, and recursion constructs, which differ from each other in the connection topology of these DDF primitives and SDF modules. Therefore, if the user wants to use one of the above three
27.2 CG Domain

dynamic constructs, there is no need to write a new DDF primitive. Like a DDF primitive, the Case primitive has dynamic output portholes as shown in the CGCCase.pl file. For example:

```plaintext
outmulti
{
    name { output }
    type { =input }
    num { 0 }
}
```

The for construct consists of an UpSample type primitive and a DownSample type primitive, where UpSample and DownSample are not the primitive names but the types of the primitives: if a primitive produces more than it consumes, it is called an UpSample primitive. In the preprocessor file, we define a method readTypeName, as shown below.

```plaintext
method
{
    name { readTypeName }
    access { public }
    type { "const char *" }
    code { return "UpSample"; }
}
```

Examples of UpSample type primitives are Repeater and DownCounter. These primitives have a data input and a control input. The number of output data tokens is the value of the integer control input, and is thus data-dependent. Conversely, we can design a DownSample primitive that has the following method:

```plaintext
method
{
    name { readTypeName }
    access { public }
    type { "const char *" }
    code { return "DownSample"; }
}
```

Examples of DownSample type primitives are LastOfN, and SumOfN. These primitives have a data input and a control input. The number of input tokens consumed per invocation is determined by the value of the control input.

As explained above, all customized DDF-type primitives for dynamic constructs will be either an UpSample type or a DownSample type. We do not expect that a casual user will need to write new DDF primitives if we provide some representative UpSample and DownSample primitives in the corresponding code generation domains. Currently, we have DDF primitives in the CGC code generation domain only.

### 27.2.6 Primitives

As mentioned earlier, primitives in the CG domain are used only to test and debug schedulers. The primitives in the palette generate only comments, and allow the user to model primitive parameters that are relevant to schedulers such as the number of samples produced and consumed on
each firing, and the execution time of the primitive. By default, any primitive that is derived from
CGPrimitive (the base class for all code generation primitives), including all the primitives in
the CG domain, have the parameter procId. This parameter is used during manual partitioning
to specify the processor that the primitive should be scheduled on. The default value of the param-
eter is −1 which specifies to the scheduler that automatic partitioning should be used. Processors
are numbered 0, 1, 2, · · ·; hence, if the parameter is set to 1, then the primitive will be scheduled
on the second processor in the architecture. Note that the target parameter manualAssignment
should be YES for this to work; if manualAssignment is NO, then the value of procID will
be ignored (due to a bug in the current implementation). If the user wants to specify a processor
assignment for only a subset of the primitives in the system, and do automatic assignment for the
remaining primitives, then this is currently not possible. It can be done in a roundabout manner
using the resources parameter. This is done by defining a resources parameter in the prim-
ine. The value of this parameter is a number that specifies the processor on which this primitive
should go on. The target parameter resources is left empty. Then, the scheduler will inter-
pret the value of the resources parameter as the processor on which the primitive should be
scheduled; primitives that do not specify any resources are mapped automatically by the scheduler.

The resources parameter just described is used mainly for specifying any special resources that
the primitive might require in the system. For example, an A/D converter primitive might require
an input port, and this port is accessible by only a subset of all the processors in the system; in
this case, we would like the A/D primitive to be scheduled on a processor that has access to the
input port. In order to specify this, the resources parameter in the primitive is defined and set
to a string containing the name of the resource (e.g., input_port). Use commas to delimit mul-
tiple resources (e.g., input_port, output_port). The target parameter resources is specified
using the same resource names (e.g., input_port). The scheduler will then schedule primitives
that request certain resources on processors that have them. By default, primitives do not have the
resources parameter.

The following gives an overview of CG domain primitives.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiIn</td>
<td>Takes multiple inputs and produces one output.</td>
</tr>
<tr>
<td>MultiInOut</td>
<td>Takes multiple inputs and produces multiple outputs.</td>
</tr>
<tr>
<td>MultiOut</td>
<td>Takes one input and produces multiple outputs.</td>
</tr>
<tr>
<td>RateChange</td>
<td>Consumes consume samples and produces produce samples.</td>
</tr>
<tr>
<td>Sink</td>
<td>Swallows an input sample.</td>
</tr>
<tr>
<td>Source</td>
<td>Generic code generator source primitive; produces a sample.</td>
</tr>
<tr>
<td>Switch</td>
<td>This primitive requires an BDF scheduler. It switches input events to one of two outputs, depending on the value of the control input.</td>
</tr>
<tr>
<td>Through</td>
<td>Passes data through. The run time can be set to reflect computation time.</td>
</tr>
<tr>
<td>TestMultirate</td>
<td>(five icons) The TestMultirate primitives parallel those in the SDF domain. These primitives are useful for testing schedulers. The number of tokens produced and consumed can be specified for each</td>
</tr>
</tbody>
</table>
primitive, in addition to its execution time.

## 27.2.7 Demos

There are four demos in the CG domain.

**NOTE:** Since the CG domain is not supported, the demos mentioned here are not directly accessible through demo palettes, although they are included in the MLDesigner package suite. To access the demos, use the file view tree and look in $MLD/MLD_Experimentals/CG/demo.

*Please remember that the demos are not tested and may be unable to run properly!*

- **pipeline**
  
  This demo demonstrates a technique for generation of pipelined schedules with MLDesigner’s parallel schedulers, even though MLDesigner’s parallel schedulers attempt to minimize makespan (the time to compute one iteration of the schedule) rather than maximize the throughput (the time for each iteration in the execution of a very large number of iterations). To retime a graph, we simply add delays on all feedforward arcs (arcs that are not part of feedback loops). We must not add delays in feedback loops as that will change the semantics. The effect of the added delays is to cause the generation of a pipelined schedule. The delays marked as "(conditional)" in the demo are parameterized delays; the delay value is zero if the system parameter retime is set to NO, and is 100 if the system parameter is set to YES. The delay in the feedback loop is always one.

  Schedules are generated in either case for a three-processor system with no communication costs. If this were a real-life example, the programmer would next attempt to reduce the ”100” values to the minimum values that enable the retimed schedule to run; there are other constraints that apply as well when there are parallel paths, so that corresponding tokens arrive at the same primitive. If the system will function correctly with zero values for initial values at points where the retiming delays are added, the generated schedule can be used directly. Otherwise, a preamble, or partial schedule, can be prepended to provide initial values.

- **schedTest**
  
  This is a simple multiprocessor code generation demo. By changing the parameters in the RateChange primitive, you can make the demo more interesting by observing how the scheduler manages to parallelize multiple invocations of a primitive.

- **Sih-4-1**
  
  This demo allows the properties of the parallel scheduler to be investigated, by providing a system in which the run times of primitives, the number of processors, and the communication cost between processors can be varied. The problem, as presented by the default parameters, is to schedule a collection of data flow actors on three processors with a shared bus connecting them. Executing the demo causes a Gantt chart display to appear, showing the partitioning of the actors onto the...
three processors. Clicking the left mouse button at various points in the schedule causes the associated primitives to be highlighted in the system palette.

After exiting from the Gantt chart display, code is written to a separate file for each processor (here the "code" is simply a sequence of comments written by the dummy CG primitives). It is interesting to explore the effects of varying the communication costs, the number of processors, and the communication topology. To do so, execute the edit-target command. A display of possible targets comes up. Of the available options, only SharedBus and FullyConnected will use the parallel scheduler, so select one of them and click on Ok. Next, a display of target parameters will appear. The interesting ones to vary are nprocs, the number of processors, and sendTime, the communication cost. Try using two or four processors, for example. Sometimes you will find that the scheduler will not use all the processors. For example, if you make the communication cost very large, everything will be placed on one processor. If the communication cost is 1 (the default), and four processors are provided, only three will be used.

This is a simple demo of the dummy primitives provided in the CG domain. Each primitive, when executed, adds code to the target. On completion of execution for two iterations, the accumulated code is displayed in a pop-up window, showing the sequence of code produced by the three primitives.

### 27.3 CGC Domain

#### 27.3.1 Introduction

The CGC domain generates code for the C programming language. This domain supports both synchronous data flow (SDF, see "SDF Domain" on 18-1) and Boolean-controlled data flow (BDF, see "BDF Domain" on 20-1) models of computation. The model associated with a particular program graph is determined by which target is selected. The bdf-CGC target supports the BDF model, while all other targets in the CGC domain support only the SDF model. Code can be generated for both single-processor and multi-processor computers. The targets that support single processors include default-CGC, Makefile.C, TclTk_Target, and bdf-CGC. The multi-processor targets are unixMulti_C and NOWam.

#### 27.3.2 CGC Targets

The targets of the CGC domain generate C code from data flow program graphs. Code generation is controlled by the host, directory, and file parameters as described in "Targets" in ch. 27.2. The command used to compile the code is determined by the compileCommand, compileOptions, and linkOptions parameters. Compilation and execution are controlled by the display?, compile?, and run? parameters. The other parameters common to all
CGC targets are listed below. Not all of these parameters are made available to the user by every target, and some targets define additional parameters.

- **staticBuffering**: (INT) Default = TRUE
  If TRUE, then attempt to use static, compile-time addressing of data buffers between primitives. Otherwise, generate code for dynamic, run-time addressing.

- **funcName**: (STRING) Default = main
  The name of the main function. The default value of main is suitable for generating stand-alone programs. Choose another name if you wish to use the generated code as a procedure that is called from your own main program.

- **compileCommand**: (STRING) Default = cc
  Command name of the compiler.

- **compileOption**: (STRING) Default =
  Options passed to the compiler. The default is the empty string.

- **linkOptions**: (STRING) Default = -lm
  Options passed to the linker.

- **resources**: (STRING) Default = STDIO
  List of abstract resources that the host computer has.

### 27.3.2.1 Single-Processor Targets

The default-CGC target generates C code for a single processor from a SDF program graph. The parameters available to the user are shown below.

```
compile?    file    Looping Level
compileCommand funcName resources
compileOptions host run?
directory    linkOptions staticBuffering
display?
```

The `Makefile_C` target compiles CGC binaries with makefiles so that compile time architecture and site dependencies can be handled. The `Makefile_C` target generates a small makefile that is rcp’d over to the remote machine. The generated makefile is named after the system. If the system is called `bigBang`, then the makefile will be called `bigBang.mk`. We name the generated makefiles so that more than one makefile can exist in the users’ directory.

The generated makefile uses `$MLD/lib/cgc/makefile_C.mk` as a starting point, and then appends lines to it. The generated makefile includes `$MLD/mk/config-$PTARCH.mk`, which determines architecture and site dependencies, such as which compiler to use, or where the X11 include files are. The user may modify `makefile_C.mk` and add site-dependent rules and variables there. If the user wants to have site dependent include files on the remote machines, then they could add include `$(ROOT)/mk/mysite.mk` to `makefile_C.mk`, and that file would be
included on the remote machines at compile time.

On the remote machine, the Makefile_C target assumes:

- $MLD and $PTARCH are set on the remote machine when rshing.
- $MLD/mk/config-$PTARCH.mk and any makefile files included by that file are present.
- A make binary is present. The Makefile_C target does not assume GNU make, so the default makefile_C.mk does not include mk/common.mk. The reason not to assume GNU make is that we are not sure what the user’s path is like when they log in. The user can require that GNU make be used by setting the skeletonMakefile target parameter to the name of a makefile that requires GNU make.

If the remote machine does not fulfill these constraints, then the user should use the Default_C target.

skeletonMakefile  (STRING) Default=
The default value of this target parameter is the empty string, which means that we use
$MLD/lib/cgc/makefile_C.mk
as the skeleton makefile. If this parameter is not empty then the value of the parameter refers to the skeleton makefile to be copied into the generated makefile.

appendToMakefile  (INT) Default = 1
This target parameter controls whether we append rules to the generated makefile or just copy it over to the remote machine. In the default situation, appendToMakefile is true and we append our rules after copying
$MLD/lib/cgc/makefile_C.mk

The parent target of the Makefile_C target is default-CGC. If the parent target parameter compileOptions is set, then we process any environment variables in that string, and then add it to the end of the generated makefile as part of OTHERCFLAGS=. In a similar fashion, the parent target parameter linkOptions ends up as part of the right-hand side of LOADLIBES=.

The TclTk_Target target, which is derived from the Makefile_C target, must be used when Tcl/Tk primitives are present in the program graph. The initial default of one parameter differs from that of the parent target.

skeletonMakefile  (STRING) Default=$MLD/lib/cgc/TclTk_Target.mk
The TclTk_Target overrides this parent target parameter and sets it to the name of a skeleton makefile to be copied into the generated makefile.

The bdf-CGC target supports the BDF model of computation. It must be used when BDF primitives are present in the program graph. It can also be used with program graphs that contain only SDF primitives. The bdf-CGC target has the same parameters as the default-CGC target with the exception that the Looping Level parameter is absent. This is because a loop-generating algorithm is always used for scheduling.
27.3 CGC Domain

27.3.2.2 Multi-Processor Targets

Currently, the CGC domain supports two multi-processor targets: unixMulti_C and NOWam. The unixMulti_C target generates code for multiple networked workstations using a shared bus configuration for scheduling purposes. Inter-processor communication is implemented by splicing send/receive primitives into the program graph. These communication primitives use the TCP/IP protocol. In addition to the target parameters mentioned above, the unixMulti_C target has extended parameters:

- adjustSchedule
- ignoreIPC
- amortizedComm
- inheritProcessors
- childType
- logFile
- compile?
- machineNames
- resources
- directory
- manualAssignment
- run?
- display?
- nameSuffix
- file
- nprocs
- usersCluster
- ganttChart
- onePrimitiveOneProc
- tabular

**portNumber** (INT) Default = 7654
The starting TCP/IP port number used by send/receive primitives. The port number is incremented for each send/receive pair. It is the responsibility of the user to ensure that the port number does not conflict with any that may already be in use.

**machineNames** (STRING) Default = herschel
The host names of the workstations which form the multi-processor. The names should be separated by a comma (',').

**nameSuffix** (STRING) Default =
The default is the empty string. The domain suffix for the workstations named in machineNames. If left blank, which is the default, then the workstations are assumed to be part of the local domain. Otherwise, specify the proper domain name, including a leading period. This string is appended to the names in machineNames to form the fully qualified host names.

The NOWam target uses Networks Of Workstations (NOW) active messages to communicate between machines. The NOW project is an effort to use many commodity workstations to create a building-wide supercomputer. For more information about the NOW project, see http://now.cs.berkeley.edu. The NOWam target has the following target parameters:

**machineName** (STRING) Default = lucky, babbage
The host names of the workstations which form the multi-processor. The names should be separated by a comma (','). The NOWam target will not work on the local machines, the machines named by this parameter must be remote machines. Note that the default of this parameter differs from the default in the UnixMulti_C target.

**nameSuffix** (STRING) Default =
The default is the empty string. See the description of `nameSuffix` above.

### 27.3.3 An Overview of CGC Primitives

The primitives are divided into the categories sources, sinks, arithmetic functions, nonlinear functions, control, Sun UltraSparc VIS-conversion, signal processing, boolean-controlled data flow, Tcl/Tk and higher-order function (HOF) primitives. Icons for delay, bus, and BlackHole appear in most palettes for easy access. Many of the primitives in the CGC domain have equivalent counterparts in the SDF domain. Brief descriptions of the primitives unique to the CGC domain are given in the following sections.

#### 27.3.3.1 Source Primitives

Source primitives have no inputs and produce data on their outputs. The following primitives are equivalent to the SDF primitives of the same name: Const, IIDUniform, Ramp, Rect, singen, WaveForm, TclScript, TkSlider, RampFix, RectFix, RampInt, expgen. Primitives that are unique to the CGC domain are described briefly below.

- **StereoIn**: Reads Compact Disc format audio data from a file given by `fileName`. The file can be the audio port `/dev/audio`, if supported by the workstation. The data read is linear 16 bit encoded and stereo (2 channel) format.
- **TkStereoIn**: Just like StereoIn, except that a Tk slider is put in the master control panel to control the volume.
- **MonoIn**: Reads mono (1 channel) data with either `linear16` or `ulaw8` encoding from a file given by `fileName`. The file can be the audio port `/dev/audio`, if supported by the workstation.
- **TkMonoIn**: Just like MonoIn, except that a Tk slider is put in the master control panel to control the volume.
- **SGImonoIn**: (SGI only) Average the stereo audio output of an SGIAudioIn primitive into one mono output.
- **SGIAudioIn**: (SGI only) Get samples from the audio input port.
- **dtmfKeyPad**: Generate a Dual-Tone Modulated Frequency (DTMF) signal.
- **TkCheckButton**: A simple Tk on/off input source.
- **TkEntry**: Output a constant signal with value determined by a Tk entry box (default 0.0).
- **TkImpulse**: Output a specified value when a button is pushed. Optionally synchronize by halting until the button is pushed.
- **TkRadioButton**: Graphical one-of-many input source.
27.3.3.2 Sink Primitives

Sink primitives have no outputs and consume data on their inputs. The following primitives are equivalent to the SDF primitives of the same name: XMgraph, XYgraph, Xscope, TkBarGraph, TkPlot, TKXYPlot, TclScript, Printer. Primitives that are unique to the CGC domain are described briefly below.

- **StereoOut**: Writes Compact Disc audio format to a file given by fileName. The file can be the audio port /dev/audio, if supported by the workstation. The data written is linear 16 bit encoded and stereo (2 channel) format.
- **TkStereoOut**: Just like StereoOut except that Tk sliders are put in the master control panel to control the volume and balance.
- **TkMonoOut**: Just like MonoOut except that Tk sliders are put in the master control panel to control the volume.
- **MonoOut**: Writes mono (1 channel) data with either linear16 or ulaw8 encoding to a file given by fileName. The file can be the audio port /dev/audio, if supported by the workstation. If the aheadlimit parameter is non-negative, then it specifies the maximum number of samples that the program is allowed to compute ahead of real time.
- **SGIAudioOut**: (SGI Only) Put samples into an audio output port.
- **SGIMonoOut**: (SGI Only) A module that takes a mono output and drives the stereo SGIAudioOut primitive below.

27.3.3.3 Arithmetic Primitives

Arithmetic primitives perform simple functions such as addition and multiplication. All these primitives are equivalent to the SDF primitives of the same name: Add, Gain, Integrator, Mpy, Sub.

27.3.3.4 Nonlinear Primitives

Nonlinear primitives perform simple functions. The following primitives are equivalent to the SDF primitives of the same name: Abs, cexp, conj, Cos, Dirichlet, Exp, expjx, Floor, Limit, Log, MaxMin, Modulo, ModuloInt, OrderTwoInt, Reciprocal, Sgn, Sin, Sinc, Sqrt, powerEst, Quant, Table, TclScript. Primitives that are unique to the CGC domain are described briefly below.

- **Expr**: General expression evaluation. This primitive evaluates the expression given by the expr parameter and writes the result on the output. The default expression, which is $\text{ref(in#1)}$, simply copies the first input to the output.
- **fm**: Modulate a signal by frequency.
- **Thresh**: Compares input values to threshold. The output is 0, if input;=threshold, otherwise it is 1.
- **xor**: Exclusive-OR two signals.
27.3.3.5 Control Primitives

Control primitives are used for routing data and other control functions. The following primitives are equivalent to the SDF primitives of the same name: Fork, Chop, ChopVarOffset, Commutator, DeMux, Distributor, DownSample, Mux, Repeat, UpSample. Primitives that are unique to the CGC domain are described briefly below.

- **Collect**: Takes multiple inputs and produces one output. This primitive does not generate code. In multiprocessor code generation, it is automatically attached to a porthole if it has multiple sources. Its role is just opposite to that of the Spread primitive.

- **Copy**: ‘Copy’ primitives are added if an input/output PortHole is a host/embedded PortHole and the buffer size is greater than the number of Particles transferred.

- **Delay**: Delay an input by delay samples.

- **Sleep**: Suspend execution for an interval (in milliseconds). The input is passed to the output when the process resumes.

- **Spread**: Takes one input and produces multiple outputs. This primitive does not generate any code. In multiprocessor code generation, this primitive is automatically attached to a porthole whose outputs are passed to more than one destination (one ordinary block and one Send primitive, more than one Send primitive, and so on.)

27.3.3.6 Logic primitives

27.3.3.7 Conversion Primitives

Conversion primitives are used to convert between complex and real numbers. All of the primitives are equivalent to the SDF primitives of the same name: CxToRect, PolarToRect, RectToCx, RectToPolar.

27.3.3.8 Signal Processing Primitives

The following primitives are equivalent to the SDF primitives of the same name: DB, FIR, FIRFix, FFTCx, GAL, GGAL, Goertzel, LMS, LMSOscDet, LMSTkPlot. Primitives that are unique to the CGC domain are described briefly below.

- **GoertzelPower**: Second-order recursive computation of the power of the kth coefficient of an N-point DFT using Goertzel’s algorithm. This form is used in touch tone decoding.

- **ParametricEq**: A two-pole, two-zero parametric digital IIR filter (a biquad).

- **rms**: Calculate the Root Mean Squared of a signal.
27.3 CGC Domain

### 27.3.3.9 BDF Primitives

BDF primitives are used for conditionally routing data. These primitives require the use of the bdf-CGC target. Unlike their simulation counterparts, these primitives can only transfer single tokens in one firing.

**Select**
This primitive requires a BDF scheduler. If the value on the control line is nonzero, trueInput is copied to the output; otherwise, falseInput is.

**Switch**
This primitive requires a BDF scheduler. Switches input events to one of two outputs, depending on the value of the control input. If control is true, the value is written to trueOutput; otherwise it is written to falseOutput.

### 27.3.3.10 Tcl/Tk Primitives

Tcl/Tk primitives require the use of the TclTk_Target target. They can be used to provide an interactive user interface with Tk widgets. A palette of Tcl/Tk primitives are available in the CGC domain. Most of these primitives are described in sources, sinks and nonlinear palettes.

**TkParametricEq**
Just like ParametricEq primitive, except that a Tk slider is put in the master control panel to control the gain, bandwidth, and center and cut-off frequencies.

### 27.3.4 An Overview of CGC Demos

The demos are divided into categories: basic, multirate, signal processing, multi-processor, sound, Tcl/Tk, BDF, HOF and SDF-CGC wormhole demos. Many of the demos in the CGC domain have equivalent counterparts in the SDF or BDF domains. Brief descriptions of the demos unique to the CGC domain are given in the sections that follow.

**NOTE:** Since the CGC domain is not supported, the demos mentioned here are not directly accessible through demo palettes, although they are included in the MLDesigner package suite. To access the demos, use the file view tree and look in $MLD/MLD_Experiments/CGC/demo.

*Please remember that the demos are not tested and may be unable to run properly!*

### 27.3.4.1 Basic Demos

The following demos are equivalent to the SDF demos of the same name: butterfly, chaos, integrator, quantize. The other demos in this palette are described briefly below.

**chaoticBits**
Chaotic Markov map with quantizer to generate random bit sequence.

**nonlinear**
This simple system plots four nonlinear functions over the range 1.0 to 1.99. The four functions are exponential, natural logarithm, square root, and reciprocal.
This demo is a slight modification of the nonlinear demo. It uses the pragma mechanism to indicate the parameters that are to be made settable from the command-line.

pseudoRandom
Generate pseudo-random sequences.

27.3.4.2 Multirate Demos

The following demos are equivalent to the SDF demos of the same name: interp, filterBank. The other demos in this palette are described briefly below.

upsample
This simple up-sample demo tests static buffering. Each invocation of the XMgraph primitive reads its input from a fixed buffer location since the buffer between the UpSample primitive and the XMgraph primitive is static.

loop
This demo demonstrates the code size reduction achieved with a loop-generating scheduling algorithm.

27.3.4.3 Signal Processing Demos

The following demos are equivalent to the SDF demos of the same name: adaptFilter, dft. The animatedLMS demo is described in "Tcl/Tk Demos".

DTMFCodec
Generate and decode touch tones.

iirDemo
Two equivalent implementations of IIR filtering. One of the implementations uses the IIR primitive.

27.3.4.4 Multi-Processor Demos

MLDesigner contains two multi-processor targets, unixMulti_C and NOWam. The demos in each target sub-palette are the same. These demos would actually run faster on a single processor, but they do serve as a ‘proof of concept’.

adaptFilter_multi
This is a multi-processor version of the adaptFilter demo. The graph is manually partitioned onto two networked workstations.

spread
This system demonstrates the Spread and Collect primitives. It shows how multiple invocations of a primitive can be scheduled onto more than one processor.

27.3.4.5 Sound-Making Demos

Your workstation must be equipped with an audio device that can accept 16-bit linear or µ-law encoded PCM data, for these demos to work.

alive
(SGI Only) Processes audio in real time, with an effect similar to the effects Peter Frampton used in the late 70’s rock album ‘Frampton Comes Alive’.
This demo generates the same dual-tone multi-frequency tones you hear when you dial your telephone. The interface resembles the keypad of a telephone.

This demo uses frequency modulation (FM) to synthesize a tone on the workstation speaker. You can adjust the modulation index, pitch, and volume in real time.

FM synthesis with a spectral display.

This demo generates tones on the workstation speaker with decaying amplitude envelopes using frequency modulation synthesis. You can make tones by pushing a button. You can adjust the pitch, modulation index, and volume in real time.

Generate a sound to play over the workstation speaker (or headphones).

Produce a sound made by adding a fundamental and its harmonics in amounts controlled by sliders.

This demo generates sinusoidal tones. You can control the pitch with a piano-like interface.

This demo produces a tremolo (amplitude modulation) effect on the workstation speaker. You can adjust the pitch, modulation frequency, and volume in real time.

These demos show off the capabilities of the Tcl/Tk primitives, which must be used with the TclTk.Target target. Graphical user interface widgets are used to control input parameters and to produce animation. Many of these demos also produce sound on the workstation speaker with the TkMonoOut primitive. The following audio demos are documented in the previous section: dtmf, fm, audioio, impulse, synth, tremolo.

This demo is a simplified version of the SDF demo of the same name.

This demo exhibits sinusoidal motion with a ball moving back and forth.

This demo is the same as the ball demo except that animation is updated asynchronously.

Generate a number of sinusoids with controllable additive noise.

This demo shows the use of several kinds of Tk widgets for user input. Push buttons generate tones or noise, and sliders adjust the frequency and volume in real time.

This demo shows the movements of the Sun, Venus, Earth, and Mars in a Ptolemaic (Earth-centered) system.

Demonstrate the TkXYPlot primitive.
27.3.4.7 BDF Demos

Some demonstrations of how to use BDF primitives in the CGC domain. The timing demo is equivalent to the BDF simulation demo of the same name. The demos bdf-if and bdf-doWhile are equivalent to the BDF simulation demos named ifThenElse and loop.

27.3.4.8 SDF-CGC Wormhole demos

Some systems that demonstrate the use of the CreateSDFPrimitive CGC target, which allows cgc primitives that are reloaded back into MLDesigner for use inside the SDF domain. The SDF-CGC Wormhole demos are found under the "Mixed Domain Demos" palette.

- CDtoDAT: Convert two sine waves sampled at CD sample rate to DAT sample rate. The outer module is in the SDF domain, while the cd2dat module is in the CGC domain. cd2dat uses the CreateSDFPrimitive target.
- wormTest: A simple test of the CreateSDFPrimitive target.
- fixCGC: Another simple test of the CreateSDFPrimitive target.

27.3.4.9 Tycho Demos

These demos demonstrate the use of the TychoTarget to create customized Control Panels. Graphical user interface widgets are used to control input and output parameters and to produce animation. The demos make use of the TkStereoIn and TkStereoOut to record and play sound on the workstation speaker.

- audioio: This is a simple real time audio demonstration which illustrates MLDesigner’s ability to support CD quality audio.
- graphicEq: This demo consists of 10 band-pass filters with center frequencies spaced out by octaves. Using the customized control panel, you can adjust the gain of each band-pass filter, the record and play volumes and balance in real time.
- parametricEq: In this demo, there is a single band of parametric equalization, with control over the band frequency, band width, and band gain. The frequency range is settable; in the future, it will also be possible to select low-pass, band-pass, or high-pass filtering as well.
- tonecontrol: The demo consists of one of each of the high, band and low-pass filters. There is a single control panel, with control over the band gain for each filter.
27.4 CG56 Domain

27.4.1 Introduction

The CG56 domain generates assembly code for the Motorola 56000 series of digital signal processors. The graphs that we can describe in this domain follow the synchronous data flow (SDF) model of computation. SDF allows us to schedule the Blocks and allocate all the resources at compile time.

The Motorola 56000 series are fixed-point digital signal processors. The 56000 and 56001 processors have 24-bit data and instructions, and operate at a maximum clock rate of 40 MIPS. The 56100 processor has 16-bit data and instructions, operates at a maximum rate of 30 MIPS, and has analog/digital and digital/analog converters integrated on the chip. The 56301 has 24-bit data and instructions, operates at a maximum rate of 80 MIPS, and has several built-in input/output interfaces. Although the processors have pipelines of different lengths, the assembly code is backward compatible. The CG56 domain generates assembly code for the 56000 processor and has been tested on the Motorola simulator and on a 56001 board.

Since the 56000 processors are fixed point, the floating point data type has no meaning in the CG56 domain. Fixed-point values can take on the range \([-1,1)\). The most positive value is \(1 - 2^{-23}\) for the 56000 and 56300, and \(1 - 2^{-15}\) for the 56100. The domain defines a new constant ONE set to this maximum positive value. In this chapter, whenever data types are not mentioned, fixed-point is meant. The complex data type means a pair of fixed-point numbers. The complex data type is only partially supported in that it is not supported for primitives that have anytype inputs or outputs, except for fork primitives. Integers are the same length as the fixed-point representation. Matrix data types are not supported yet.

Some of the demos use the Motorola 56000 assembler and simulator. You do not need to have a 56000 chip to run the simulator demos, the assembler and simulator are available for downloading from Motorola at http://www.mot.com/SPS/DSP/developers/clas.html.

27.4.2 An overview of CG56 primitives

For the CG56 domain, the primitive library is large enough that it has been divided into subpalettes as was done with the SDF main palette. The palettes are Signal Sources, I/O, Arithmetic, Nonlinear Functions, Logic, Control, Conversion, Signal Processing, and Higher Order Functions. The primitives on the Higher Order Functions (HOF) palette are used to help lay out schematics graphically. The HOF primitives are in the HOF domain, and not the CG56 domain. The names of the others palettes are modeled after the SDF primitive palettes of the same name, except the I/O palette which contains target-specific I/O primitives for the Ariel S-56X DSP board and the Motorola 56001 simulator. Each palette is summarized in more detail below.

At the top of each palette, for convenience, are instances of the delay icon, the bus icon, and the following primitive:
27.4.2.1 Source primitives

Source primitives are primitives with only outputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli.

- **Impulse**: Generate a single impulse of size `impulseSize` (default ONE).
- **IIDGaussian**: Generate a white Gaussian pseudo-random process with mean 0 and standard deviation 0.1. A Gaussian distribution is realized by summing `noUniforms` (default 16) number of uniform random variables. According to the central limit theorem, the sum of `N` random variables approaches a Gaussian distribution as `N` approaches infinity.
- **IIDUniform**: Generate an i.i.d. uniformly distributed pseudo-random process. Output is uniformly distributed between `range` and `range` (default ONE).
- **Tone**: Generate a sine or cosine wave using a second order oscillator. The wave will be of amplitude (default 0.5), frequency (default 0.2), and `calcType` (default "sin")

27.4.2.2 I/O Primitives

I/O primitives are target specific primitives that allow input and output of stimuli to a target architecture. Currently there are I/O primitives for both the Ariel S-56X DSP and the Motorola 56k simulator which are divided hierarchically.

**Motorola 56000 Simulator I/O Primitives**

- **ReadFile**: Read fixed-point ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.
- **IntReadFile**: Read integer ASCII data from a file. The simulation can be halted on end-of-file, or the file contents can be periodically repeated, or the file contents can be padded with zeros.
- **WriteFile**: Write data to a file. The simulator dumps the data presented at the input of this primitive into a specified file.
- **Xgraph**: This primitive shares the same parameters as its SDF and CGC primitive equivalents. However, with this primitive, you can only have one input signal.

**Ariel S-56X DSP Board I/O Primitives**

To use the s56xio palette you will need access to an S-56X DSP board. These blocks are divided into three sub-categories: generic S-56X, QDM S-56X and CGC-S56X. The QDM primitives
requires installing qdm, a debugger for DSP systems which was developed by Phil Lapsley at U.C. Berkeley. Qdm is currently available from Mike Peck, the designer of the S-56X board.

**Generic S-56X**

- `adjustableGainGX`: Create an interactive adjustable gain using `HostSliderGX`.
- `da`: Send the input to both input ports of the SSI primitive.
- `HostAOut`: Output data from the DSP to host via host port asynchronously.
- `HostSldrGX`: Generate an athena widget slider for interactive asynchronous input over the host port.
- `MagnavoxIn`: Read data from a Magnavox CD player.
- `Magnavox`: Read data from and write data to a Magnavox CD player.
- `MagnavoxOut`: Write data to a Magnavox CD player.
- `PrPrtAD`: Read from the A/D in Ariel ProPort.
- `PrPrtADDA`: Read from the A/D and write to the D/A on the Ariel ProPort. To use both the A/D and D/A on a ProPort you must use this primitive and not the separate A/D and D/A primitives.
- `PrPrtDA`: Write to the D/A on the Ariel ProPort.
- `SSI`: A generic input/output primitive for the DSP56001 SSI port.
- `SSISkew`: Interface to the 56001 SSI’s port with timing-skew capability.

**QDM S-56X**

To use these primitives you must have qdm installed and be using the uniprocessor s-56x target. The target parameter `monitor` must be set to `qdmterm_s56x -run`.

- `HostButton`: Graphical two-valued input source. There are two types of buttons: push-buttons and check-buttons. Both present a single button to the user that may be “pressed” with the mouse. The buttons differ in the semantics of the push. When the pushbutton is pressed, the `onVal` parameter is output, otherwise `offVal`.
- `HostMButton`: Graphical one-of-many input source. The primitive always outputs one of a finite number of values: the output is controlled by the user selecting one of several buttons. Exactly one button in the group is on.
- `HostSldr`: Graphical host slider for asynchronous input source.
- `SwitchDelay`: This module synchronously switches between the input value and the value of the input delayed by `TotalDelay` (default 8000) samples.
- `adjustableGain`: A user adjustable gain, uses `HostSlider`.

---

1Mike Beck, Berkeley Camera Engineering, http://www.bcam.com
CGC-S56X

checkButtonInt This module creates a Tk check button widget that produces the given onValue (default 1) when pressed and offValue (default 0) otherwise.

checkButton This module creates a Tk check button widget that produces the given onValue (default 1.0) when pressed and offValue (default 0.0) otherwise.

radioButtonInt This module creates a Tk radio button widget that allows the user to select from among a set of possible output values given by pairs (default "One 1” “Two 2”).

radioButton This module creates a Tk radio button widget that allows the user to select from among a set of possible output values given by pairs (default "One 1” “Two 2”)

slider This module creates a Tk slider widget that produces the given value indicated by the slider position which is between low (default 0.0) and high (default 1.0) and initially set to value (default 0.0).

adjustableGain This module multiplies the input by a gain value taken from a Tk slider position between low (default 0.0) and high (default 1.0), which is initially set to value (default 0.0).

SwitchDelay This module synchronously switches between the input value and the value of the input delayed by TotalDelay (default 8000) samples.

s56XPlot This module plots the input interactively using TkPlot.

Xgraph This module simply contains a CGCXgraph primitive for use in a CG56 module. The module parameters are identical to those of the enclosed primitive.

PeekPoke Nondeterminate communication link that splices in a peek/poke pair. In this context, it provides a link between the S-56X Motorola 56001 board and the workstation.

27.4.2.3 Arithmetic primitives

Add Output the sum of the inputs. If saturation is set to yes, the output will saturate.

Sub Outputs the ”pos” input minus all of the “neg” inputs.

Mpy Outputs the product of all of the inputs.

Gain The output is set the input multiplied by a gain term. The gain must be in [-1,1).

AddCx Output the complex sum of the inputs. If saturation is set to yes, the output will saturate.

SubCx Outputs the ”pos” input minus all of the “neg” inputs.
27.4 CG56 Domain

*MpyCx*  
Outputs the product of all of the inputs.

*AddInt*  
Output the sum of the inputs. If *saturation* is set to yes, the output will saturate.

*SubInt*  
Outputs the "pos" input minus all of the "neg" inputs.

*MpyInt*  
Outputs the product of all of the inputs.

*GainInt*  
The output is set the input multiplied by an integer *gain* term.

*DivByInt*  
This is an amplifier. The integer output is the integer input divided by the integer *divisor* (default 2). Truncated integer division is used.

*MpyRx*  
Multiply any number of rectangular complex inputs, producing an output.

*MpyShift*  
Multiply and shift.

*Neg*  
Output the negation of the input.

*Shifter*  
Scale by shifting left *leftShifts* bits. Negative values of *leftShifts* implies right shifting.

15.2.4 Nonlinear primitives  
The nonlinear palette in the CG56 domain includes transcendental functions, quantizers, table lookup primitives, and miscellaneous nonlinear functions.

*Abs*  
Output the absolute value of the input.

*ACos*  
Output the inverse cosine of the input, which is in the range $[-1.0, 1.0]$. The output, in the principle range of 0 to $\pi$, is scaled down by $\pi$.

*ASin*  
Output the inverse sine of the input, which is in the range $[-1.0, 1.0]$. The output, in the principle range of $[-\frac{\pi}{2}, \frac{\pi}{2}]$ is scaled down by $\pi$.

*Cos*  
Output the cosine, calculated the table lookup. The input range is $[-1, 1]$ scaled by $\pi$.

*expjx*  
Output the complex exponential of the input.

*Intgrtr*  
An integrator with leakage set by *feedbackGain*. If there is an overflow, the *onOverflow* parameter will designate a wrap around, saturate or reset operation.

*Limit*  
Limits the input between the range of $[bottom, top]$.

*Log*  
Outputs the base two logarithm.

*MaxMin*  
Output the maximal or minimal (MAX) sample out of the last $N$ input samples. This can either *compareMagnitude* or take into account the sign. If *outputMagnitude* is YES the magnitude of the result is written to the output, otherwise the result itself is written.

*ModuloInt*  
Output the remainder after dividing the integer input by the integer *modulo* parameter.

*OrderTwoInt*  
Takes two inputs and outputs the greater and lesser of the two integers.

*Quant*  
Quantizes the input to one of $N + 1$ possible output levels using $N$
| **QuantIdx** | The primitive quantizes the input to one of $N+1$ possible output levels using $N$ thresholds. It also outputs the index of the quantization level used. |
| **QuantRange** | Quantizes the input to one of $N + 1$ possible output levels using $N$ thresholds. |
| **Reciprocal** | Outputs the reciprocal to $Nf$ precision in terms of a fraction and some left shifts. |
| **Sgn** | Outputs the sign of the input. |
| **SgnInt** | Outputs the sign of the integer input. |
| **Sin** | Outputs the sine, calculated using a table lookup. The input range is [-1,1) scaled by $\pi$. |
| **Sinc** | Outputs the sinc functions calculated as $\sin(x)/x$. |
| **Sqrt** | Outputs the square root of the input. |
| **Table** | Implements a real-valued lookup table. The values parameter contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received. |
| **TableInt** | Implements an integer-valued lookup table. The values parameter contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received. |
| **Expr** | General expression evaluation. |
| **LookupTbl** | The input accesses a lookup table. The interpolation parameter determines the output for input values between table-entry points. If interpolation is ”linear” the primitive will interpolate between table entries; if interpolation is set to ”none”, it will use the next lowest entry. |
| **Pulse** | Generates a variable length pulse. A pulse begins when a non-zero trigger is received. The pulse duration varies between 1 and maxDuration as the control varies between [-1,1). |
| **QntBtsInt** | Outputs the two’s complement number given by the top noBits of the input (for integer output). |
| **QntBtsLin** | Outputs the two’s complement number given by the top noBits of the input, but an optional offset can be added to shift the output levels up or down. |
| **Skew** | Generic skewing primitive. |
| **Sqr** | Outputs the square of the input. |
| **VarQuasar** | A sequence of values(data) is repeated at the output with period $N$ (integer input), zero-padding or truncating the sequence to $N$ if necessary. A value of 0 for $N$ yields an aperiodic sequence. |
Xor  Output the bit-wise exclusive-or of the inputs.

27.4.2.4 Logic primitives

Test  Test to see if two inputs are equal, not equal, greater than, and greater than or equal. For less than and less than or equal, switch the order of the inputs.

And  True if all inputs are non-zero.

Nand  True if all inputs are not non-zero.

Or  True if any input is non-zero.

Nor  True if any input is zero.

Xor  True if an odd number of inputs is non-zero.

Xnor  True if an even number of inputs is not non-zero.

Not  Logical inverter.

27.4.2.5 Control primitives

Control primitives manipulate the flow of tokens. All of these primitives are polymorphic; they operate on any data type. The SDF equivalent primitives are: Fork, DownSample, Commutator, Distributor, Mux, Repeat, Reverse, and UpSample.

ChopVarOffset  This primitive has the same functionality as the Chop primitive except now the offset parameter is determined at run time through a control input.

Cut  On each execution, this primitive reads a block of nread samples (default 128) and writes nwrite of these samples (default 64), skipping the first offset samples (default 0). It is an error if nwrite + offset > nread. If nwrite > nread, then the output consists of overlapping windows, and hence offset must be negative.

Delay  A delay primitive of parameter totalDelay unit delays.

Pad  On each execution, Pad reads a block of nread samples and writes a block of nwrite samples. The first offset samples have value fill, the next nread output samples have values taken from the inputs, and the last nwrite - nread - offset samples have value fill again.

Rotate  The primitive reads in an input block of a certain length and performs a circular shift of the input. If the rotation is positive, the input is shifted to the left so that output[0]=input[rotation]. If the rotation is negative, the input is shifted to the right so that output[rotation]=input[0].

sampleNholdModule  This sample-and-hold module is more memory efficient than using a downsample primitive for the same purpose.
27 Code Generation Domains - unsupported

VarDelay
A variable delay that will vary between 0 and maxDelay as the control input varies between -1.0 and 1.0.

WasteCycles
Stalls the flow of data for cyclesToWaste number of cycles.

27.4.2.6 Conversion primitives

This library contains primitives for format conversions from fixed point to complex fixed point. The complex data type is only partially implemented in CG56. Complex ports can be connected only to complex ports. Anytype ports can only be connected to fixed and integer ports.

CxToRect
Output the real part and imaginary part of the input of separate output ports.

RectToCx
Output a complex signal with real and imaginary part inputs.

BitsToInt
Convert a stream of bits to an integer.

IntToBits
Convert an integer into a stream of bits.

FixToCx
Convert fixed-point numbers to complex fixed-point numbers.

FixToInt
Convert fixed-point numbers to complex fixed-point numbers.

CxToFix
Convert fixed-point numbers to complex fixed-point numbers.

CxToInt
Convert fixed-point numbers to complex fixed-point numbers.

IntToFix
Convert fixed-point numbers to complex fixed-point numbers.

27.4.2.7 Signal processing primitives

This library contains primitives which perform digital signal processing functions. Descriptions of the filter primitives follow. The Goertzel and IIR primitives are identical to their SDF counterparts.

Allpass
An all-pass filter with one pole and one zero. The location of these is given by the “polezero” input.

Biquad
A two-pole, two-zero IIR filter (a biquad).

\[ H(z) = \frac{1 + n_1z^{-1} + n_2z^{-2}}{1 + d_1z^{-1} + d_2z^{-2}} \]

Comb
A comb filter with a one-pole low-pass filter in the delay loop.

BiquadDSPlay
A two-pole, two zero IIR filter (a biquad). This biquad is tailored to use the coefficients from the DSPlay filter design tool. If DSPlay gives the coefficients: A B C D E then define the parameters as follows: a = A, b = B, c = C, d = -(D + 1), e = -E. This Filter only works if a, b, c, d, and e, are in the range [-1,1). The default coefficients implement a low pass filter.

\[ H(z) = \frac{a + b \cdot z^{-1} + c \cdot z^{-2}}{1 + (d + 1)z^{-1} - e \cdot z^{-2}} \]
27.4 CG56 Domain

FIR  A finite impulse response (FIR) filter. Coefficients are specified by the taps parameter. The default coefficients give an 8th order, linear-phase, low-pass filter. To read coefficients from a file, replace the default coefficients with `<filename>`, preferably specifying a complete path. Polyphase multirate filtering is also supported.

LMS  An adaptive filter using the LMS adaptation algorithm. The initial coefficients are given by the coef parameter. The default initial coefficients give an 8th order, linear phase low-pass filter. To read default coefficients from a file, replace the default coefficients with `<filename>`, preferably specifying a complete path. This primitive supports decimation, but not interpolation.

LMSGanged  An LMS filter were the coefficients from the adaptive filter are used to run an FIR filter in parallel. The initial coefficients default to a low-pass filter of order 8.

LMSRx  A Complex LMS filter

RaisedCos  A FIR filter with a magnitude frequency response shaped like the standard raised cosine used in digital communications. See the SDFRaisedCosine primitive for more information.

The spectral estimation primitives follow. The GoertzelDetector, GoertzelPower, and LMSOscDet are identical to their SDF counterparts.

FFTCx  Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter order (default 8) is the transform size. The parameter direction (default 1) is 1 for forward, -1 for the inverse FFT.

Window  Generate standard window functions or periodic repetitions of standard window functions. The possible functions are Rectangle, Bartlett, Hanning, Hamming, Blackman, Steep-Blackman, and Kaiser. One period of samples is produced on each firing.

The communications primitives are exactly like their SDF counterparts.

27.4.3 An overview of CG56 Demos

A set of CG56 demonstration programs have been developed. The demos are grouped by the CG56 target on which they are implemented. If you do not have the require compiler, simulator, or DSP card, then you can still run the demos to see the generated code. To do this make sure that the run and compile target parameters are to NO. By default, the generated code is written to $HOME/MLD_SYSTEMS directory.

NOTE: Since the CG56 domain is not supported, the demos mentioned here are not directly accessible through demo palettes, although they are included in the MLDesigner package suite. To access the demos, use the file view tree and look in $MLD/MLD_Experimental/CG56/demo.
Please remember that the demos are not tested and most likely unable to run properly!

### 27.4.3.1 Basic/Test demos

The Basic/Test palette contains six demonstrations.

- **goertzelTest**: Test the Goertzel filters for computing the discrete Fourier transform.
- **iirTest**: Test the infinite impulse response.
- **logicTest**: Test various comparison tests and Boolean functions.
- **miscIntOps**: Test integer arithmetic operations.
- **multiFork**: Test the AnyAsmFork primitive. An AnyAsmFork primitive is one of a group of primitives that do produce any code at compile time.
- **testPostTest**: Test the DTMFPostTest primitive used in touch tone decoding.

### 27.4.3.2 Motorola Simulator Demos

The demos will generate stand alone applications. These applications will consist of: a shell script to control the simulator and output display programs; a simulator command file; and the assembled code to run on the simulator. The simulator can be run in either an interactive mode or in the background by setting the `interactive` target parameter.

- **chirp**: This system uses two integrators and a cosine to generate a chirp signal.
- **DTMFCodec**: Demonstration of touch tone detection using the discrete Fourier transform implemented by using Goertzel filters.
- **lms**: A noise source is connected to an eighth-order least-mean squares (LMS) adaptive filter with initial taps specifying a low-pass filter. The taps adapt to a null filter (the impulse response is an impulse) and the error signal is displayed.
- **lmsDTMFCodec**: Demonstration of touch tone detection using Normalized Direct Frequency Estimation implemented by using Least-Mean Squares (LMS) adaptive filters.
- **phoneLine**: A telephone channel simulator. A tone is passed through some processing which implements various distortions on a telephone channel. The parameters that are controllable are: noise, channel filter, second harmonic, third harmonic, frequency offset, phase jitter frequency, and phase jitter amplitude.
- **sin**: A sine wave is generated by using two integrators in a feedback loop.
- **transmitter**: A simple 4-level PAM transmitter.
- **tune**: A tune is generate using FM synthesis of notes stored in a table. The sounds produced are not particularly musically appealing, partly because the modulation index is not variable and the attack and decay profiles are too limited.
**27.4.3.3 S-56X Demos**

The demos require an Ariel S-56X DSP board to be installed in the workstation. In addition, all but the first demo requires QDM. These demos generate a stand alone application consisting of: a shell script to download and run the assembled code; a file specifying the asynchronous user I/O interface; and the assembled code.

- **ADPCM**
  This demo implements an ADPCM coder and decoder. The user at run time can vary the number of quantization bits, the quantization range, and a delay so that signal can be heard instantaneously or a second later. Requires an Ariel Proport and a microphone.

- **amtx**
  Amplitude Modulation Transmitter. The results of the transmitter are displayed asynchronously at run time.

- **CD Volume**
  A system implementing a volume control with CG56HostSliderGX primitives. Requires a modified CD player.

- **echoCanceling**
  A system implementing a pair of echo cancellation filters. The first echo cancellation filter cancels an artificial echo introduced by an FIR filter. The second echo cancellation filter is used to cancel the echoes produced by have one microphone next to loud speaker. Another microphone is used for desired input, such as speech. Requires an Ariel Proport and two microphones.

- **recv-2psk**
  2-PSK Bandpass filter.

- **reverb**
  This system implements a reverberation system using Comb filters. Requires an Ariel Proport and a microphone.

- **xmit-2psk**
  2-PSK transmitter.

**27.4.3.4 CGC-S56X Demos**

All of the demos in this palette use the `CompileCGSubsystems` target.

**Stand alone Application Demos**

The first set demos generate stand alone applications consisting of two parts: a program generate in C that implements the sub-graph that runs on the host, and a program generated in Motorola 56k assembly that is to be run on the S-56X. The C program initializes and downloads the S-56X program automatically. The first four of the demos, lms, phoneLine, DTMFCodec and lmsDTMFCCodec are identical to the simulator demos.

- **Modem**
  The modem palette contain 3 phased shift keying modem demos. These demos illustrate the use of peek/poke actors and hierarchical scheduling. Requires an Ariel Proport and a microphone.
dtmfSpectrum  
This demos implements a DTMF tone generator and displays the resultant frequency spectrum.

synth  
An FM music synthesis demonstration. Requires an Ariel Proport.

synthFFT  
An FM music synthesis demonstration showing the resultant frequency spectrum. Requires an Ariel Proport.

PRfilterBank  
A perfect reconstruction filter bank.

ADPCM  
This demo implements an ADPCM coder and decoder. The user at run time can vary the number of quantization bits, the quantization range, and a delay so that signal can be heard instantaneously or a second later. Requires an Ariel Proport and a microphone.

Simulation SDF-Wormhole Demos

The simulation SDF wormhole demos create simulation SDF primitives in ptlang and also a load file for the S-56X card. Unlike the other CG56 demos, the applications produced here will not run as stand alone applications. The wormhole allows the user to imbed a CG56 system running on an Ariel S-56X DSP board into an MLDesigner simulation.

MultiTone  
Generates three sine waves on the S-56X which are at different rates relative to one another.

DSPWorm  
Demonstrates multirate I/O between MLDesigner and the S-56X board.

PRfilterBank  
A perfect reconstruction filter bank.

27.4.4 Targets

Seven CG56 targets are included in the MLDesigner distribution.

27.4.4.1 Default CG56 (default-CG56) target

The default target is used only for code generation. It has the following set of options:

host  
(STRING) Default =
The default is the empty string. Host machine to compile or assemble code on. All code is written to and compiled and run on the computer specified by this parameter. If a remote computer is specified here then rsh commands are used to place files on that computer and to invoke the compiler. You should verify that your .rhosts file is properly configured so that rsh will work.

directory  
(STRING) Default = $HOME/MLD_SYSTEMS
This is the directory to which all generated files will be written to.

file  
(STRING) Default =
The default is the empty string. This represents the prefix for filenames for all generated files.

Looping Level  
Specifies if the loop scheduler should be used.
27.4 CG56 Domain

display?     (INT) Default = YES
If this flag is set to YES, then the generated code will be displayed on
the screen.

compile?     This is a dummy flag since the default target only generates code.
run?         This is a dummy flag since the default target only generates code.

xMemMap      (STRING) Default = 0-4095
Valid x memory address locations. Default is 0-4095, which means
x:0 through x:4095 are valid memory addresses. Disjoint segments
of memory can be specified by separating the contiguous ranges with
spaces, e.g. "0-4095 5000-5500."

yMemMap      (STRING) Default = 0-4095
Valid y memory address locations. Default is 0-4095, which means
y:0 through y:4095 are valid memory addresses.

subroutines? (INT) Default = -1
Setting this parameter to N makes the target attempt to generate a sub-
routine instead of in-line code for a primitive if the number of repeti-
tions of that primitive is greater than N (use N = 0 to generate subrou-
tines even for primitives with just 1 repetition). Set subroutines? to
-1 (or any other negative integer) to disable the feature.

show memory usage?  (INT) Default = NO
If YES, then the target will report the actual amount of program, x
data memory, and y data memory used by the program in words.

27.4.4.2 CG56 Simulator (sim-CG56) target

This target is used for generating DSP56000 assembly code, assembling it, and running it on a Mo-
torola DSP56000 simulator. For this to work properly, the Motorola 56000 assembler (asm56000)
and the simulator (sim56000) must be in the user path. Otherwise a run on this target produces
code only, and an error message will appear indicating the absence of the required programs in the
user path. Input and output files specified in ReadFile and WriteFile primitives are passed
on to the simulator by an automatically generated system.cmd file, which is sourced by the
simulator.

The options for this target are mostly the same as the ones for default-CG56 above, except for
the following:

compile?     (INT) Default = YES
If this option is set to YES, then generated code is assembled using the
asm56000 program.

run?         (INT) Default = YES
If YES, then the assembled code is run on the Motorola simulator
sim56000.

Interactive Sim.  (INT) Default = YES
If YES the simulator is run interactively (in which case one can add
breakpoints, single step through code, etc.)

27.4.4.3 Ariel S-56X (S-56X) target

This target generates stand alone applications that will run on the Ariel S-56X DSP board. An optional graphical debugger, QDM, is available from the board designer, Mike Peck. This debugger is needed for some of the user I/O primitives that are specific to this target.

The options for this target are mostly the same as the ones for default-CG56, except for the following:

```
monitor (STRING) Default =
```

The default is the empty string. This parameter specifies an optional monitor of debugger for use with the S-56X target. If the application has QDM primitives, this parameter should be set to `qdmterm_s56x -run`.

27.4.4.4 CG56 Subroutine (sub-CG56) target

This target is used to generate subroutines that can be called from hand-written 56000 code. The options are identical to those of default-CG56 target.

27.5 C50 Domain

27.5.1 Introduction

The C50 domain generates assembly code for the Texas Instruments TMS320C5x series of digital signal processors. The graphs that we can describe in this domain follow the synchronous data flow (SDF) model of computation. SDF allows us to schedule the Blocks and allocate all the resources at compile time.

The TMS320C5x series are fixed-point digital signal processors which have 16 bit data and instructions and operate at a maximum rate of 50MIPS.

Since the C5x processors are fixed point, the floating point data type has no meaning in the C50 domain. Fixed-point values can take on the range \([-1, 1]\). The most positive value is \(1 - 2^{-15}\). The domain defines a new constant `C50_ONE` set to this maximum positive value. In this chapter, whenever data types are not mentioned, fixed-point is meant. The complex data type means a pair of fixed-point numbers. The complex data type is supported for primitives that have anytype inputs or outputs. Integers are the same length as the fixed-point representation. Matrix data types are not supported yet.

27.5.2 An overview of C50 primitives

For the C50 domain, the primitive library is large enough that it has been divided into sub-palettes as was done with the SDF main palette.
The sub-palettes are Signal Sources, I/O, Arithmetic, Nonlinear Functions, Logic, Control, Conversion, Signal Processing, and Higher Order Functions. The primitives on the Higher Order Functions (HOF) palette are used to help lay out schematics graphically. The HOF primitives are in the HOF domain, and not the C50 domain. Each palette is summarized in more detail below.

At the top of each palette, for convenience, are instances of the delay icon, the bus icon, and the following primitive:

**BlackHole** 
Discard all inputs. This primitive is useful for discarding signals that are not useful.

### 27.5.3 Source primitives

Source primitives are primitives with only outputs. They generate signals, and may represent external inputs to the system, constant data, or synthesized stimuli. The primitives with SDF equivalents are: Const, ConstCx, ConstInt, Ramp, RampInt, Rect, singen, and WaveForm.

- **Impulse** 
  Generate a single impulse or an impulse train. The size is determined by `impulseSize` (default ONE). If `period` (default is 0) is positive, an impulse train with this period is generated, otherwise a single impulse is generated. If `delay` (default 0) is positive, the impulse (or impulse train) is delayed by this amount.

- **IIDUniform**
  Generate an IID uniformly distributed pseudo-random process. Output is uniformly distributed between -`range` and `range` (default ONE).

- **IIDGaussian**
  Generate a white Gaussian pseudo-random process with mean 0 and standard deviation 0.1. A Gaussian distribution is realized by summing `noUniforms` (default 16) number of uniform random variables. According to the central limit theorem, the sum of $N$ random variables approaches a Gaussian distribution as $N$ approaches infinity.

- **Tone**
  Generate a sine or cosine wave using a second order oscillator. The wave will be of amplitude (default 0.5), frequency (default 0.2), and `calcType` (default "sin")

### 27.5.3.1 I/O Primitives

I/O primitives are target specific primitives that allow input and output of stimuli to a target architecture. Currently there are I/O primitives only for the C50 DSK board so these primitives should only be used with the DSKC50 target. These primitives are located on the TI 320C5x IO palette inside the Input/Output palette (not shown here).

- **AIn**
  This is an interrupt driven primitive to receive samples from the A/D converter in the Analog Interface Chip. The sample rate is determined by `sampleRate`. The actual conversion rate is 285.7KHz/$N$ where $N$ is an integer from 4 to 64. This primitive supports an internal buffer
to hold the received samples. The size of this buffer can be set manually by changing the `interruptBufferSize` parameter. Setting `interruptBufferSize` to a negative value will set the size of the buffer equal to the number of times the primitive is fired on each iteration of the system.

**AOut**

This is an interrupt driven primitive to send samples to the D/A converter in the AIC chip. The parameters are identical to those of the `AIn` primitive.

### 27.5.3.2 Arithmetic primitives

The arithmetic primitives that are available are:

- **Add**
  
  Output the sum of the inputs. If `saturation` is set to yes, the output will saturate.

- **Sub**
  
  Outputs the "pos" input minus all of the "neg" inputs.

- **Mpy**
  
  Outputs the product of all of the inputs.

- **Gain**
  
  The output is set the input multiplied by a gain term. The gain must be in $[-1, 1]$.

- **AddCx**
  
  Output the complex sum of the inputs. If `saturation` is set to yes, the output will saturate.

- **SubCx**
  
  Outputs the "pos" input minus all of the "neg" inputs.

- **MpyCx**
  
  Outputs the product of all of the inputs.

- **AddInt**
  
  Output the sum of the inputs. If `saturation` is set to yes, the output will saturate.

- **SubInt**
  
  Outputs the "pos" input minus all of the "neg" inputs.

- **MpyInt**
  
  Outputs the product of all of the inputs.

- **GainInt**
  
  The output is set the input multiplied by an integer gain term.

- **DivByInt**
  
  This is an amplifier. The integer output is the integer input divided by the integer divisor (default 2). Truncated integer division is used.

- **MpyShift**
  
  Multiply and shift.

- **Neg**
  
  Output the negation of the input.

- **Shifter**
  
  Scale by shifting left `leftShifts` bits. Negative values of `leftShifts` implies right shifting.

### 27.5.3.3 Nonlinear primitives

The nonlinear palette in the C50 domain includes transcendental functions, quantizers, table lookup primitives, and miscellaneous nonlinear functions.

- **Abs**
  
  Output the absolute value of the input.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACos</td>
<td>Output the inverse cosine of the input, which is in the range -1.0 to 1.0. The output, in the principle range of 0 to π, is scaled down by π.</td>
</tr>
<tr>
<td>ASin</td>
<td>Output the inverse sine of the input, which is in the range -1.0 to 1.0. The output, in the principle range of [-π/2, π/2] is scaled down by π.</td>
</tr>
<tr>
<td>Cos</td>
<td>Output the cosine, calculated the table lookup. The input range is [-1, 1] scaled by π.</td>
</tr>
<tr>
<td>expjx</td>
<td>Output the complex exponential of the input.</td>
</tr>
<tr>
<td>Integrtr</td>
<td>An integrator with leakage set by feedbackGain. If there is an overflow, the onOverflow parameter will designate a wrap around, saturate or reset operation.</td>
</tr>
<tr>
<td>Limit</td>
<td>Limits the input between the range of [bottom, top].</td>
</tr>
<tr>
<td>Log</td>
<td>Outputs the base two logarithm.</td>
</tr>
<tr>
<td>MaxMin</td>
<td>Output the maximal or minimal (MAX) sample out of the last N input samples. This can either compareMagnitude or take into account the sign. If outputMagnitude is YES the magnitude of the result is written to the output, otherwise the result itself is written.</td>
</tr>
<tr>
<td>ModuloInt</td>
<td>Output the remainder after dividing the integer input by the integer modulo parameter.</td>
</tr>
<tr>
<td>OrderTwoInt</td>
<td>Takes two inputs and outputs the greater and lesser of the two integers.</td>
</tr>
<tr>
<td>Quant</td>
<td>Quantizes the input to one of N + 1 possible output levels using N thresholds.</td>
</tr>
<tr>
<td>QuantIdx</td>
<td>The primitive quantizes the input to one of N+1 possible output levels using N thresholds. It also outputs the index of the quantization level used.</td>
</tr>
<tr>
<td>QuantRange</td>
<td>Quantizes the input to one of N + 1 possible output levels using N thresholds.</td>
</tr>
<tr>
<td>Reciprocal</td>
<td>Outputs the reciprocal to Nf precision in terms of a fraction and some left shifts.</td>
</tr>
<tr>
<td>Sgn</td>
<td>Outputs the sign of the input.</td>
</tr>
<tr>
<td>SgnInt</td>
<td>Outputs the sign of the integer input.</td>
</tr>
<tr>
<td>Sin</td>
<td>Outputs the sine, calculated using a table lookup. The input range is [-1, 1) scaled by π.</td>
</tr>
<tr>
<td>Sinc</td>
<td>Outputs the sinc functions calculated as sin(x)/x.</td>
</tr>
<tr>
<td>Sqrt</td>
<td>Outputs the square root of the input.</td>
</tr>
<tr>
<td>Table</td>
<td>Implements a real-valued lookup table. The values parameter contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received.</td>
</tr>
<tr>
<td>TableInt</td>
<td>Implements an integer-valued lookup table. The values parameter</td>
</tr>
</tbody>
</table>
contains the values to output; its first element is element zero. An error occurs if an out of bounds value is received.

**Expr**
General expression evaluation.

**LookupTbl**
The input accesses a lookup table. The interpolation parameter determines the output for input values between table-entry points. If interpolation is "linear" the primitive will interpolate between table entries; if interpolation is set to "none", it will use the next lowest entry.

**Pulse**
Generates a variable length pulse. A pulse begins when a nonzero trigger is received. The pulse duration varies between 1 and maxDuration as the control varies between \([-1, 1]\).

**QntBtsInt**
Outputs the two’s complement number given by the top noBits of the input (for integer output).

**QntBtsLin**
Outputs the two’s complement number given by the top noBits of the input, but an optional offset can be added to shift the output levels up or down.

**Skew**
Generic skewing primitive.

**Sqr**
Outputs the square of the input.

**VarQuasar**
A sequence of values(data) is repeated at the output with period \(N\) (integer input), zero-padding or truncating the sequence to \(N\) if necessary. A value of 0 for \(N\) yields an aperiodic sequence.

**Xor**
Output the bit-wise exclusive-or of the inputs.

### 27.5.3.4 Logic primitives

**Test**
Test to see if two inputs are equal, not equal, greater than, and greater than or equal. For less than and less than or equal, switch the order of the inputs.

**And**
True if all inputs are non-zero.

**Nand**
True if all inputs are not non-zero.

**Or**
True if any input is non-zero.

**Nor**
True if any input is zero.

**Xor**
True if its inputs differ in value.

**Xnor**
True if its inputs coincide in value.

**Not**
Logical inverter.

### 27.5.3.5 Control primitives

Control primitives manipulate the flow of tokens. All of these primitives are polymorphic; they operate on any data type. The SDF equivalent primitives are: **Fork**, **DownSample**, **Commutator**, **Distributor**, **Mux**, **Repeat**, **Reverse**, and **UpSample**.
ChopVarOffset  
This primitive has the same functionality as the Chop primitive except now the offset parameter is determined at run time through a control input.

Cut  
On each execution, this primitive reads a block of nread samples (default 128) and writes nwrite of these samples (default 64), skipping the first offset samples (default 0). It is an error if nwrite + offset > nread. If nwrite > nread, then the output consists of overlapping windows, and hence offset must be negative.

Delay  
A delay primitive of parameter totalDelay unit delays.

Pad  
On each execution, Pad reads a block of nread samples and writes a block of nwrite samples. The first offset samples have value fill, the next nread output samples have values taken from the inputs, and the last nwrite − nread − offset samples have value fill again.

Rotate  
The primitive reads in an input block of a certain length and performs a circular shift of the input. If rotation is positive, the input is shifted to the left so that output[0] = input[rotation]. If the rotation is negative, the input is shifted to the right so that output[rotation] = input[0].

sampleNholdModule  
This sample-and-hold module is more memory efficient than using a downsample primitive for the same purpose.

VarDelay  
A variable delay that will vary between 0 and maxDelay as the control input varies between -1.0 and 1.0.

WasteCycles  
Stalls the flow of data for cyclesToWaste number of cycles.

### 27.5.3.6 Conversion primitives

This library contains primitives which perform format conversions from fixed point to complex fixed point.

**CxToRect**  
Output the real part and imaginary part of the input of separate output ports.

**RectToCx**  
Output a complex signal with real and imaginary part inputs.

**BitsToInt**  
Convert a stream of bits to an integer.

**IntToBits**  
Convert an integer into a stream of bits.

**FixToCx**  
Convert fixed-point numbers to complex fixed-point numbers.

**FixToInt**  
Convert fixed-point numbers to integer numbers.

**CxToFix**  
Output the magnitude squared of the complex number.

**CxToInt**  
Output the magnitude squared of the complex number.

**IntToFix**  
Convert an integer input to a fixed point output.

**IntToCx**  
Convert an integer input to a complex output.
27.5.3.7 DSP (Digital Signal Processing)

The primitives in this library perform digital signal processing functions. The Goertzel and IIR primitives are identical to their SDF counterparts.

**Allpass**
An all-pass filter with one pole and one zero. The location of these is given by the "polezero" input.

**Biquad**
A two-pole, two-zero IIR filter (a biquad).

\[
H(z) = \frac{1 + n_1 z^{-1} + n_2 z^{-2}}{1 + d_1 z^{-1} + d_2 z^{-2}}
\]

**Comb**
A comb filter with a one-pole low-pass filter in the delay loop.

**BiquadDSPlay**
A two-pole, two zero IIR filter (a biquad). This biquad is tailored to use the coefficients from the DSPlay filter design tool. If DSPlay gives the coefficients: \(A\ B\ C\ D\ E\) then define the parameters as follows: \(a = A, b = B, c = C, d = -(D + 1), e = -E\). This only works if \(a, b, c, d,\) and \(e,\) are in the range \([-1, 1)\). The default coefficients implement a low pass filter.

\[
H(z) = \frac{a + b \cdot z^{-1} + c \cdot z^{-2}}{1 - (d + 1) z^{-1} - e \cdot z^{-2}}
\]

**FIR**
A finite impulse response (FIR) filter. Coefficients are specified by the taps parameter. The default coefficients give an 8th order, linear-phase, low-pass filter. To read coefficients from a file, replace the default coefficients with < filename, preferably specifying a complete path. Polyphase multirate filtering is not yet supported.

**LMS**
An adaptive filter using the LMS adaptation algorithm. The initial coefficients are given by the coef parameter. The default initial coefficients give an 8th order, linear phase low-pass filter. To read default coefficients from a file, replace the default coefficients with < filename, preferably specifying a complete path. This primitive supports decimation, but not interpolation.

**LMSGanged**
An LMS filter were the coefficients from the adaptive filter are used to run an FIR filter in parallel. The initial coefficients default to a low-pass filter of order 8.

**RaisedCos**
An FIR filter with a magnitude frequency response shaped like the standard raised cosine used in digital communications. See the SDFRaisedCosine primitive for more information.

The spectral estimation primitives follow. The GoertzelDetector, GoertzelPower, and LMSOscDet are identical to their SDF counterparts.

**FFTCx**
Compute the discrete-time Fourier transform of a complex input using the fast Fourier transform (FFT) algorithm. The parameter order (default 8) is the transform size. The parameter direction (default 1) is 1 for forward, -1 for the inverse FFT.
Window

Generate standard window functions or periodic repetitions of standard window functions. The possible functions are Rectangle, Bartlett, Hanning, Hamming, Blackman, Steep-Blackman, and Kaiser. One period of samples is produced on each firing.

The communications primitives are exactly like their SDF counterparts.

### 27.5.4 An overview of C50 Demos

A set of C50 demonstration programs have been developed. The demos are meant to be run on the C50DSK board. If you do not have the required DSK tools, then you can still run the demos to see the generated code. To do this make sure that the run and compile target parameters are to NO. By default, the generated code is written to the `$HOME/MLD_SYSTEMS/C50` directory.

**NOTE:** Since the C50 domain is not supported, the demos mentioned here are not directly accessible through demo palettes, although they are included in the MLDesigner package suite. To access the demos, use the file view tree and look in `$MLD/MLD_Experimentals/C50/demo`. Please remember that the demos are not tested and most likely unable to run properly!

#### 27.5.4.1 Basic/Test demos

The Basic/Test palette contains 7 demonstrations.

- **goertzelTest**: Test the Goertzel filters for computing the discrete Fourier transform.
- **firTest**: Test the finite impulse response (FIR) filters.
- **iirTest**: Test the infinite impulse response (IIR) filters.
- **logicTest**: Test various comparison tests and Boolean functions.
- **miscIntOps**: Test integer arithmetic operations.
- **multiFork**: Test the `AnyAsmFork` primitive. An `AnyAsmFork` primitive is one of a group of primitives that do produce any code at compile time.
- **testPostTest**: Test the `DTMFPrefTest` primitive used in touch tone decoding.

#### 27.5.4.2 DSK 320C5x demos

The DSK 320C5x demo palette contains demonstrations meant to be run on the Texas Instruments DSP Starter Kit board.

- **chirp**: This system uses two integrators and a cosine to generate a chirp signal.
- **DTMFCCodec**: Demonstration of touch tone detection using the discrete Fourier transform implemented by using Goertzel filters.
- **lms**: A noise source is connected to an eighth-order least-mean squares (LMS) adaptive filter with initial taps specifying a low-pass filter. The
taps adapt to a null filter (the impulse response is an impulse) and the error signal is displayed.

lmsDTMFCodec Demonstration of touch tone detection using Normalized Direct Frequency Estimation implemented by using Least-Mean Squares (LMS) adaptive filters.

phoneLine A telephone channel simulator. A tone is passed through some processing which implements various distortions on a telephone channel. The parameters that are controllable are: noise, channel filter, second harmonic, third harmonic, frequency offset, phase jitter frequency, and phase jitter amplitude.

sin A sine wave is generated by using two integrators in a feedback loop.

transmitter A simple 4-level PAM transmitter

27.5.5 Targets

Three C50 targets are included in the MLDesigner distribution.

27.5.5.1 Default C50 (default-C50) target

The default target is used only for code generation. It has the following set of options:

host (STRING) Default =
The default is the empty string. Host machine to compile or assemble code on. All code is written to and compiled and run on the computer specified by this parameter. If a remote computer is specified here then rsh commands are used to place files on that computer and to invoke the compiler. You should verify that your .rhosts file is properly configured so that rsh will work.

directory (STRING) Default = $HOME/MLD_SYSTEMS/C50
This is the directory to which all generated files will be written to.

file (STRING) Default =
The default is the empty string. This represents the prefix for filenames for all generated files.

Looping Level Specifies if the loop scheduler should be used.
display? (INT) Default = YES
If this flag is set to YES, then the generated code will be displayed on the screen.

compile? This is a dummy flag since the default target only generates code.
run? This is a dummy flag since the default target only generates code.

bMemMap (STRING) Default = 768-1279
Address range for C50 Dual Access RAM blocks. C50 Instructions that operate on data run faster if the data is stored in one of the DARAM
blocks. Disjoint segments of memory can be specified by separating the contiguous ranges with spaces, e.g. "768-800 1200-1279."

uMemMap (STRING) Default = 2432-6848
Data address range in the C50 Single Access RAM block. This can also specify a valid address range in external memory.

subroutines? (INT) Default = -1
Setting this parameter to \( N \) makes the target attempt to generate a subroutine instead of in-line code for a primitive if the number of repetitions of that primitive is greater than \( N \) (use \( N = 0 \) to generate subroutines even for primitives with just 1 repetition). Set subroutines? to -1 (or any other negative integer) to disable the feature.

27.5.5.2 C50 Subroutine (sub-C50) target

This target is used to generate subroutines that can be called from hand-written C50 code. The options are identical to those of default-C50 target.

27.5.5.3 C50 DSP Starter Kit (DSKC50) target

This target is used to generate C50 code to be run on Texas Instruments’ DSP Starter Kit board. In addition to the regular file.asm generated by the other targets, this target will produce a second file (fileDSK.asm) which is the same as the original file but with all lines truncated to 80 characters. This is done because the TI DSK assembler will give false error messages if lines in the input file exceed 80 characters. The options are identical to those of default-C50 target with four exceptions:

 compile? If this flag is set the target will issue the command asmc50 fileDSK.asm where fileDSK.asm is the name of the file containing the generated code. This should run the DSK assembler and produce a file fileDSK.dsk. Note that asmc50 can be a shell script that invokes the user’s DSK assembler. Scripts to use the TI DSK assembler and loader in Linux are presented at the end of this section.

 run? If this flag is set the target will issue the command loadc50 fileDSK.dsk which should load fileDSK.dsk to the DSK board. Note that loadc50 can be a shell script that invokes the user’s DSK loader.

bMemMap (STRING) Default = 768-1270
Valid addresses on the Dual Access RAM block 1. The last 9 words in this (addresses 1271 - 1279) are reserved by the target to store configuration information for the Analog Interface Chip.

uMemMap (STRING) Default = 2432-6847
Valid addresses on the Single Access RAM memory. Locations 6848 - 11263 are reserved to store the user’s program and locations 2048-2431 are reserved by the TI DSK debugger kernel.
The following scripts invoke the TI DSK assembler and loader from Linux through dosemu (a DOS emulator). Note that before invoking the assembler and loader MLDesigner executes a cd to the directory target parameter. Since you need to unmount the DOS partition to run dosemu you can not have directory set to the DOS partition. One solution is to set directory to your home directory and set file to include the path to the directory where you want the file written. For example, if your home directory is /ptuser, the dos partition dosemu will use is /dos/c and you want the output files written to /dos/c/dsk/src you could set directory to /users/ptdesign and file to /dos/c/dsk/filename where filename is the name of the output file. These scripts are also included in $MLD/src/domains/c50.

#!/bin/sh
# Version: @(##asmc501.604/07/97
# Copyright (c) 1996-1997 The Regents of the University of California.
# All Rights Reserved.
## asmc50
# script to assemble files with TI’s DSK assembler(dsk5a.exe)
# Uses dosemu to run dsk5a.exe. The person running it must be root to
# mount/unmount the dos partition.
# This script was tested on a machine running linux (red-hat 3.0.3
# distribution) with dosemu-0.63.1.33 installed.
## Written by Luis Gutierrez.
# Converted from csh to sh by Brian L. Evans

# User’s home directory.
homedir=/root

# User’s dos partition.
dospartition=/dos/c

# The root path of DOS drive where DSK files and DOS binaries are stored.
dosroot=c:

dksrc=dsk\src

dskbin=dsk

# The file used to temporarily save autoexec.emu.
autoexecsave=autoexec.bak

cd $dospartition
mv autoexec.emu $autoexecsave

# The text between the first xxxx and the second xxxx will be
# piped to unix2dos and will end up in autoexec.emu.

unix2dos > $dospartition/autoexec.emu << xxxx
path $dosroot\dskbin;$dosroot\dos
cd $dosroot\dsksrc
dsk5a.exe $1:t
exitemu
xxxx
cd $homedir

# Unmount DOS partition to run dosemu
umount $dospartition
dos > /dev/null

# Mount DOS partition after running dosemu
mount -t msdos /dev/sda1 $dospartition

# Restore autoexec.emu
cd $dospartition
mv -f $dospartition/$autoexecsave $dospartition/autoexec.emu

The following script is used to load files.

#!/bin/csh
# Version: @(loadc501.5 03/29/97
# Copyright (c) 1996-1997 The Regents of the University of California.
# All Rights Reserved.
## loadc50
# script to load files with TI’s DSK loader(dsk5l.exe)
# Uses xdos to run dsk5l.exe. The person running it must be root to
# mount/unmount the dos partition.
# This script was tested on a machine running linux(red-hat 3.0.3
# diistribution) with dosemu-0.63.1.33 installed.
# Written by Luis Gutierrez.
## Converted from csh to sh by Brian L. Evans
# User’s home directory.
homedir=/root

# User’s dos partition.
dospartition=/dos/c

dosroot=c:


# The DOS directory (relative to dosroot) where the *.asm and *.dsk files
# are stored. Replace the \ in the DOS path with \\.

dsksrc=dsk\src

# The DOS directory (relative to dosroot) where the DSK
# executables (dsk5a.exe, dsk5l.exe) are stored.
# Replace the \ in the DOS path with \\.
dskbin=dsk

# The file used to temporarily save autoexec.emu
autoexecsave=autoexec.bak

cd $dospartition
mv autoexec.emu $autoexecsave

# The text between the first xxxx and the second xxxx will be
# piped to unix2dos and will end up in autoexec.emu.

unix2dos > $dospartition/autoexec.emu << xxxx
path $dosroot\$dskbin;$dosroot\dos

cd $dosroot\$dsksrc
dsk5l.exe $1:t
exitemu
xxxx

cd $homedir

# Unmount DOS partition to run xdos
umount $dospartition
xdos

# After running xdos mount DOS partition
mount -t msdos /dev/sda1 $dospartition

# Restore autoexec.emu

cd $dospartition
mv -f $dospartition/$autoexecsave $dospartition/autoexec.emu
Part IV

Appendix
Appendix A

General

A.1 System Requirements

Following table lists general requirements which are independent on used operating system, distribution, or version.

<table>
<thead>
<tr>
<th>Category</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Server running at</td>
<td>1024 X 768 pixels minimum</td>
</tr>
<tr>
<td></td>
<td>1280 X 1024 pixels or more recommended</td>
</tr>
<tr>
<td>Hard Drive Space</td>
<td>700 MB minimum</td>
</tr>
<tr>
<td>Physical memory</td>
<td>256 MB minimum</td>
</tr>
<tr>
<td></td>
<td>512MB recommended</td>
</tr>
<tr>
<td>Swap</td>
<td>256 MB minimum</td>
</tr>
<tr>
<td></td>
<td>1 GB recommended</td>
</tr>
<tr>
<td>GNU compiler gcc</td>
<td>3.2 or later (see table below)</td>
</tr>
<tr>
<td>GNU make</td>
<td>3.8 or later</td>
</tr>
<tr>
<td>GNU tar</td>
<td>1.13 or later</td>
</tr>
<tr>
<td>GNU gzip package</td>
<td>1.3 or later</td>
</tr>
<tr>
<td>PDF Viewer</td>
<td>A PDF document viewer such as Acroread or GhostView for reading online documentation</td>
</tr>
<tr>
<td>NS2</td>
<td>2.26 or later (only needed if you want to use the NS2 domain)</td>
</tr>
<tr>
<td>MySQL</td>
<td>4.0 or later (only needed for accessing MySQL databases)</td>
</tr>
</tbody>
</table>

Table A.1: General Requirements

NOTE: The X.org or XFree86 development package including header files is required for the compilation and loading of primitives that use Qt or OpenGL code as well as for external
simulations using C++.

NOTE: GNU compiler version shown in the table above must be interpreted as common requirements. The compiler version required for a certain operating system is listed in the table below.

<table>
<thead>
<tr>
<th>Dependent</th>
<th>Compiler Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linux SuSE 9.3 (32 and 64 bit)</strong></td>
<td>gcc 3.3.5</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux openSuSE 10.2 (32 and 64 bit)</strong></td>
<td>gcc 4.1.2</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux openSuSE 10.3 (32 and 64 bit)</strong></td>
<td>gcc 4.2.1</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux openSUSE 11.0 (32 and 64 bit)</strong></td>
<td>gcc 4.3.1</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux openSuSE 11.1 (32 and 64 bit)</strong></td>
<td>gcc 4.3.2</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux openSuSE 11.2 (32 and 64 bit)</strong></td>
<td>gcc 4.4.1</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux RHEL 3 (32 bit and 64 bit)</strong></td>
<td>gcc 3.2</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
<tr>
<td><strong>Linux RHEL 4 (32 and 64 bit)</strong></td>
<td>gcc 3.4.6</td>
</tr>
<tr>
<td>GNU compiler</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Platform Dependent Requirements

NOTE: Other Linux platforms not listed in the table above may work without problems. Because of the broad range of Linux distributions and versions, we are only able to support Linux versions listed above.

NOTE: On Debian based Linux distributions packages xorg-dev, gcc, g++, libstdc++5, libstdc++6 and libc6-i386 (only on 64 bit architectures) have to be installed.

NOTE: On newer openSuSE systems package libstd++33 have to be installed in addition to standard development packages.

Gnu tools, including gmake 3.8 or later are normally part of a standard Linux installation.
A.2 Environment Variables

This section lists the standard environment variables for the MLDesigner environment.

- **$MLD**: MLDesigner installation directory
- **$MLD_HOME**: $MLD
- **$MLD_USER**: directory where user defined models reside (default is $HOME/MLD)
- **$MLD_EDITOR**: user preferred editor used for editing primitive source
- **$SIM_NO**: current simulation number
- **$SIMNO**: = $SIM_NO
- **$NO**: = $SIM_NO
- **$SIM_IT**: current iteration number
- **$SIMIT**: = $SIM_IT
- **$IT**: = $SIM_IT
- **$MLDARCH**: machine architecture, possible values are:
  - i386-linux-gcc3
  - i386-linux-gcc34
  - i386-linux-gcc4
  - x86_64-linux-gcc3
  - x86_64-linux-gcc34
  - x86_64-linux-gcc4
  - spar-sunos58
- **$HOME**: $MLD
- **$MLD_SHARED**: Shared library for workgroups
- **$MLD_RSH**: has to be set to ssh if you want to use ssh for distributed simulation
- **$MYSQL_DEV_HEADERS**: directory where the MySQL development headers reside
  - (default is /usr/include/mysql)
- **$MYSQL_LIB_PATH**: directory containing the MySQL shared library
  - (default is /usr/lib)

A.3 Valid File Names

If you have file naming conventions where you use the underscore, you need to define where the separator is as MLDesigner interprets an underscore as an environment variable separator. To ensure that the variables are properly interpreted you can use the dollar ($) sign and curly braces. The following formats are supported by MLDesigner:

- \$XYZ/...
- \$XYZ_123/...
A.4 Uninstall MLDesigner

NOTE: You do not need to uninstall MLDesigner in order to update to a later release. All files and folders in the $MLD directory are erased when you install the new version of the program. All user directories and libraries are not erased.

In some cases it may be necessary to remove MLDesigner before updating to a newer version. This is especially so where hard disk space is limited. In such cases proceed as follows:

• Log in as root.
• Execute the uninstall script by typing mld2uninstall.
• Confirm that you wish to remove all files by typing yes.
• License files are moved to the home directory of the root user (this is normally the /root directory). All files stored in the MLDesigner program directory are completely erased. This may take some time.
• Lastly the call scripts from /usr/bin or /usr/local/bin are removed.

You have successfully removed MLDesigner from your system.

Please remember that any locally created libraries, systems, modules, primitives and their sources should reside in your home directory and were not erased by this procedure.

A.5 Version Update Warning

NOTE: Make a backup of all your libraries before starting a new MLDesigner version for the first time!

When you open an existing model or model library in MLDesigner version 2.7 or below for the first time, a version update is executed. This is necessary because model libraries shipped with MLDesigner as well as internal mechanisms may change from version to version. Extensive testing has been done to ensure that existing models are not adversely effected after updating, but we recommend you make a backup of all your existing libraries because there is no way to revert your models back to the previous version if problems arise. If for any reason problems are experienced after updating your models, contact support using the form at:

http://www.mldesigner.com/support.php

NOTE: The models and libraries to be updated must be writable.
When opening a library or model that needs a version update or contains models that need a version update, a warning dialog displays to inform about the necessity of a version update and to recommend that a backup be made before continuing.

![Version Update Warning](image)

Figure A.1: Version Update Warning

Upgrading to the latest version is FREE of charge and is recommended if you are to enjoy the new features and improved performance of the latest release.
Appendix B

Support

B.1 How to Contact Us

For technical support use our online support form at:

http://www.mldesigner.com/support.php

NOTE: We are unable to respond to support requests unless they are sent via the support web interface.

B.2 Reporting Problems/Bugs

Please refer to app. E and the Frequently Asked Questions document before contacting the MLDesigner support group. The Help button in the top right corner has a menu option Search Index which takes you directly to the MLDesigner manual index. Here you find hundreds of references to topics that may help answer your question.

B.3 Viewing the Online Documentation

You need a PDF viewer to view the online documentation. The first time you try to open an online document you are prompted to define a PDF viewer. This is done via the Settings option on the main menu bar. Type in the name of the PDF viewer you wish to use if it is installed in your search path. If not, use the file selection dialog to find the PDF viewer or type in the full path if you know where it is located.

If your question remains unanswered send your query including the following:

- Which Linux or Solaris system do you use? (Vendor, version and kernel if available)
- Is your system connected to a network?

Please also specify where you experienced the problem. Was it:

- during the installation procedure or license installation
B Support

- on starting MLDesigner
- while working with MLDesigner
  While compiling a primitive
  Working with the standard libraries installed with the MLDesigner.
  Working with custom made libraries or creating a custom library.
  While running a simulation (extern, interactive, with animation)
- using external program systems such as SatLab™

Report at which step the error occurs. Report your input and the reaction of MLDesigner, such as error boxes or messages at the command prompt, as accurately as possible.

**NOTE:** These points are a guideline. You can add or omit points when reporting a problem.

The more details we receive, the easier it is for us to reproduce the error. Only if we can reproduce the error can we provide a valid workaround (if possible) or fix the bug in future versions and alternately provide a patch for download via FTP.
Appendix C

Frequently Asked Questions

This chapter is designed to help you find quick and short answers to your questions. All these topics are covered in detail in the MLDesigner user manual which can be opened by clicking the Help button in the top right corner of the GUI and selecting the option Search Index.

C.1 General Questions

Do I need to learn Ptolemy and BONeS to be able to design with MLDesigner? Go to Answer D.1.

C.2 Error Messages and Their Most Common Causes

1. Cannot create library "xxx" to location "file:/user/xxx/yyy/zzz/xxx.mml". Maybe you have no write permission. Cannot create model "xxx". You must create the library "yyy" first. Go to Answer 1.
2. Compilation error, unable to load the primitive Dynamic. Go to Answer 2.

C.3 Segmentation Faults

What are the most common causes of segmentation faults that cause MLDesigner to crash? Go to Answer D.3.

C.4 Data Structures

1. When must a data structure be deleted and when is it a clone?
2. What is a memory leak?

Go to Answer D.4.
C.5 Load Mode

What is the difference between Load Mode Dynamic and Load Mode Permanent? Go to Answer D.5.

C.6 Plotting Systems

How do I make data available to, or save data for analysis using external programs? Go to Answer D.6.

C.7 Setting Environment Variables

1. How do I change the environment variables so I can use another Editor to debug or change source code in my primitives? Go to Answer D.7.
3. How do I change the environment variables so I can use an external debugger? Go to Answer D.7.
4. Is setting the Working Directory entry the same as setting the $MLD_USER environment variable? Go to Answer D.7.

C.8 ddd debugger and Red Hat

I am running MLDesigner under Red Hat and I am unable to debug using the standard debugger ddd. Go to Answer D.8.

C.9 Linked Objects

Why must I restart MLDesigner before changes made to Linked Object take effect? Go to Answer D.9.

C.10 Shared Libraries

What is the environment variable $MLD_SHARED used for and how do I set this variable? Why are there invalid modules in my Shared Libraries? Go to Answer D.10.
Appendix D

Answers to FAQ’s

D.1 Answers for the General Questions

No but it is necessary to bear in mind that the Ptolemy base type int is different to any data structure type including those like Root.Integer. It is only possible to connect ports of the same type. Back to Question C.1.

D.2 Error Messages and Their Causes

1. This error message normally occurs when you do not have permission to write to the target directory or to one of the directories higher in the hierarchical structure. Another reason could be caused by a file that is write protected in the target directory when you have checked the option “Overwrite existing Files” in the appropriate Import dialog window. Back to Question C.2.

2. There are a number of reasons for this error message.

   - Firstly your primitive contains syntactical errors and can not be compiled by the C++ compiler. In that case you will get an error dialog (“Compilation failed.”), when you compile the primitive in MLD. You should open the primitive editor, try to compile it and correct the reported errors until successful compilation is confirmed.
   - Secondly a primitive with the same name already exists in the same Domain.
   - Another reason could be the primitive was already loaded with Load mode set to Permanent. If a Primitive is loaded permanent it is not possible to make any changes to its source code and then recompile the primitive. You must shutdown MLDesigner and restart before the changes take effect. You only need to load a primitive as permanent if you want it to be inherited by another primitive. A primitive that is a child of another primitive can only be recompiled if the parent is loaded as permanent.

3. This error is often caused by compiler incompatibility. What version of make do you use? You need gmake 3.74 or later. Please look at the section on system requirements in the app. A. Back to Question C.2.
D Answers to FAQ’s

D.3 Segmentation Faults / System Crashes

The most common reason for system crashes with MLDesigner is programmer error in user-defined primitives. When a segmentation fault occurs you should first check to see if your system contains user defined primitives. Often the crash is caused by inconsistencies in memory allocated to output datasets. The next step is to run the simulation `extern` possibly using an external debugger to debug the system. External simulations occur outside of the MLDesigner environment. See the MLDesigner manual index for more information on debugging. Back to Question C.3.

D.4 Data Structures

1. **When to clone a particle** Generally you don’t have to clone particles. In sending particles over the ports you can use operators like `<<` or `=`. For data structures the situation is more complicated, because you have to take into account how it is instantiated as a data structure and how it is used.

   **When to create a new particle** When you instantiate a new data structure for direct usage (not to be used as parameter in a method that takes `const Type*` as parameter - methods of `setType()` kind), you have to use the method `DsHandler::makeNewStructure()`.

   **When is a particle a clone?** When you set a data structure to a Memory, Event or a `DataStructMember`, the parameter is cloned inside the method so there is no need for a new created object.

   **When to delete a data structure:** When you get a data structure using the method `DsHandler::makeNewStructure()` be sure to delete it to avoid overloading memory. Every new data structure created using the `clone()` method must be deleted. The exception is when you place it on an output port. `DataStructParticles` take care to delete the data structure when the last reference to it is deleted. Do not delete `const` pointers to Type or class `derivedfrom`, returned by some methods. These are references to the class members and the object takes care of deleting them. On the contrary, non `const` pointers to Types must be deleted if you are not in the exceptional case previously mentioned.

   **Do not delete particles when** As an exception, don’t delete the non `const Type*` returned by `DataStructMember`’s methods `getData()` or `fieldWithName()` of the `DataStructure` class, or every time you write or read a Memory, using `writeMemory()`.

2. **A Memory Leak** occurs when new memory is allocated dynamically and never deallocated. In C++ new memory is created by the `new` operator and deallocated by the `delete` or the `delete []` operator. Memory leaks accumulate over time and can crash the program.

Back to Question C.4.

D.5 Load Mode

While creating a new primitive model component, you can define its load mode. By selecting the load mode Dynamic, a shared library containing the primitive code is created on loading. This shared library is linked dynamically to MLDesigner so that you can reload the primitive any time. All changes made to the primitive are effective immediately. Conversely, by selecting load mode Permanent, the primitive is linked to MLDesigner on loading as it would be with a built-in
D.6 Plotting Systems

1. There are a number of ways to display or handle datasets and results of simulations.
   - You may use SatLab to display data. (see Demos/DopplerIR, you need to start SatLab before running the demo!). It uses a Tcl script $MLD/MLD_Libraries/DE/Contrib/WiNeS.tcl.
   - You may save your data to one or more files and postprocess/display it with another tool of your choice. You could for example use a DEPrinter primitive to dump data to a file, then use GnuPlot, Matlab or even a spreadsheet program like StarCalc (within StarOffice) to display the information.

D.7 Setting Environment Variables

Change the Default Editor

To define the editor you prefer to use when working with source code it is no longer necessary to set the environment variables. A dialog is available where a variety of setting can be changed. To define which editor you prefer click the Settings option in the main menu. Expand the tree by clicking the Primitive Source Editor item as shown in fig. D.1. The following options are available:

1. Use external editors with xterm. Some editors open in a terminal and others not. Often the terminal is not required and only gets in the way. To stop the xterm from opening with your editor, click the check box to remove the tick. If the editor cannot open without a console, you will get an error message in the MLDesigner console when attempting to open a source file. In this case make sure you have a tick in the check box. Try open vi without a tick in this check box or set the editor variable to emacs in the $EDITOR input field.
2. Internal. This points to the built-in editor installed with MLDesigner.
3. $EDITOR This radio button sets the default editor to the one you have defined by setting your environment variable. i.e.,

   For **sh** and **bash** command shells:
   
   ```
   export EDITOR=emacs
   ```

   For **csh** and **tcsh** command shells:
   
   ```
   setenv EDITOR=emacs
   ```

   You must close and restart MLDesigner before changes take effect.
4. User-defined Has the same effect as the $EDITOR radio button except you do not have to close and reopen MLDesigner to get the new setting to work. You must only change the
editor name. The \$ are variables that instruct the editor you choose to open the source code of the open primitive in your Model Editor Window.

![Figure D.1: Generated hypertext documentation](image)

**NOTE:** The built-in editor is used as default editor in cases where compile errors occur. The reason is that you can highlight errors in the built-in editor error console and the cursor will be automatically placed in the correct line of the source code editor.

Back to Question C.7.

### Set the MLD_USER variable

It is possible to set your MLD_USER environment variable to point to a project library or external library. Let assume you want to work on a project called MLD.project. This project is the $MLD_USER directory of another user. You want to access the systems and share libraries that exist in the other user’s environment. Enter the following command where you would normally open MLDesigner:

- for **bash** or **sh** shells

  ```bash
  export MLD_USER=/home/user/MLD.project
  ```

- for **tcsh** and **csh** shells

  ```bash
  setenv MLD_USER /home/user/MLD.project
  ```
setenv MLD_USER /home/user/MLD.project

You could also do the same locally on your own computer if you wanted to separate libraries and projects. You can create a new directory /MLDProject in your home directory. Set your MLD_USER environment variable to point to the new directory. When you open MLDesigner again you will see the tree view with MLDesigner libraries and no user libraries. You could then create a top level library with read and write rights for a workgroup on your network.

Back to Question C.7.

Use an External Debugger

Depending on the type of shell you are using, enter one of the following at the prompt:

```
export MLD_PREBIN=ddd (sh and bash)
```

or

```
setenv MLD_PREBIN ddd (csh and tcsh)
```

The ddd entry refers to the ddd debugger supplied with every Linux package and can be replaced with a call referring to your favorite debugger.

You must now start MLDesigner as normal. The debugger you have chosen will start. If you are using ddd proceed as follows:

- Click the View menu and choose Command Tool.
- Click Run to start MLDesigner. This can take longer than normal.
- You can now work as normal with MLDesigner.

Back to Question C.7.

Differences Between $MLD_USER and Working Directory

$MLD_USER and Working Directory are not related to each other at all. The variable $MLD_USER defines where MLDesigner looks for user defined models and libraries where the option Working Directory defines which directory is the working directory for the command console window after starting MLDesigner. Both these topics are explained in the MLDesigner manual which can be opened by clicking on the Search Index option of the Help menu found in the top right corner of the GUI. Find the index entries Command Console or Working Directory to get more detail on the topic.

Back to Question C.7.

D.8 Using ddd Debugger under Red Hat

It is not possible to debug MLDesigner using ddd because Red Hat was developed completely using GCC 2.96 contrary to the recommendations of the Free Software Foundation (see http://www.fsf.org/software/gcc/gcc-2.96.html).
As a result ddd compiled under GCC 2.96 needs a symbol `dynamic_cast` defined in `libstdc++-libc6.2-2.so.3`.

With a SuSE system compiled using GCC 2.95 running under Red Hat, the system crashes when accessing the system internal `libstdc++-libc6.2-2.so.3`. This is apparently caused by a change in the library interface.

At the moment there are no solutions to the problem and we can only suggest you to use `gdb` or `xxgdb` as an alternative debugger.

Back to Question C.8.

### D.9 Linked Objects

Files containing object code must be loaded before the primitive that calls the external functions in the object file is compiled. All changes made to the external library will only be actualized when MLDesigner is shutdown and restarted. The reason is that it is not possible to delete the relevant primitive from memory without conflicts arising in the kernel’s reload mechanism. The result would be a primitive containing a mixture of new and old code.

Back to Question C.9.

### D.10 Shared Libraries

The Shared Libraries directory was introduced to make it easier for design teams to exchange models and work on group projects.

It is possible to develop a library within the environment specified by the `$MLD_USER` environment variable and then move this library to the shared environment specified by the `$MLD_SHARED` environment variable. However, since the `$MLD_USER` variable is dynamic (because it can be different for every user) the following prerequisite applies.

- All modules and files needed by systems in the library must be located within this library, or in a location that never changes, such as the directory to which `$MLD_SHARED` and `$MLD` point.

The reason for this is that the `systemName.mml` file contains references to all model elements needed for the system to function. If these variables change then MLDesigner will not be able to locate the missing model elements.

Back to Question C.10.
Appendix E

Troubleshooting

E.1 Closing complex models becomes slower and slower after simulations

Closing complex models becomes slower and slower after simulations. We see degradation because when a model completes execution, not all blocks are released. A second run on the same model means that the available memory is fragmented because some blocks from the previous run were not released. This makes memory allocations for successive runs more complex and makes closing slower because MLD designer hops around to release memory. One solution is to close and start MLD designer between simulations of complex models.

E.2 DHCP Client/License problem

Error: There are some reported errors while using a DHCP client with a non-permanent network connection (e.g. notebooks). MLD needs a connected network card for authentication of the license file. Some DHCP clients shut down the network connection if no valid DHCP server is found resulting in a change of the host id used for validating the license file (as long as there is no connection present).

Fix: We are working on this problem. In the meantime the only solution is to avoid changes in the network system. If you run into the error of inconsistent Host Id’s, then you can manually start the network card using the command 'ifconfig eth0 up' (as root).

NOTE: This usually works after the DHCP client has shut down the network card and will only work till the next reboot. The best solution is to deactivate the DHCP client in the Linux setup and to manually enter the network information so your computer will work both with and without a network connection.

E.3 Waiting for Users Lock

Sometimes it may happen when you start MLD Designer that you get a couple of the following messages:
Waiting for user’s lock on map file /home/user/MLD/.url.map

The reason is that MLDesigner creates a lock file on URL mapping files when it saves the URL map to the file. The message is thrown when another save operation tries to create a lock on the same URL map for write operation while the lock file from the former write operation still exists. This may for example happen when MLDesigner crashed during the save operation of the URL map for any reason. Waiting for the lock file slows down the model base access a lot. To solve this problem remove the file /home/user/MLD/.url.map.lock by hand.

E.4 Distributed Simulation Timeout

If you start an distributed C++ simulation, it may happen that the simulation process for an iteration started on a remote machine does not start with a given timeout period. This can happen if the remote host is very busy. In that case you get an error message as follows:

Iteration did not start within 10 seconds.

In that case, You try to increase the timeout value by changing the value in the Settings Dialog using the Start timeout for iterations [sec] input field of the Distributed item in the Run Control category.

E.5 ddd Debugger Problems

The following error message could display when attempting to run a simulation using ddd:

error loading shared library libDs.so: cannot open shared object: No such file or directory

The reason for this error is that in ddd the variable MLD_LIBRARY_PATH is not correctly set. This could be caused by the variable being set in ~/.cshrc without taking into account an already existing variable setting.

Check your .bashrc and/or .cshrc. The following would be incorrect:

.bashrc:

    export LD_LIBRARY_PATH=<any_path>

.cshrc/.tcshrc:

    setenv LD_LIBRARY_PATH <any_path>

Correct the files to look like:

.bashrc:

    if [ "$LD_LIBRARY_PATH" = "" ]; then
        export LD_LIBRARY_PATH=<any_path>
    else

If you have produced your own makefile and now are suddenly faced with a long list of errors and warnings you need to check if you have the following entry:

\[-I$(MLD)/include/kernel/\]

The above line needs to be changed to look like:

\[-I$(MLD)/include/kernel/ -I$(MLD)/include/\]

If a DE primitive has two or more output ports and it outputs more than one particle in one execution, the only way to ensure that the particles are received at their connected input ports in the same order is to fulfill the following conditions:

- Do not alternate the ports - output all the data for the first port, then send on the second port, and so on;
- Call sendData() after each output.

The library `MLD Examples/Tutorials/Multiple Outputs In DE` contains an example that illustrates all four possible cases (none of the conditions is fulfilled, only one or both conditions are fulfilled). The correct order is obtained by setting the parameter `Alternate` to `FALSE` and `SendData` to `TRUE`.

If the main window of MLDesigner is not displayed after the splash screen disappears, proceed as follows:

- Set the `MLD` variable to `mldpwdinf`, e.g. for `bash` shell:
Troubleshooting

```bash
export MLD=\'mldpwdinf\'
```

- Execute the `mld-x` script:

  ```bash
  $MLD/mld-x
  ```

Check the output messages displayed in the command window to detect what the problem might be. If you do not find a way to solve the problem after checking app. C and app. E, please read the app. B on how to contact the MLDesigner support group for further assistance.

## E.9 Red Hat Linux 9

<table>
<thead>
<tr>
<th>Error message</th>
<th>Reason</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wrong version during install</strong></td>
<td>Gcc 3.x is not properly detected during installation of an old MLDesigner version (2.4.r02 first build, or earlier). This renders the whole installation invalid!</td>
<td>Switch to a newer MLDesigner version or manually change 'system/probeversions' from the installation archive.</td>
</tr>
<tr>
<td><strong>No 'gunzip' found during installation</strong></td>
<td>The MLDesigner installer uses the 'gunzip' program for extracting the compressed archives. For Linux, it does not supply its own program, but uses the one given with the distribution. In one case, the 'gzip' program was not installed with Red Hat (or was broken).</td>
<td>(Re-)Installation of the gzip/-gunzip package from the distribution CDs.</td>
</tr>
<tr>
<td><strong>Account 'nobody' is not active</strong></td>
<td>This happens when you try to start the MLDesigner license daemon locally. The account 'nobody', used for security reasons, is not accessible.</td>
<td>Make the user valid by changing the necessary line in /etc/passwd from /sbin/nologin to /bin/bash or modify the file 'lmgrd-mld' to start license daemon not as 'nobody' but as 'root' (includes a warning whenever the daemon is started) or another appropriate user.</td>
</tr>
</tbody>
</table>
### Table E.1: Red Hat 9 Troubleshooting

<table>
<thead>
<tr>
<th>Error message</th>
<th>Reason</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>License daemon gives &quot;permission denied&quot; error</td>
<td>When starting the daemon, either by <code>mldminst</code>, booting or <code>mldlmgrd start</code>, the &quot;/root/.bashrc: Permission denied&quot; error occurs.</td>
<td>Either use a patch greater than or equal to 2.4.r02.p11/2.4.r03.p01 or edit the file <code>lmgrd-mld</code> manually and change <code>su nobody[...]</code> into <code>su - nobody[...]</code> on lines 84 and 91. Change the permission of &quot;/var/tmp&quot; manually to -m777.</td>
</tr>
<tr>
<td>Permission denied on &quot;/var/tmp&quot;</td>
<td>If the directory &quot;/var/tmp&quot; is not writable for everyone (including 'nobody'), the license daemon cannot start.</td>
<td></td>
</tr>
</tbody>
</table>

### E.10 Red Hat Enterprise Linux 4 32 bit

A possible solution to get MLDesigner running on a Red Hat Enterprise Linux 4 32 bit system is to downgrade the GCC compiler to version 3.2 (GCC 3.2 is shipped with Red Hat Enterprise Linux 4 as compatibility packages) following these steps:

- install packages in category "Old development packages"
- rename `/usr/bin/gcc` to `/usr/bin/gcc34`
- rename `/usr/bin/g++` to `/usr/bin/g++34`
- rename `/usr/bin/gcc32` to `/usr/bin/gcc`
- rename `/usr/bin/g++32` to `/usr/bin/g++`

We have tested MLDesigner after downgrading GCC without any problem. Please note that we cannot guarantee that this compiler downgrade has no effect on other compilations like the installation of software that is downloaded from network and has to be compiled from source code.

### E.11 Security-Enhanced Linux (SELinux)

When installing or running MLDesigner on an operating system with SELinux enabled, you may get the message:

```
error while loading shared libraries: <library_name>: cannot restore segment prot after reloc: Permission denied
```

There are two ways to solve this problem:

1. Change the security context for the MLDesigner’s shared libraries by issuing the command (as super user):
find $INSTALL/lib -name "*.so" \
-exec chcon -t texrel_shlib_t {} \;

where $INSTALL points to the directory where MLDesigner installation files reside. If you install from a CD-ROM, copy the installation files to a local directory and run the command mentioned above. Then run the installation from that directory. The same problem could appear with the installed MLDesigner libraries. If so, run the above command with $INSTALL pointing to the MLDesigner installation directory. While simulating a model, MLDesigner may create temporary shared libraries and load them to the running process. If the same problem appears with those shared libraries, you probably have to disable SELinux.

2. Disable SELinux by replacing ‘enforcing’ with ‘disable’ in the SELinux’s config file:

```
#SELINUX=enforcing
SELINUX=disable
```

E.12 VMWare Player

It may happen that during an MLDesigner session under VMWare Player the mouse cursor disappears if a mouse click is followed by a second one in less than 30s and reappears after a short period of inactivity. Once unleashed, this problem is present also when other applications are used (for example, mail client, Internet browser) and only a restart of the X-Server can fix it. A workaround is to edit the file /etc/X11/xorg.conf in the virtual machine’s operating system by adding the lines

```
Option "SWcursor" "true"
Option "HWcursor" "off"
```

to Section "Device" and the line

```
InputDevice "Mouse[1]" "CorePointer"
```

to Section "ServerLayout".

E.13 QClipboard::Unknown SelectionNotify

Sometimes the message "QClipboard::Unknown SelectionNotify" appears when KDE clipboard tool ‘klipper’ is running. ‘klipper’ has to be closed to avoid this messages.
E.14 Value of MLD_USER variable is lost

The value for the MLD_USER is read from the corresponding environment variable during MLDesigner start-up time. This value can be changed using the Settings dialog, but it is available only for the current MLDesigner session. To have it persistent for several MLDesigner sessions, the MLD_USER environment variable must be set in the console from which MLDesigner is started.

E.15 The license manager fails on a system with multiple NICs

During system startup, there exists a race condition in initialization of the network interface cards on a system with multiple NICs. If eth0 is not assigned to the card whose MAC address is in the license file, the license manager complains that there is no available license. To ensure that eth0 is always assigned to the network interface card whose MAC address has been used to obtain the license, the line

```
PERSISTENT_NAME='eth0'
```

must be added to the configuration file of that network card. In a similar way, different names must be specified for the other network interface cards.

E.16 MySQL

Check the app. A.2 to find out what environment variables are required in case MLDesigner does not find the MySQL development headers or the shared library 'libmysqlclient.so'.
**Appendix F**

**Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>Asynchronous Transfer Mode network protocol</td>
</tr>
<tr>
<td>EPS</td>
<td>Encapsulated PostScript</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is Not Unix</td>
</tr>
<tr>
<td>MDI</td>
<td>multiple Document Interface</td>
</tr>
<tr>
<td>MLD</td>
<td>Mission Level Design</td>
</tr>
<tr>
<td>PS</td>
<td>PostScript</td>
</tr>
<tr>
<td>PTCL</td>
<td>Ptolemy Tcl</td>
</tr>
<tr>
<td>ptlang</td>
<td>Ptolemy language</td>
</tr>
<tr>
<td>Tcl</td>
<td>Tool command language</td>
</tr>
<tr>
<td>UTR</td>
<td>Utah Raster Toolkit</td>
</tr>
</tbody>
</table>
# Appendix G

## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MLD</td>
<td>The directory in which the MLD software is installed.</td>
</tr>
<tr>
<td>$MLD_USER</td>
<td>The directory where the user libraries are stored, usually <code>~ /MLD</code>.</td>
</tr>
<tr>
<td>actor</td>
<td>An atomic (indivisible) function in a data flow model of computation.</td>
</tr>
<tr>
<td>ATM</td>
<td>A sub-domain of the synchronous data flow and discrete-event domains to provide the infrastructure for simulating ATM networks.</td>
</tr>
<tr>
<td>auto-fork</td>
<td>A fork primitive that is automatically inserted when a single output is connected to more than one input.</td>
</tr>
<tr>
<td>base class</td>
<td>A C++ object that is used to define common interfaces and common code for a set of derived classes. An object may be a base class and a derived class simultaneously.</td>
</tr>
<tr>
<td>BDF</td>
<td>A domain using the Boolean-controlled data flow model of computation. This domain attempts to use compile-time scheduling, but will fall back to run-time scheduling if necessary.</td>
</tr>
<tr>
<td>behavioralmodeling</td>
<td>System modeling consisting of functional specification plus modeling of the timing of an implementation (cf. functional modeling).</td>
</tr>
<tr>
<td>block</td>
<td>A graphical representation of a model instance.</td>
</tr>
<tr>
<td>block diagram</td>
<td>A graphical representation of a functional model that embeds a number of model components using model instances (blocks) which are connected by net objects.</td>
</tr>
<tr>
<td>Block</td>
<td>The base class defined in the kernel for primitives, modules, systems, and targets.</td>
</tr>
</tbody>
</table>
**block**
A primitive or a module.

**boolean-controlled data flow**
A model of computation that includes synchronous data flow, but adds actors that may or may not produce or consume tokens on any given input or output. Whether these actors produce or consume tokens depends on a Boolean signal.

**code generation**
The synthesis of a standalone implementation in some target language from a network of MLDesigner blocks.

**code generation domain**
A domain that supports code generation, but not simulation.

**CG**
A domain that defines many of the base classes and schedulers used in code generation domains. It has no direct application by itself.

**CG56**
A domain that synthesizes assembly code for the family of Motorola DSP56000 digital signal processors. It uses the synchronous data flow model of computation.

**CG96**
A domain that synthesizes assembly code for the family of Motorola DSP96000 digital signal processors. It uses the synchronous data flow model of computation.

**CGC**
A domain that synthesizes C code. It uses the synchronous data flow or Boolean-controlled data flow model of computation.

**CG-DDF**
A code generation domain that uses the dynamic data flow model of computation. This has not been maintained beyond version 0.4.1 of Ptolemy.

**codesign**
The simultaneous design of the software and hardware composing a system.

**communicating processes**
A model of computation in which multiple processes execute concurrently and communicate with one another by passing messages.

**compile-time scheduling**
A scheduling policy in which the order of execution of blocks is precomputed when the execution is started. The execution of the blocks thus involves only sequencing through this precomputed order one or more times (cf. run-time scheduling).

**CP**
A simulation domain using the communicating processes model of computation. Each primitive forms a process that runs under the Sun lightweight
process library.

derived class A C++ object that is derived from some base class. It inherits all of the members and methods of the base class.

data flow A model of computation in which actors process streams of tokens. Each actor has one or more firing rules. Actors that are enabled by a firing rule may fire in any order.

DDF A simulation domain that uses the dynamic data flow model of computation.

DE A simulation domain that uses the discrete-event model of computation. In the DE domain, particles transmitted between blocks represent events that trigger changes in system state. Events carry an associated timestamp, and are processed in chronological order.

discrete event A model of computation used to model systems that change state abruptly at arbitrary points in time, such as queuing networks, communication networks, and computer architectures. A block is enabled when an event at one of its inputs is the "oldest" event in the system, in that its timestamp has the smallest value. Once enabled, the block may be executed, and in the process may produce more events.

domain A specific implementation of a model of computation.

Domain The base class in the MLDesigner kernel from which all domains are derived.

drag The action of holding a mouse button while moving the mouse.

dynamic data flow A model of computation supporting any computable firing rule for actors. This model of computation requires run-time scheduling.

event A particle generated by a block in a discrete-event model of computation. This particle carries a timestamp.

event horizon The interface between domains that manages the flow of particles from one domain to another.

FFT The Fast Fourier Transform is an efficient way to implement the discrete Fourier transform in digital hardware.

firing A unit invocation of an actor in a data flow model of computation.

firing rule A rule that specifies how many tokens are required on each input of a data flow actor for that actor to be enabled for firing.
<table>
<thead>
<tr>
<th><strong>fork primitive</strong></th>
<th>A primitive that reads one input particle and replicates it on any number of outputs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FSM</strong></td>
<td>Finite State Machine. A system of states, controlled by conditions. Often used to describe and analyze sequential control problems.</td>
</tr>
<tr>
<td><strong>FSM model</strong></td>
<td>The functional model of an FSM primitive component defined by states and transitions.</td>
</tr>
<tr>
<td><strong>FSM primitive</strong></td>
<td>A primitive model component that models an FSM.</td>
</tr>
<tr>
<td><strong>functional modeling</strong></td>
<td>System modeling that specifies input/output behavior without specifying timing (cf. behavioral modeling).</td>
</tr>
<tr>
<td><strong>galaxy</strong></td>
<td>A synonym for module components.</td>
</tr>
<tr>
<td><strong>Galaxy</strong></td>
<td>The class (derived from Block) in the MLDesigner kernel that represents a network of other blocks.</td>
</tr>
<tr>
<td><strong>Gantt chart</strong></td>
<td>A graphical display of a parallel schedule of tasks. In MLD, the tasks are the firings of primitives and modules.</td>
</tr>
<tr>
<td><strong>higher-order functions</strong></td>
<td>Functional programming constructs that apply a function a determined number of times to one or more streams of inputs. Examples of higher-order functions from Lisp include mapcar and apply.</td>
</tr>
<tr>
<td><strong>HOF</strong></td>
<td>A domain implementing higher-order functions that are expanded at compile-time and incur no run-time overhead. HOF primitives are typically embedded in other domains, and provide graphical expression of parameterized parallel, cascaded, and recursive structures.</td>
</tr>
<tr>
<td><strong>homogeneous synchronous data flow</strong></td>
<td>A particular case of the synchronous data flow model of computation, where actors produce and consume exactly one token on each input and output.</td>
</tr>
<tr>
<td><strong>icon</strong></td>
<td>A graphical object that represents a single block or library.</td>
</tr>
<tr>
<td><strong>iteration</strong></td>
<td>A set of executions of blocks that constitutes one pass through the precomputed order of a compile-time schedule.</td>
</tr>
<tr>
<td><strong>kernel</strong></td>
<td>The set of classes defined in the directory $MLD/src/kernel.</td>
</tr>
<tr>
<td><strong>library</strong></td>
<td>A model component that contains a number of model instances that refer to</td>
</tr>
</tbody>
</table>
model components that belong to a library.

**master**  
In MLDesigner, the directory that contains all model related files.

**MDSDF**  
A simulation domain that uses a multidimensional extension to the synchronous data flow model of computation. Actors in MDSDF consume data defined on rectangular grids, e.g., a sub-block in an image.

**member**  
A C++ object that forms a portion of another object.

**method**  
A function defined to be part of an object in C++.

**model component**  
A part of a system model that consists of a functional model and an icon representation.

**model instance**  
A reference to a model component that is embedded into another model

**model of computation**  
A set of semantic rules defining the behavior of a network of blocks.

**module**  
A model component that contains a network of embedded other model components.

**net**  
A graphical connection between ports in MLDesigner.

**node**  
An edge point of a net object that can be used as connector for a number of connections.

**object**  
A data type in C++ consisting of members and methods. These members and methods may be private, protected, or public. If they are private, they can only be accessed by methods defined in the object. If they are protected, then they can also be accessed by methods in derived classes. If they are public, then they can be accessed by any C++ code.

**OCT**  
A design database developed by the CAD Group at U. C. Berkeley. Oct is used to store graphical representations of MLDesigner models.

**palette**  
A synonym for library models.

**parameter**  
A member of a block that stores data values from one invocation of the block to the next.

**particle**  
A datum (e.g., a floating-point value) communicated between blocks.
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Plasma</strong></td>
<td>A class in the MLDesigner kernel that serves as a repository for used particles of any particular types. When new particles of the appropriate type are needed, they are taken from the Plasma, if possible, thus avoiding memory allocation.</td>
</tr>
<tr>
<td><strong>PN</strong></td>
<td>A simulation domain based on the process networks computational model. Each primitive forms a process under this domain.</td>
</tr>
<tr>
<td><strong>port</strong></td>
<td>An input or output of a primitive or module.</td>
</tr>
<tr>
<td><strong>PortHole</strong></td>
<td>The base class in the MLDesigner kernel for all ports.</td>
</tr>
<tr>
<td><strong>primitive</strong></td>
<td>An atomic (indivisible) unit of computation in an MLDesigner application. Every MLDesigner simulation ultimately consists of executing the methods of the primitives used to define the simulation.</td>
</tr>
<tr>
<td><strong>primitive source</strong></td>
<td>The functional model of primitive model components written in the Ptolemy language.</td>
</tr>
<tr>
<td><strong>PTCL</strong></td>
<td>The built-in textual, interactive command interpreter of MLDesigner. As the name implies, PTCL is based on Tcl.</td>
</tr>
<tr>
<td><strong>ptlang</strong></td>
<td>A schema language used to define primitives in MLDesigner.</td>
</tr>
<tr>
<td><strong>MLD</strong></td>
<td>An environment variable with value equal to the name of the directory in which the MLDesigner system is installed.</td>
</tr>
<tr>
<td><strong>real time</strong></td>
<td>The actual time (cf. simulated time).</td>
</tr>
<tr>
<td><strong>run-time scheduling</strong></td>
<td>A scheduling policy in which the order of execution of the blocks is determined “on-the-fly,” as they are executed (cf. compile-time scheduling).</td>
</tr>
<tr>
<td><strong>Scheduler</strong></td>
<td>An object associated with a domain that determines the order of execution of blocks within the domain. Domains may have multiple schedulers.</td>
</tr>
<tr>
<td><strong>schematic</strong></td>
<td>A block diagram.</td>
</tr>
<tr>
<td><strong>SDF</strong></td>
<td>A simulation domain using the synchronous data flow model of computation.</td>
</tr>
<tr>
<td><strong>Silage</strong></td>
<td>(1) A functional language developed by Paul Hilfinger at U. C. Berkeley for specifying signal processing systems. It is used primarily as input for VLSI synthesis tools.</td>
</tr>
<tr>
<td></td>
<td>(2) A code generation domain in MLDesigner that synthesizes Silage code.</td>
</tr>
</tbody>
</table>

MLDesigner Version 2.8
and uses the synchronous data flow model of computation.

**simulated time**  In a simulation domain, the real number representing time in the simulated system (cf. real time).

**simulation**  The execution of a system specification (an MLDesigner block diagram) from within the MLDesigner process (i.e., without generating code and spawning a new process to execute that code).

**simulation domain**  A domain that supports simulation, but not code generation.

**snap**  In MLDesigner, an invisible grid defining the points at which graphical objects can have endpoints or corners.

**star**  A synonym for primitive model components.

**Star**  The base class in the MLDesigner kernel for all primitives.

**State**  The base class in the MLDesigner kernel for all parameters.

**stop time**  Within a timed domain, the time at which a simulation halts.

**synchronous data flow**  A data flow model of computation where the firing rules are particularly simple. Every input of every actor requires a fixed, pre-specified number of tokens for the actor to fire. Moreover, when the actor fires, a fixed, pre-specified number of tokens is produced on each output. This model of computation is particularly well-suited to compile-time scheduling.

**system**  An entire MLDesigner system model, that is, the top-level model.

**target**  An object that manages the execution of a simulation or code generation process. Thus, for example, in code generation, the target would be responsible for compiling the generated code and spawning the process to execute that code, if desired.

**Target**  The base class in the kernel for all targets.

**Tcl**  Tool command language, a textual, interpreted language developed by John Ousterhout at U.C. Berkeley. Tcl is embedded in PTCL.

**timestamp**  A real number associated with a particle in timed domains that indicates the point in simulated time at which the particle is valid.
<table>
<thead>
<tr>
<th><strong>timed domain</strong></th>
<th>A domain that models the evolution of a system in time.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tk</strong></td>
<td>An X windows toolkit for Tcl. Tk is embedded in MLDesigner, which uses it extensively. The interactive sliders, buttons, and plotting capabilities of pigi are implemented in Tcl/Tk.</td>
</tr>
<tr>
<td><strong>token</strong></td>
<td>A unit of data in a data flow model of computation. Tokens are implemented as particles in MLDesigner.</td>
</tr>
<tr>
<td><strong>universe</strong></td>
<td>A synonym for system models.</td>
</tr>
<tr>
<td><strong>URT</strong></td>
<td>The Utah Raster Toolkit for image and video processing. It is used by the image processing primitives in the synchronous data flow domain. The multidimensional synchronous data flow domain treats images as matrices and does not use the Utah Raster Toolkit.</td>
</tr>
<tr>
<td><strong>VHDL</strong></td>
<td>The VHSIC hardware description language, a standardized language for specifying hardware designs at multiple levels of abstraction.</td>
</tr>
<tr>
<td><strong>VHDLF</strong></td>
<td>A code generation domain for functional modeling of hardware. This domain synthesizes a system description in VHDL.</td>
</tr>
<tr>
<td><strong>VHDLB</strong></td>
<td>A code generation domain for behavioral modeling of hardware. This domain synthesizes a system description in VHDL.</td>
</tr>
<tr>
<td><strong>wormhole</strong></td>
<td>A primitive in a particular domain that internally contains a module in another domain.</td>
</tr>
</tbody>
</table>
Appendix H

Bibliography


[nam] Network Animator. http://www.isi.edu/nsnam/nam/. 25.3.2.3, 25.3.2.4

[NS2a] Network Simulator Documentation. http://www.isi.edu/nsnam/ns/ns-documentation.html. 25.2.2.1, 25.2.2.2, 25.2.2.3, 25.2.2.4, 25.3.3.1, 25.3.3.5


[Ste] Enrico Steiger. Detailed model for IEEE802.11 WLAN; Add-on for MLDesigner. Technical report, MLDesign Technologies, Inc. 25.2.1


Appendix I

Index
abortRun function, 13-39
accessMessage function, 14-20
acknowledge item, 13-11
actor, G-1
actual parameter, 3-11
Add Member dialog, 12-3
Add Text Label dialog, 3-34
Add Transition, 2-29
addPoint function, 13-41
alias command, 9-2, 9-8
animation command, 9-2, 9-14
Argument, 1-1
ArrayState class, 13-28, 13-29
arrivalTime, 17-8
author item, 13-11
displayList, 3-36
displayList function, 12-29
auto-fork, G-1
Auto-Forking, 3-29
bar function, 13-48
BarGraph class, 13-42, 13-43
Baseline class, 14-44
BDF domain, 5-3, 6-3, 17-6, 20-1, G-1
BDFPortHole class, 17-6
begin item, 13-17, 13-18, 13-18, 13-22
block, G-2
Block class, 5-2, G-1
block diagram, 3-1
displayList function, 12-29
displayList, 3-36
displayList function, 12-29
editor, 3-1
body item, 13-21
BONES
    Primitive Categories, 11-16
    Primitives Alphabetically, 11-37
BONES Conversion
    Convert Project Libraries, 11-5
    Error Messages after Conversion, 11-8
    Troubleshooting, 11-7
Boolean Data Flow, 6-3
boolean-controlled data flow, G-2
Breakpoints
    Dynamic Instance Breakpoints, 4-4
    Module breakpoint example, 4-3
    Module Breakpoints, 4-3
    Place a breakpoint, 4-1
    The Breakpoint Console, 4-1
    Unconditional Breakpoints, 4-2
Bus, 3-32
callFunction, 13-37
cancelAction command, 9-2, 9-19
canGetFired, 17-15
cascade mode, 2-11
cclinclude item, 13-20
cd command, 9-2, 9-17
CG domain, 6-6, G-2
CG56 domain, 6-6, G-2
CG96 domain, G-2
GCG domain, 6-6, G-2
Cloning Data Structures, 15-8
code generation domain, 5-1, G-2
Code Generation Domains, 27-1
code item, 13-20, 13-22
codeblocks item, 13-21
codesign, G-2
communicating processes, G-2
Compile
    with Debug, 4-10
    Compile Primitive, 13-6
compile-time scheduling, G-2
completionTime, 17-8
Complex class, 13-27, 13-28, 14-1, 14-24
Complex type, 14-2, 14-2
ComplexArrayState class, 13-27, 13-28
ComplexParticle class, 13-26
ComplexState class, 13-27
ComplexSubMatrix class, 14-40
ComplexMatrix B = A.hermitian(); class, 14-30
ComplexMatrix class, 14-23, 14-24, 14-30
ComplexMatrixParticle class, 14-43
ComplexState class, 13-27, 13-28
Composite Data Structures, 12-13
connect command, 9-2, 9-7
conscalls item, 13-17, 13-20
console window, 2-20
const char* key() function, 13-52
ConstTypeRef class, 15-2
constructor function, 13-37
constructor item, 13-17, 13-22
ConstTypeRef class, 15-3
cont command, 9-2, 9-13
copyright item, 13-11
COSSAP
Model Definition File(.mdef), 11-59
Conversion Process, 11-57
Troubleshooting, 11-67
COSSAP Model Conversion
Conversion Limitations, 11-64
Model Conversion, 11-53
CP domain, G-3
Create Model from Source, 2-44
Create New Model dialog, 2-37
CTDE Domain, 23-1
Introduction, 6-4
CTDE domain, 23-1
cursystem command, 9-2, 9-6
curuniverse command, 16-9
data flow, G-3
  boolean-controlled, G-2
dynamic, G-3
  homogeneous synchronous, G-4
  synchronous, G-7
Data structure, 1-3
Data Structure Editor dialog, 12-2
Data Structure Libraries, 12-13
Data Structures
  Base types, 12-11
  Clone, 15-1
  Composite, 12-13
  Create New DS, 12-2
  Delete, 15-1
  Known Problems, 15-10
  Overview, 12-5
  String, 12-10
  Using Data Structures, 12-1
  Vectors, 12-11
DataStructState class, 13-27
dataNew, 17-13
DataStructClass class, 15-11
DataStructMember class, 15-6, 15-8, 15-10, 15-11
DataStructParticle class, 13-26
DataStructure class, 15-8, 15-11
DataTypeException class, 15-7
DCTImage class, 14-44
DDF
  Backward Scheduler, 19-3
  DDF domain, 5-3, 6-2, 17-2, 19-1, G-3
  DDFStar class, 17-3
  DE Delay Primitive, 17-8
  DE Domain
    Introduction, 6-3
    DE domain, 5-5, 17-7, 22-1, G-3
    Debug Mode, 4-1, 7-8
    Debugging
      External Debugger, 4-11
      Using Probes, 4-6
defevent item, 13-13
definition of parameters, 3-11
defmemory item, 13-14
defmodule command, 9-2, 9-10
defparameter item, 13-12, 13-27
defprimitive item, 13-7, 13-8, 13-22
defresource item, 13-14
defstate item, 13-12
Delay
  DE Delay Primitives, 17-8
  Delays, 3-33
    SDF, 18-4
    When to use, 3-29
  Delays and Buses, 3-32
delayType, 17-8
delds command, 9-2, 9-26
delete [] function, 13-37
delete function, 13-37, 13-38
delnode command, 9-2, 9-15
delprimitive command, 9-2, 9-14
delsystem command, 9-2, 9-6
Demos
  Digital Signal Processing, 18-46
  Image processing, 18-48
  Sound, 18-47
DEPortHole
dataNew, 17-13
DERepeatStar
canGetFired, 17-15
refireAtTime, 17-15
DERepeatStar class, 17-15, 17-17
Derive, 3-4
derived item, 13-10, 13-10
Derived Primitive
go, 13-18
derivedfrom item, 13-10
desc item, 13-10
descriptor command, 9-3, 9-12
descriptor item, 13-10, 13-13
Designing Primitives
  BDF Domain, 17-6
  DDF Domain, 17-2
  DE Domain, 17-7
  Introduction, 13-1
  SDF Domain, 17-1
DEStar
  arrivalTime, 17-8
  completionTime, 17-8
  delayType, 17-8
  setMode, 17-8
DEStar class, 17-8, 17-15
destructor item, 13-17, 13-17, 13-22
dialog
  Add Member dialog, 12-3
  Add Text Label dialog, 3-34
  Create New Model dialog, 2-37
  Data Structure Editor dialog, 12-2
  Print dialog, 2-47
  Printer Setup, 2-48
  Save As New Model dialog, 2-40
  Select color dialog, 3-35
  Select Model dialog, 2-28, 3-15
  disconnect command, 9-3, 9-15
Discrete event, 6-3
discrete event, G-3
Distributed External Simulations, 7-15
domain, 5-1, 6-7, G-3
  BDF domain, 5-3, 6-3, G-1
  CG domain, 6-6, G-2
  CG56 domain, 6-6, G-2
  CG96 domain, G-2
  CGC domain, 6-6, G-2
  code generation domain, 5-1, G-2
  CP domain, G-3
  DDF domain, 5-3, 6-2, G-3
  DE domain, 5-5, G-3
  HOF domain, 6-2, G-4
  MDSDF domain, 6-5, G-5
  PN domain, G-6
  SDF domain, 5-3, 6-2, G-6
  simulation domain, 5-1, G-7
  SR domain, 6-5
  supported domains, 6-2
  unsupported domains, 6-5
  VHDL domain, 6-6
  VHDLB domain, 6-6, G-8
  VHDLF domain, G-8
  domain command, 9-3, 9-7
  domain concept, 6-1
domain item, 13-9
domains
  BDF, 17-6, 20-1
  CTDE, 23-1
  DDF, 17-2, 19-1
  DE, 17-7, 22-1
  HOF, 21-1
  NS2, 25-1
  SDF, 17-1, 18-1
  domains command, 9-3, 9-7
drag, G-3
DSFieldIter class, 15-11
DsHandler class, 15-10
DSMemberIter class, 15-11
Dynamic
  Instances, 3-38
  Linking libraries, 3-39
  Dynamic Data Flow, 6-2
Index

dynamic data flow, G-3
Dynamic Referencing, 2-53
DynDFStar class, 17-3

edit menu, 2-23
  Add Default Entrance, 2-29
  Add History, 2-29
  Add Input Port, 2-28
  Add Memory, 2-28
  Add Model Instance, 2-28
  Add Output Port, 2-28
  Add Resource, 2-28
  Add State, 2-29
  Add Text Label, 2-29
  Background Color, 2-28
  Compile Source, 2-26
  Copy, 2-25
  Cut, 2-25, 2-26
  Delete, 2-25
  Disconnect Connected, 2-27
  Dynamic Instance, 2-27
  Hide Port Labels, 2-27
  Mirror Ports, 2-27
  Online Documentation, 2-26
  Open Base Primitive, 2-26
  Open FSM, 2-26
  Open Model, 2-26
  Open Source, 2-26
  Redo, 2-25
  Replace Instance, 2-27
  Rotate Ports, 2-27
  Select All, 2-26
  Select Icon, 2-27
  Show Port Labels, 2-27
  Swap Ports, 2-27
  Terminate Unconnected, 2-28
  Undo, 2-25
  Unterminate Terminated, 2-28

Editor
  Built-in, 3-45
  External, 3-45
  Envelope class, 14-20
  EnumState class, 13-27
  EnumType class, 15-5
  Envelope class, 13-24, 14-16, 14-18, 14-19, 14-21, 14-22, 14-33, 14-37
  Environment Variables
    $MLD_USER, 2-54
    MLDesigner, A-3
    Dynamic Referencing, 2-53
    Source Code Editor, 3-45
  EPS
    Vector Graphic Export, 2-50
  ErrAdd function, 13-38
  Error class, 13-39, 13-39
  Error::abortRun function, 13-39
  ErrorHandle parameter, 22-19
  event, G-3
  event horizon, G-3
  Events, 3-36, 10-3
  Examples
    FSM Model, 24-28
  exectime item, 13-21, 13-21, 13-22
  execute command, 9-3, 9-12, 9-16
  exit command, 9-3, 9-18
  expandPathName function, 13-37, 13-38, 13-41
  explanation item, 13-11
  Export
    Parameters, 3-12
    Export Libraries, 2-51
      Precompiled Objects, 2-51
    External Editor, 3-45
  File
    Naming conventions, A-3
  file menu, 2-21
    Close, 2-22
    Convert BONeS, 2-23
    Export EPS, 2-23, 2-50
    Import COSSAP, 2-23
    New, 2-22
    New Model, 2-37
    Open, 2-22
    Print, 2-23, 2-47
    Quit, 2-23
    Reload, 2-22
    Save, 2-22
    Save All, 2-23
    Save As, 2-23
File Names, A-3
File View, 2-12
FileMessage class, 14-36
FileParticle class, 14-36
firing, G-3
firing rule, G-3
Fix class, 14-1, 14-4–14-7, 14-10–14-15, 14-24
Fix type, 14-4, 14-4, 14-5
FixArrayState class, 13-27, 13-28
FixParticle class, 13-26
FixState class, 13-27
FixSubMatrix class, 14-40
FixArray class, 14-5
FixMatrix class, 14-23, 14-24, 14-26, 14-27
FixMatrixParticle class, 14-43
FloatArrayState class, 13-27, 13-28–13-30
FloatParticle class, 14-16
FloatState class, 13-27, 13-29
FloatSubMatrix class, 14-40
FloatArrayState class, 13-31
FloatMatrix class, 14-23, 14-24, 14-27, 14-32, 14-33, 17-20
FloatMatrixParticle class, 14-43
FloatParticle class, 13-26, 14-32
FloatState class, 13-27, 13-28
FloatVector class, 15-5
Fork
   Auto, 3-29
   fork primitive, G-4
   formal parameter, 3-11
Formal Ports, 5-6
free function, 13-37
FSM, 13-1, G-4
   primitive, 1-2
FSM Domain
   Concurrent Models, 24-26
   Creating an FSM, 24-28
   Design Check, 24-25
   Design Objects, 24-22
   Elevator Example, 24-17
   Execution Semantics, 24-15
   Introduction, 6-4, 24-1
   Model Editor, 24-21
   Model Interface, 24-18
   Reverting Converted Models, 24-35
Semantics, 24-2
   States, 24-3
   Transitions, 24-4
full mode, 2-11
Gantt chart, G-4
dataGet function, 13-44
dataMessage function, 14-16, 14-20
dataGetSimulEvent, 17-14
Global Memory
   Read, 10-3
   Write, 10-3
GrayImage class, 14-44
GUI, 1-3
cascade mode, 2-11
can console window, 2-20
File View, 2-12, 2-15
full mode, 2-11
Library View, 2-14, 2-15
Logical View, 2-14
Physical View, 2-13
tile mode, 2-11
Tree View, 2-11
Tree View filter, 2-16
workspace, 2-11
halt command, 9-3, 9-13
HashEntry class, 13-52
hashstring function, 13-39
HashTable class, 13-51, 13-51, 13-52
HashTableIter class, 13-52
header item, 13-20, 13-22
help command, 9-3, 9-18
hierarchical, 13-1
Higher Order Functions, 21-1
Higher-order functions, 6-2
higher-order functions, G-4
hinclude item, 13-20
Histogram class, 13-44, 13-44
histogram class, 13-40
HOF Demos, 21-15
HOF Domain, 21-1
HOF domain, 6-2, 21-1, G-4
homogeneous synchronous data flow, G-4
htmlDoc item, 13-10, 13-11, 13-11

ifstream class, 13-41
Image processing
  SDF Demos, 18-48
Import Libraries item, 12-9
Import Libraries Parameter, 12-9
InDEPort
  getSimulEvent, 17-14
  numSimulEvents, 17-14
InDEPort class, 17-13
IndexOutOfRange class, 15-4
InfString class, 13-48, 14-37, 14-37
InfString class, 13-38, 13-46, 13-47, 13-48
Init Primitive, 17-17
inline item, 13-17–13-19, 13-21
inmulti item, 13-15
inout item, 13-15
inoutmulti item, 13-15
input item, 13-15, 13-23
InSDFMPHIter class, 13-26
InSDFPort class, 13-22
InSDFPorts class, 13-25
IntArrayState class, 13-27, 13-28
IntParticle class, 14-16
IntState class, 13-27
IntSubMatrix class, 14-40
IntMatrix class, 14-23, 14-24, 14-27
IntMatrixParticle class, 14-43
IntParticle class, 13-26
IntState class, 13-27, 13-28
IntVecData class, 14-19, 14-21
IntVector class, 15-5
InXXXPort class, 13-22
isA, 14-17
ISA_INLINE, 14-17
Iteration, 1-18
iteration, G-4
Iterations
  SDF Domain, 18-3
kernel, 5-1, G-4
kernel class
  Block class, 5-2, G-1
  Scheduler class, 5-2
  Wormhole class, 5-3
knownlist command, 9-3, 9-11

Libraries
  DSHandling, 12-13
  Multi-valued Logic, 22-19
  Shared Libraries, 2-50
Library, 1-2
  create a, 1-6
    independent, 1-3
    top-level, 1-3
Library View, 2-14
  link command, 9-3, 9-17
  Linked Objects, 3-39
Linking
  Dynamic, 3-39
  linking parameter, 3-20
    direct, 3-20
    indirect, 3-20
ListIter class, 13-49
listobjs command, 9-3, 9-12
Load Mode
  Permanent/Dynamic, 13-3
location item, 13-11
Logical View, 2-14
makeLower function, 13-37
malloc function, 13-37
master, G-5
mathematica command, 9-3, 9-19
Matlab
  Interface primitives, 18-24
matlab command, 9-3, 9-19
matrix classes
  ComplexMatrix, 14-23
  FixMatrix, 14-23
  FloatMatrix, 14-23
  IntMatrix, 14-23
MatrixEnvParticle class, 14-24
MatrixEnvParticle class, 14-24, 14-34
MDI, 2-3
MDSDF domain, 6-5, G-5
Memories, 3-36, 10-1
memory leak, 15-9
menu
  edit menu, 2-23
  file menu, 2-21
view menu, 2-29
window menu, 2-30
menu bar, 2-21
Merge
  Auto, 3-29
Message class, 13-24, 14-1, 14-16, 14-20, 14-22–14-24, 14-33, 14-36, 14-37, 14-39, 17-19
message type, 14-16
MessageParticle class, 14-24
MessageParticle class, 13-26, 14-16, 14-19, 14-20, 14-24, 14-33, 14-34
method item, 13-20, 13-20–13-22
Model, 1-1
  Delete, 2-46
  Update, 2-46
model
  block, G-2
  class, 3-15
  Compile Options, 3-5
  component, G-5
  computation, G-5
  Copyright, 3-4
  Description, 3-5
  Documentation, 3-5
  Domain, 3-4
  FSM, 13-1
  hierarchical, 13-1
  Import Libraries, 3-4
  instance, G-5
  Load Mode, 2-39, 3-4
  Logical Name, 3-3
  Model Type, 3-4
  primitive, 13-1
  Target, 3-4
  Version, 3-4
Model base, 1-3, 2-1
Model instance, 1-1
  add a, 1-13, 1-16
  connecting, 1-14, 1-17
model instance
  class, 3-15
  label, 3-15
  name, 3-15
  model of computation, G-5
  modeling
  behavioral, G-1
  functional, G-4
Module, 1-2
  create a, 1-11
  module, 13-1
Multi-valued Logic, 22-19
Multidimensional synchronous data flow, 6-5
MultiInSDFPort class, 13-24
multilink command, 9-3, 9-17
MultiOutSDFPort class, 13-25
multiple porthole, 3-6
MultiPortHole class, 13-24, 13-25, 13-26
MVImage class, 14-44
name
  model instance, 3-15
name item, 13-9
net, G-5
Network Building Set, 25-2
new function, 13-37, 13-38, 13-40
newds command, 9-3, 9-24
newdsmember command, 9-3, 9-24
newenum command, 9-4, 9-25
newenummember command, 9-4, 9-25
newevent command, 9-4, 9-23
newmemory command, 9-4, 9-22
newparam command, 9-4, 9-9
newquantity command, 9-4, 9-23
newserver command, 9-4, 9-23
newsystem command, 9-4, 9-6
newTrace function, 13-41
next function, 13-49, 13-52
node, G-5
node command, 9-4, 9-8
nodeconnect command, 9-4, 9-8
NOP primitive, 21-12
NotFoundException class, 15-2
NS2 domain, 25-1
numberPorts function, 13-26
numports command, 9-4, 9-10
numSimulEvents, 17-14
OCT, G-5
ofstream class, 13-41
OutDEPort class, 17-14
outmulti item, 13-15
output item, 13-15, 13-23
OutSDFMPHIter class, 13-25
OutSDFPort class, 13-22
OutSDFPorts class, 13-25
OutXXXPort class, 13-22
OverflowHandler, 14-10

Parameter, 1-1
actual, 1-2
add a, 1-15
create a, 1-21
export a, 1-18
formal, 1-1
link a, 1-15
Parameter Expressions, 3-18
parameter, 3-11
actual parameter, 3-11
definition of parameters, 3-11
direct linking, 3-20
ErrorHandle, 22-19
External parameters, 7-7
formal parameter, 3-11
indirect linking, 3-20
parameter linking, 3-20
Reading from File, 3-27
set, 7-11
system parameter, 3-11
target parameter, 3-11
Parameter set, 1-18
paramvalue command, 9-4, 9-18
particle, 3-7, 3-7
anynumber, 3-7
numeric, 3-7
type conversion, 3-7
Particle class, 13-22, 13-23, 13-24, 13-26,
14-1, 14-5, 14-16, 14-19, 14-36, 14-37, 14-43
permlink command, 9-4, 9-17
Physical View, 2-13
PN domain, G-6
Pointer class, 13-49, 13-51, 13-52
Pointer value() function, 13-52
Port, 1-1
actual, 1-1
add a, 1-12
create a, 1-21
formal, 1-1
port, 3-6, G-6
actual, 3-7
changing properties, 3-8
creation of, 3-8
formal, 3-7
multiple porthole, 3-6
particle, 3-7
single porthole, 3-6
terminal, 3-7
token, 3-7
porthole, 3-6
PortHole class, 13-22, 13-22
PortHoles class, 13-24
pragma command, 9-4, 9-16
pragmaDefaults command, 9-5, 9-16
Primitive, 1-2
Convert Ptolemy Stars, 2-44
create a, 1-20
source code, 1-1, 1-22
special, 1-13
primitive, 13-1
dynamic, 2-39, 3-4
source code, 1-1
primitive class
ArrayState, 13-28, 13-29
BarGraph, 13-42, 13-43
BaseImage, 14-44, 14-44
BDFPortHole, 17-6
Complex, 13-27, 13-28, 14-1, 14-24
ComplexArrayState, 13-27, 13-28
ComplexParticle, 13-26
ComplexState, 13-27
ComplexSubMatrix, 14-40
ComplexMatrix, 14-23, 14-24, 14-30
ComplexMatrix B = A.hermitian();, 14-30
ComplexMatrixParticle, 14-43
ComplexState, 13-27, 13-28
ConstTypeRef, 15-2
ConstTypeRef, 15-3
DataStructState, 13-27
DataStructClass, 15-11
DataStructMember, 15-6, 15-8, 15-10, 15-11
Index

DataStructParticle, 13-26
DataStructure, 15-8, 15-11
DataTypeException, 15-7
DCTImage, 14-44
DDFStar, 17-3
DERepeatStar, 17-15, 17-17
DEStar, 17-8, 17-15
DSFieldIter, 15-11
DsHandler, 15-10
DSMemberIter, 15-11
DynDFStar, 17-3
Envelope, 14-20
EnumState, 13-27
EnumType, 15-5
Envelope, 13-24, 14-16, 14-18, 14-19, 14-21, 14-22, 14-33, 14-37
Error, 13-39, 13-39
FileStream, 14-36
FileStream, 13-26
Fix, 14-1, 14-4–14-7, 14-10–14-15, 14-24
FixArrayState, 13-27, 13-28
FixParticle, 13-26
FixState, 13-27
FixSubMatrix, 14-40
FixArray, 14-5
FixMatrix, 14-23, 14-24, 14-26, 14-27
FixMatrixParticle, 14-43
FloatArrayState, 13-27, 13-28
FloatParticle, 14-16
FloatState, 13-27, 13-29
FloatSubMatrix, 14-40
FloatArrayState, 13-31
FloatMatrix, 14-23, 14-24, 14-27, 14-32, 14-33, 17-20
FloatMatrixParticle, 14-43
FloatParticle, 13-26, 14-32
FloatState, 13-27, 13-28
FloatVector, 15-5
GrayImage, 14-44
HashTable, 13-51, 13-51, 13-52
HashTableIter, 13-52
Histogram, 13-44, 13-44
histogram, 13-40
ifstream, 13-41
InDEPort, 17-13
IndexOutOfRangeException, 15-4
InfString, 13-48, 14-37, 14-37
InfString, 13-38, 13-46, 13-47, 13-48
InSDFMPHIter, 13-26
InSDFPort, 13-22
InSDFPorts, 13-25
IntArrayState, 13-27, 13-28
IntParticle, 14-16
IntState, 13-27
IntSubMatrix, 14-40
IntMatrix, 14-23, 14-24, 14-27
IntMatrixParticle, 14-43
IntParticle, 13-26
IntState, 13-27, 13-28
IntVecData, 14-19, 14-21
IntVector, 15-5
InXXXPort, 13-22
ListIter, 13-49
MatrixEnvParticle, 14-24
MatrixEnvParticle, 14-24, 14-34
Message, 13-24, 14-1, 14-16, 14-16–14-20, 14-22–14-24, 14-33, 14-36, 14-37, 14-39, 17-19
MessageParticle, 14-24
MessageParticle, 13-26, 14-16, 14-19, 14-19, 14-20, 14-24, 14-33, 14-34
MultiInSDFPort, 13-24
MultiOutSDFPort, 13-25
MultiPortHole, 13-24, 13-25, 13-26
MVImage, 14-44
NotFoundException, 15-2
ofstream, 13-41
OutDEPort, 17-14
OutSDFMPHIter, 13-25
OutSDFPort, 13-22
OutSDFPorts, 13-25
OutXXXPort, 13-22
Particle, 13-22, 13-23, 13-24, 13-26, 14-1, 14-5, 14-16, 14-19, 14-36, 14-37, 14-43
Pointer, 13-49, 13-51, 13-52
PortHole, 13-22, 13-22
PortHoles, 13-24
pt_ifstream, 13-40, 13-41
pt_ofstream, 13-37, 13-40, 13-40, 13-41
Index

PtMatrix, 14-23, 14-23, 14-24, 14-24, 14-32, 14-34, 14-40, 14-44
Queue, 13-49
SDFFix, 14-10
SequentialList, 13-49, 13-49
Stack, 13-49
State, 13-27, 17-19
StringArrayState, 13-27, 13-28
StringListIter, 13-48
StringMessage, 14-37
StringParticle, 14-37
StringState, 13-28
StringState, 13-28
TextTable, 13-51, 13-52
TextTableIter, 13-52
TypeDef, 15-2, 15-3, 15-8
Wormhole, 9-11
XGraph, 13-40, 13-41, 13-41
XHistogram, 13-44
primitive command, 9-5, 9-7
primitive item
  acknowledge, 13-11
  author, 13-11
  begin, 13-17, 13-18, 13-18, 13-22
  body, 13-21
  cccomment, 13-20
  cleanup, 13-18
  code, 13-20, 13-22
  codeblocks, 13-21
  conscalls, 13-17, 13-20
  constructor, 13-17, 13-22
  copyright, 13-11
  defevent, 13-13
  defmemory, 13-14
  defparameter, 13-12, 13-27
  defprimitive, 13-7, 13-8, 13-22
  defresource, 13-14
  defstate, 13-12
  derived, 13-10, 13-10
  derivedfrom, 13-10, 13-10
desc, 13-10
descriptor, 13-10
destructor, 13-17, 13-17, 13-22
domain, 13-9
directtime, 13-21, 13-21, 13-22
description, 13-11
header, 13-20, 13-22
hinclude, 13-20
html, 13-10, 13-11, 13-11
Import Libraries, 12-9
inline, 13-17–13-19, 13-21
inmulti, 13-15
inout, 13-15
input, 13-15, 13-23
location, 13-11
method, 13-20, 13-20–13-22
name, 13-9
outmulti, 13-15
output, 13-15, 13-23
private, 13-17, 13-19, 13-21, 13-22, 13-27
protected, 13-17, 13-19, 13-21, 13-22, 13-27
public, 13-17, 13-19, 13-21, 13-22, 13-27
public, 13-17
pure, 13-21
pure virtual, 13-21
state, 13-12, 13-13
static, 13-21
version, 13-10
virtual, 13-21
wrapup, 13-17, 13-19, 13-19, 13-22
primitive item
  descriptor, 13-13
primitive type
  Complex, 14-2, 14-2
  Fix, 14-4, 14-4, 14-5
  message, 14-16
Primitives
  WrapUp and Init, 17-17
print command, 9-5, 9-12
Print dialog, 2-47
printds command, 9-5, 9-26
printdsnames command, 9-5, 9-26
private item, 13-17, 13-19, 13-21, 13-22, 13-27
protected item, 13-17, 13-19, 13-21, 13-22, 13-27
ptifstream class, 13-40, 13-41
ptofstream class, 13-37, 13-40, 13-41
PTCL, 9-1
ptcl, 9-1, 16-1
ptcl command
  alias, 9-2, 9-8
  animation, 9-2, 9-14
  busconnect, 9-2, 9-8
  cancelAction, 9-2, 9-19
  cd, 9-2, 9-17
  connect, 9-2, 9-7
  cont, 9-2, 9-13
  cursystem, 9-2, 9-6
  curuniverse, 16-9
defmodule, 9-2, 9-10
delds, 9-2, 9-26
delnodes, 9-2, 9-15
delprimitive, 9-2, 9-14
delsystem, 9-2, 9-6
descriptor, 9-3, 9-12
disconnect, 9-3, 9-15
domain, 9-3, 9-7
domains, 9-3, 9-7
execute, 9-3, 9-12, 9-16
exit, 9-3, 9-18
halt, 9-3, 9-13
help, 9-3, 9-18
knownlist, 9-3, 9-11
link, 9-3, 9-17
listobjs, 9-3, 9-12
mathematica, 9-3, 9-19
matlab, 9-3, 9-19
multilink, 9-3, 9-17
newds, 9-3, 9-24
newdsmember, 9-3, 9-24
newenum, 9-4, 9-25
newenumer, 9-4, 9-25
newevent, 9-4, 9-23
newmemory, 9-4, 9-22
newparam, 9-4, 9-9
newquantity, 9-4, 9-23
newserver, 9-4, 9-23
newsystem, 9-4, 9-6
node, 9-4, 9-8
nodeconnect, 9-4, 9-8
numports, 9-4, 9-10
paramvalue, 9-4, 9-18
permLink, 9-4, 9-17
pragma, 9-4, 9-16
pragmaDefaults, 9-5, 9-16
primitive, 9-5, 9-7
print, 9-5, 9-12
printds, 9-5, 9-26
printdsnames, 9-5, 9-26
pwd, 9-17
registerAction, 9-5, 9-19
renamesystem, 9-5, 9-7
reset, 9-5, 9-14
run, 9-5, 9-13
schedtime, 9-5, 9-14
schedule, 9-5, 9-12
seed, 9-5, 9-16
setevent, 9-5, 9-23
setmemory, 9-5, 9-22
setparam, 9-5, 9-9, 13-31
setquantity, 9-5, 9-23
setserver, 9-6, 9-24
source, 9-6
stoptime, 9-6, 9-13
systemlist, 9-6, 9-6
target, 9-6, 9-15
targetSystem, 9-6, 9-15
targets, 9-6, 9-15
topblocks, 9-6, 9-18
wrapup, 9-6, 9-13
PtMatrix class, 14-23, 14-23, 14-24, 14-24, 14-32, 14-34, 14-40, 14-44
Ptolemy
  language, 1-1
Ptolemy interpreter, 9-1
public item, 13-17, 13-19, 13-21, 13-22, 13-27
public item, 13-17
pure item, 13-21
pure virtual item, 13-21
pwd command, 9-17
pxgraph, 7-14
INDEX

PXgraph Configuration, 8-4

Quantity Resources, 10-4
quantity resources, 10-8
Queue class, 13-49

Random Number Generation, 13-53
real time, G-6
refireAtTime, 17-15
registerAction command, 9-5, 9-19
Relation, 1-1
renamesystem command, 9-5, 9-7
ReportOverflow, 14-10
reset command, 9-5, 9-14

Resources
CPU Demo, 10-5
Quantity, 10-4
Server, 10-4
run command, 9-5, 9-13
run-time scheduling, G-6

Save
EPS, 8-6
Save As New Model dialog, 2-40
Save Simulation Results, 7-14
savestring function, 13-37
scalar classes
Complex, 14-2
Fix, 14-4
schedtime command, 9-5, 9-14
schedule command, 9-5, 9-12
Scheduler
DE Performance, 22-7
Scheduler class, 5-2
scheduling
compile-time, G-2
run-time, G-6
SDF Demos
Fixed-point, 18-49

SDF Domain
Channel Models, 18-33
Filters, 18-26
Image Processing, 18-37
Iterations., 18-3
Signal Processing, 18-26
Sources, 18-32
Spectral Analysis, 18-30
Targets, 18-4
Telecomm, 18-34
Transmitter Functions, 18-32
Vector Quantization, 18-29
SDF domain, 5-3, 6-2, 17-1, 18-1, G-6
end condition, 1-19
SDFFix class, 14-10
seed command, 9-5, 9-16
Segmentation Fault, C-1
Select color dialog, 3-35
Select Model dialog, 2-28, 3-15
Select Tool, 2-29, 3-15, 3-28, 3-34
SequentialList class, 13-49, 13-49
Server Resources, 10-4
setBDFParams function, 17-6
setevent command, 9-5, 9-23
setmemory command, 9-5, 9-22
setMode, 17-8
setparam command, 9-5, 9-9, 13-31
setquantity command, 9-5, 9-23
setSDFParams, 17-6
setSDFParams function, 13-15, 13-24, 17-1
setsserver command, 9-6, 9-24

Settings

Console View, 2-9
collision, 2-10
FSM Action Editor, 2-7
MLD, 2-3
Model Editor, 2-5
Primitive Source Editor, 2-6
Run Control, 2-8
Tree View, 2-3
Settings Dialog, 2-2
setup function, 13-40, 13-41
shared
events, 10-11
Shared Libraries, 2-50
Shared Model Elements, 3-36, 10-1
Memories, 10-1
Shared model elements, 1-2
Signal Processing
in SDF Domain, 18-26
simulated time, G-7
Simulation, 1-3, 7-1
Command line, 7-11
Generate C++, 7-3
Generate Extern, 7-2
Generate PTcl, 7-3
run a, 1-6
Statistics, 7-16
Tk primitives, 8-1
Simulation, G-7
parameter sets, 7-11
simulation domain, 5-1, G-7
Simulation Results
Save, 7-14
Write to Console, 7-14
Write to File, 7-14
single porthole, 3-6
sinMod Example
Step 1., 3-2
Step 2., 3-10
Step 3., 3-13
Step 4., 3-16
Step 5., 3-21
Step 6., 3-28
Step 7., 3-35
size function, 13-29
Sound
SDF Demos, 18-47
Source Code Editor, 3-45
source command, 9-6
Special Primitives
Creation, 2-41
SR Domain, 26-1
SR domain, 6-5
Stack class, 13-49
State class, 13-27, 17-19
state item, 13-12, 13-13
static item, 13-21
stop time, G-7
stoptime command, 9-6, 9-13
StringArrayState class, 13-27, 13-28
StringState class, 13-27
StringList class, 13-38, 13-46, 13-47–13-49,
14-18, 14-31, 14-37, 14-38
StringListIter class, 13-48
StringMessage class, 14-37, 14-37
StringParticle class, 14-37
StringState class, 13-28
Stringstate class, 13-27
sub-matrix classes
ComplexSubMatrix, 14-40
FixSubMatrix, 14-40
FloatSubMatrix, 14-40
IntSubMatrix, 14-40
supported domains, 6-2
Synchronous Data Flow, 6-2
synchronous data flow, G-7
Synchronous reactive domain, 6-5
System, 1-2
create a, 1-16
system parameter, 3-11
systemlist command, 9-6, 9-6
Target, 1-3
target, 9-15
target command, 9-6, 9-15
target parameter, 3-11
targetparam command, 9-6, 9-15
Targets, 6-11
default-SDF, 6-11
loop-SDF, 6-12
SDF Domain Targets, 6-11
SDF-to-PTcl, 6-13
targets command, 9-6, 9-15
Targets in SDF Domain, 18-4
Tcl, 16-1
tcl_file, 16-1
TclScript
Derived From, 16-1
tempFileName function, 13-37
terminal, 3-7
terminate function, 13-41
TextTable class, 13-51, 13-52
TextTableIter class, 13-52
tile mode, 2-11
timed domain, G-8
Timed Domains, 6-1
timestamp, G-7
Tk, 16-1
Tk primitives
Animation, 8-1
token, 3-7, G-8
tool button
Add Arc, 2-36
Index

Add Bus, 2-36, 3-32
Add Connection, 3-28
Add Event, 2-36
Add Initializable Delay, 2-36, 3-34
Add Input Port, 2-36, 3-8
Add Memory, 2-36
Add Model Instance, 2-36, 3-15, 3-15
Add Non-Initializable Delay, 2-36, 3-34
Add Output Port, 2-36, 3-8
Add Resource, 2-36
Add State, 2-36
Add Text Label, 2-36, 3-34
Cascade, 2-34
Compile Source, 2-37
Console Window On/Off, 2-33
Copy, 2-32
Cut, 2-32
Data Type View On/Off, 2-33
Delete, 2-32
Full, 2-34
New Model, 2-31, 2-37
Next, 2-34
Online Documentation, 2-32, 3-43
Open FSM, 2-37
Open Model, 2-32, 2-37
Open Source, 2-37
Pan Tool, 2-35
Paste, 2-32
Previous, 2-33
Print Model, 2-32
Property Editor On/Off, 2-33
Redo, 2-33
Reload Model, 2-32
Save All Models, 2-32
Save Model, 2-32
Select Tool, 2-29, 2-35, 3-15, 3-28, 3-34
Show FSM Models, 2-35
Show HTML Files, 2-35
Show Libraries, 2-34
Show Modules, 2-34
Show Primitives, 2-35
Show Probes, 2-35
Show Systems, 2-34
Switch to Simulation Mode, 2-37
Tile, 2-34
Tree View On/Off, 2-33
Undo, 2-33
Zoom In, 2-33
Zoom Out, 2-33
Zoom To Fit, 2-33
Zoom Tool, 2-35
toolbar
editor, 2-35
standard, 2-31
Tree View, 2-34
topblocks command, 9-6, 9-18
Tree View, 2-11
context menu, 2-16
filter, 2-34
Tree view
file, 1-5
library, 1-3
Tutorials
DynamicInstance, 3-38
DynamicLinking, 3-39
typeCheck, 14-19
typeError, 14-19
TypeRef class, 15-2, 15-3, 15-8
Unsupported Domains
Code Generation, 27-1
SR Domain, 26-1
unsupported domains, 6-5
Update model dialog, 2-46
Used Computers, 7-15
version item, 13-10
VHDL, 6-6
VHDL domain, 6-6
VHDLB, 6-6
VHDLB domain, 6-6, G-8
VHDLF domain, G-8
view menu, 2-29
Refresh, 2-30
Zoom In, 2-29
Zoom Out, 2-30
Zoom To Fit, 2-30
virtual item, 13-21
window menu, 2-30
Cascade, 2-31
Close, 2-30
Close All, 2-30
Index

Full, 2-31
Next, 2-30
Previous, 2-31
Tile, 2-31
wormhole, 5-3, G-8
Wormhole class, 5-3, 9-11
Wormholes
  DE in SDF, 22-5
  SDF in DE, 22-4
wrapup command, 9-6, 9-13
wrapup function, 13-34
wrapup item, 13-17, 13-19, 13-19, 13-22
WrapUp Primitive, 17-17
writableCopy, 14-19

Xgraph
  end condition, 1-19
  example, 1-19
XGraph class, 13-40, 13-41, 13-41
Xgraph Configuration, 8-4
XHistogram class, 13-44