



MIRABILIS DESIGN INC.

Advanced Modeling Guide

VisualSim Documentation



©2003-2016 Mirabilis Design Inc. All rights reserved.

The information contained herein is subject to change without notice. While every reasonable effort was made to ensure the completeness and correctness of this document, Mirabilis Design Inc. makes no warranty of any kind with regard to this material, including but not limited to any implied warranties. Mirabilis Design Inc. shall not be liable for errors or omissions contained herein or for any damages relating to the use of this material.

VisualSim and VisualSim Architect are registered trademark of Mirabilis Design Inc.

Java and all Java-related titles are trademarks or registered trademarks of Sun Microsystems in the United States and other countries. All other brand or product names may be trademarks of their respective holders.

This document is protected by US and International copyright laws. No part of this document may be reproduced in any manner without prior written consent of Mirabilis Design Inc.

Mirabilis Design Inc.
1159 Sonora Ct, Suite 116
Sunnyvale, CA 94086

Table of Contents

Table of Contents	3
Chapter 1: Simulators	14
1 Discrete-Event Simulator.....	15
1.1 Introduction	15
1.1.1 Model Time	15
1.1.2 Simultaneous events	15
1.1.3 Iteration	17
1.1.4 Getting a Model Started	17
1.1.5 Pure Events at the Current Time	17
1.1.6 Stopping Execution.....	18
1.2 Writing DE Blocks	18
1.2.1 General Guidelines	18
1.2.2 Examples Simplified Delay Block.....	19
1.2.3 Thread Blocks	22
1.3 Composing DE with Other Simulators	24
1.3.1 DE inside another Simulator	24
1.3.2 Another Simulator inside DE	26
2 CT Simulator.....	27
2.1 Introduction	27
2.1.1 System Specification	28
2.1.2 Time	30
2.2 Solving ODEs numerically	30
2.2.1 Basic Notations.....	31
2.2.2 Fixed-Point Behavior	31
2.2.3 ODE Solvers Implemented.....	32
2.2.4 Breakpoint ODE Solvers	33
2.3 Signal Types	33
2.4 CT Blocks	35
2.4.1 CT Block Interfaces	35
2.5 CT Simulators	36
2.5.1 ODE Solvers	36
2.5.2 CT Simulator Parameters.....	36
2.5.3 CTMultiSolverDirector	37
2.5.4 CTMixedSignalDirector	38
2.5.5 CTEmbeddedSimulator	38
2.6 Interacting with Other Simulators.....	38

2.7	Mixed-Signal Execution	39
2.7.1	Hybrid System Execution	40
2.8	Appendix F: Brief Mathematical Background	41
3	Untimed Digital or Synchronous Data Flow Simulator.....	42
3.1	Purpose of the Simulator	42
3.2	Using SDF	42
3.2.1	Deadlock	42
3.2.2	Consistency of data rates	43
3.2.3	How many iterations?	44
3.2.4	Granularity	44
3.3	Properties of the SDF simulator	45
3.3.1	Scheduling	46
3.3.2	Hierarchical Scheduling.....	47
3.3.3	Hierarchically Heterogeneous Models	48
4	FSM Simulator	49
4.1	Introduction	49
4.2	Building FSMs in ModelBuilder	49
4.2.1	Alternate Mark Inversion Coder	49
4.3	The Implementation of FSMActor	51
4.3.1	Guard Expressions	51
4.3.2	Actions	52
4.4	FSM-Hierarchical.....	53
4.4.1	A Schmidt Trigger Example	53
4.4.2	Applications.....	55
	Chapter 2 Modeling Libraries	56
	Introduction to Resources.....	56
	Active Resources	56
	Event Queue Blocks	57
	Timed Queue Resource Blocks.....	57
	Server_N_Priority	58
	System Resource blocks(a.k.a Scheduler Blocks)	58
	Channel Blocks	60
	Queues (a.k.a Smart_Resource).....	61
	Server (a.k.a Smart_Timed_Resource)	62
	Passive or Quantity-Shared Resource Blocks	63

Virtual Connection Blocks	64
Architecture Modeling Toolkit	66
Cache	68
Block Description.....	68
Block Usage.....	68
Functionality.....	68
Computations	69
State Plots	69
Statistics.....	69
Configuration of Parameters	70
Memory	71
Block Description.....	71
Block Usage.....	71
Functionality.....	71
Computations	72
State Plots	72
Configuration of Parameters	73
Processor Block	75
Description.....	75
Block Usage.....	75
Configuration of Parameters	80
Instruction Stack.....	83
Linear State Machine	83
Processing Flow	85
More on Processor Flow	86
Instruction_Set	88
Description.....	88
Architecture Setup	91
Description.....	91
Configuration of Parameters	92
Routing_Table	94
Processor Model Features	104
Cache and Memory Overview	108
Cache Thread.....	109
Cache Misses	109
Cache Instrs, Reads, Writes	110
DRAM Memory	110
DRAM Processing	110
External Bus	110
Cache Summary.....	110
Bus, Switch and Controller Toolkit	112

Bus Arbiter	112
Block Description	112
Block Usage	112
Functionality	112
Statistics	114
State Plots	114
Configuration of Parameters	115
Bus Interface	116
Block Description	116
Block Usage	116
Functionality	116
Configuration of Parameters	117
DMA_Controller	117
Block Description	117
Functionality	118
Routing Functionality	120
Example of DMA in a Model	121
Statistics	121
Parameters	122
Request Acknowledge Node/ Asynchronous Bus	123
Introduction	123
Block Configure Parameters	125
Data Structure: Processor_DS	126
Acknowledge Port to Data-In Port Block Commands	127
Routing Table	129
Bus Statistics	129
Power Modeling Toolkit	132
Introduction	132
How it works	133
Block Level Parameters	133
Block Ports	134
Block Methods	134
Power Utilities	136
Traffic Profile Parameters	139
Power Finite State Machine: Active, Standby, Suspend, Off States	140
Resources Modal FSMs Parameters	142
Execution Parameters	144
Power Generator Parameters	148
Reports Processing	149
Bus and Interface Standards	150

1	AMBA Buses.....	150
1.1	AMBA AHB.....	150
1.1.1	Sample Model.....	150
1.1.2	Setup.....	150
1.1.3	Model Parameters.....	150
1.1.4	Initialize the Processing Data Structure.....	150
1.1.5	A Typical AMBA Bus System Architecture.....	151
1.1.6	AMBA Bus Features supported:.....	152
1.1.7	Typical AMBA System components and the Mapping to VisualSim 152	
1.1.8	Routing Table.....	152
1.1.9	Routing Table for the Sample Model.....	153
1.1.10	Bus Statistics.....	153
1.1.11	Statistics of the Sample Model.....	153
1.1.12	Validation comparing with AMBA Spec.....	154
1.2	AMBA APB Bus.....	163
1.3	AMBA AXI.....	164
1.3.1	Setup.....	164
1.3.2	Sample Model.....	165
1.3.3	Setup.....	165
1.3.4	Model Parameters, Data Structure Fields and Ports.....	165
1.3.5	Data Structure Field.....	167
1.3.6	Explanation.....	167
1.3.7	Master and Slave Queue Depths.....	169
1.3.8	Read and Write Request Channels.....	170
1.3.9	Arbitration.....	170
1.3.10	Throttle Mechanism and Throttle block.....	170
1.3.11	Adding additional Master and Slave port.....	172
1.3.12	AXI Bus Description.....	173
1.3.13	Flow Diagram.....	175
1.3.14	Bus Statistics.....	177
1.3.15	Latency Flow.....	178
1.3.16	Operational View of the Model.....	179
2	PCI Family of Buses.....	181
2.1	PCI and PCI-X Bus.....	181
2.1.1	Sample Model.....	181
2.1.2	Setup.....	181
2.1.3	Model Parameters.....	181
2.1.4	Initialize the Processing Data Structure.....	182
2.1.5	A typical PCI, PCI-X system.....	183
2.1.6	Routing Table.....	184
2.1.7	Bus Statistics.....	184
2.1.8	Statistics Validation comparing with PCI Specification.....	185

2.1.9	Operational View of the Model	186
2.1.10	Preemption while stepping in progress	189
2.2	PCI-Express	190
2.2.1	Sample Model	190
2.2.2	Setup	190
2.2.3	Model Parameters, Data Structure Fields and Ports	191
2.2.4	Traffic Queue Depths	193
2.2.5	PCIe Bus Description	193
2.2.6	Bus Statistics	194
2.2.7	Latency Flow	195
2.2.8	Operational View of the Model	196
3	CoreConnect Bus	198
3.1.1	Introduction	198
3.1.2	Setup	198
3.1.3	Model Parameters	198
3.1.4	Data Structure: Processor_DS	199
3.1.5	Timing Diagrams	200
3.1.6	Routing Table	209
3.1.7	Bus Statistics	210
3.1.8	Operational View of the Model	212
4	Switched Ethernet	214
4.1.1	Introduction	214
4.1.2	Model Description	214
4.1.3	Sample Model	214
4.1.4	Setup	215
4.1.5	Initialize the Processing Data Structure	215
4.1.6	A Typical Ethernet Model	215
4.1.7	FULL DUPLEX OPERATION	216
4.1.8	Operational View of the Model	216
5	SpaceWire	218
5.1.1	Introduction	218
5.1.2	Model Setup	218
5.1.3	Model Parameter	219
5.1.4	SpaceWire Node	220
5.1.5	SpaceWire Link	221
5.1.6	SpaceWire Router	222
5.1.7	SpaceWire Description	223
5.1.8	Statistics	225
6	Rapid IO	227
6.1	Introduction	227
6.2	Rapid IO Blockset	227

6.3	RIO_Node Block Configuration Parameters	228
6.4	Connecting the RIO_Node Block in a model.....	229
6.4.1	RIO_Node Block added to a Model	229
6.5	Serial Switch Block Configuration Parameters	233
6.6	Connecting the Serial_Switch Block in a model	234
6.6.1	Serial_Switch Block added to a Model	235
6.7	Data Structure: Processor_DS	235
7	Ethernet Audio Video Bridging	237
7.1	Library	237
7.2	Tutorial System	239
8	Fibre Channel	240
8.1	Introduction to Fibre Channel	240
	• Point-to-point (FC-P2P).	240
	• Arbitrated loop (FC-AL).	240
	• Switched fabric (FC-SW).	240
8.2	About VisualSim Fibre Channel Library Package	240
8.3	Library Blocks	241
8.4	FC_N_Port	241
8.4.1	Flow Diagram	241
8.4.2	Data Structure Fields	241
8.4.3	Parameters	242
8.5	Fibre Channel Switch	242
8.5.1	Flow Diagram	243
8.5.2	Flow Control	243
8.5.3	Data Structure Fields	243
8.5.4	Parameters	243
8.6	FC_Link	244
8.6.1	Parameters	244
8.7	FC_Config	244
8.7.1	Parameters	244
8.8	Tutorial	245
8.8.1	Basic Rules	246
8.8.2	Construction Steps	246
9	TTEthernet	253
9.1	Introduction	253
9.2	About TTEthernet Library	253

9.3	Synchronization	255
9.4	System Level Model.....	255
9.5	Model Parameters	256
9.6	TTEthernet Node	257
9.7	TTE Bridge	257
9.8	TTE Config	258
9.9	TTE_Setup.....	258
9.10	TTE_Stats	259
9.11	TTE_Traffic.....	259
9.12	Tutorial	260
9.12.1	Basic Rules	261
9.12.2	Construction Steps	261
9.13	Advanced Tutorial.....	270
9.13.1	TTEthernet model with 8 Source Nodes.....	270
9.13.2	TTEthernet Model with 8 nodes and 3 Destination Nodes.....	271
9.13.3	TTEthernet Model with 8 Nodes, 2 Bridges and 3 Destination Nodes	271
10	IEEE1394/Firewire	273
10.1	Introduction	273
10.2	About FireWire	273
10.3	About FireWire Library.....	273
10.4	Model Parameters	274
10.5	Assumptions.....	274
10.6	FireWire Node.....	274
10.6.1	Flow Diagram for Isochronous Transfers	275
10.6.2	Flow Diagram for Asynchronous Transfers	276
10.6.3	Block Details	276
10.6.4	Block Parameters	278
10.7	FireWire Link.....	278
10.7.1	Block Parameters	278
10.8	FireWire Config	279
10.8.1	Block Parameters	279
10.9	Reports	279
10.10	Example	279
10.10.1	Basic Rules	280
10.10.2	Construction Steps.....	280

10.11	Understanding Common Errors	285
	Application and Algorithm Library	286
1	Networking	286
2	Wireless and Sensor Network System	289
	Introduction.....	289
	Installation and Quick Start	289
	Modeling Wireless Networks	289
	Running a Pre-Built Model	289
	Changing Parameters	290
	Structure of a Pre-Built Model	291
	Visual Representations (Icons)	291
	Channels.....	293
	Wireless Hierarchical Blocks.....	294
	Controlling the Execution	295
	Building a New Model	296
	Using the Plot Blocks	305
	Modeling Capabilities	305
	Channel Models.....	306
	Wireless Node Models	306
	Examples of Modeling Capabilities	307
	Packet Structure	307
	Packet Losses	307
	Battery Power	307
	Power Loss	307
	Collisions	307
	Transmit Antenna Gain	309
	Algorithmic	314
I	Analog.....	314
	Event Generator	314
	Waveform generators	314
	Control-Analog Functions	314
II	Control Systems.....	316
	Event Generator	316
	Waveform generators	316

Control-Analog Functions	316
II Petri Net	317
III Image Processing	317
Basic	317
Advanced (Using Java Advanced Imaging)	318
Media Interfaces (Using Java Media Framework).....	318
IV Signal Processing	319
Sources	319
Audio.....	319
Communications	319
Statistical.....	320
Filtering.....	320
Spectrum	321
VisualSim Custom Development.....	322
1 Custom-Coded Blocks using Java.....	322
1.1 Overview.....	322
1.2 Anatomy of an Block	322
1.2.1 Ports	323
1.2.2 Port Rates and Dependencies between Ports	329
1.2.3 Parameters	330
1.2.4 Constructors.....	331
1.2.5 Cloning.....	331
1.3 Action Methods	333
1.3.1 Initialization	333
1.3.2 Prefire.....	334
1.3.3 Fire	335
1.3.4 Postfire.....	336
1.3.5 Wrapup.....	338
1.4 Coupled Port and Parameter	339
1.5 Iterate Method.....	341
1.6 Time	341
1.7 Icons	342
1.7.1 New method	343
1.7.2 Old Method	343
1.8 Code Format	344

1.8.1	Indentation	345
1.8.2	Spaces	346
1.8.3	Comments	346
1.8.4	Names	346
1.8.5	Exceptions	347
1.8.6	Javadoc	347
1.8.7	Code Organization	349
1.9	Java Block Template	349
1.10	Debugging	350
2	Adding a Block to ModelBuilder Library List	355
2.1	Edit Ramp2.java and change:	355
2.2	Library Palette Addition	355
2.3	Testing Addition	356
2.4	Adding a new palette	356
2.5	Hints on Adding Blocks	357
2.5.1	Icon Display	357
2.5.2	Parameter Pull-Down	357
3	Creating Applets	358
3.1	Introduction	358
3.1.1	HTML Files Containing Applets	358
3.2	Java Script for embedding the VisualSim Model	359
3.2.1	Using JavaScript in files	363
4	Templates	364



Chapter 1: Simulators

The simulators implement various models of computation. Most of these models of computation can be viewed as a framework for component-based design, where the framework defines the interaction mechanism between the components. VisualSim consists of four major simulators-

- Continuous Time
- Synchronous Data Flow
- Discrete Event
- Finite State Machine

The first three simulators implement their own scheduling between blocks and do not rely on threads. These usually results in a highly efficient execution. The FSM simulator is in a category by itself, since the components are not producers and consumers of data, but rather are states

1 Discrete-Event Simulator

1.1 Introduction

The discrete-event (DE) simulator supports time-oriented modeling such as queuing systems, communication networks, and digital hardware. In this simulator, blocks communicate by sending events, where an event is a data value (a token) and a time stamp. A DE scheduler ensures that events are processed chronologically according to this time stamp by firing those blocks whose available input

events are the oldest (having the earliest time stamp of all pending events).

A key strength in the VisualSim implementation is that simultaneous events (those with identical time stamps) are handled systematically and deterministically. Another key strength is that the global event queue uses an efficient structure that minimizes the overhead associated with maintaining a sorted list with a large number of events.

1.1.1 Model Time

In the DE model of computation, time is global, in the sense that all blocks share the same global time. The current time of the model is often called the model time or simulation time to avoid confusion with current real time. As in most VisualSim Simulators, blocks communicate by sending tokens through ports. Ports can be input ports, output ports, or both. Tokens are sent by an output port and received by all input ports connected to the output port through relations. When a token is sent from an output port, it is packaged as an event and stored in a global event queue. By default, the time stamp of an output is the model time, although specialized DE blocks can produce events with future time stamps. Blocks may also request that they be fired now, or at some time in the future, by calling the `fireAt-CurrentTime()`, `fireAt()`, or `fireAtRelativeTime()`, methods of the Simulator. Each of these places a pure event (one with a time stamp, but no data) on the event queue. A pure event can be thought of as setting an alarm clock to be awakened in the future. Sources (blocks with no inputs) are thus able to be fired despite having no inputs to trigger a firing. Moreover, blocks that introduce delay (outputs have larger time stamps than the inputs) can use this mechanism to schedule a firing in the future to produce an output. The `fireAtCurrentTime()` method provides a mechanism for achieving a zero delay by atomically getting the current model time and queuing an event with that time stamp. This permits I/O blocks to have themselves fired in real-time whenever data arrives at a physical I/O port. In the global event queue, events are sorted based on their time stamps. An event is removed from the global event queue when the model time reaches its time stamp, and if it has a data token, then that token is put into the destination input port.

At any point in the execution of a model, the events stored in the global event queue have time stamps greater than or equal to the model time. The DE Simulator is responsible for advancing (i.e. incrementing) the model time when all events with time stamps equal to the current model time have been processed (i.e. the global event queue only contains events with time stamps strictly greater than the current time). The current time is advanced to the smallest time stamp of all events in the global event queue.

1.1.2 Simultaneous events

An important aspect of a DE simulator is the prioritizing of simultaneous events. This gives the simulator a dataflow-like behavior for events with identical time stamps. It is done by assigning a depth to each block and a micro step to each phase of execution within a given time stamp. Each depth is a non-negative integer, uniquely assigned; i.e. no two blocks are assigned the same depth.

The depth of a block determines the priority of events destined to that block, relative to other events with the same time stamp and the same micro step. The highest priority events are those destined to blocks with the lowest depth.

Consider the simple topology shown in Figure 1. Assume that block Y is not a delay block, meaning that its output events have the same time stamp and micro step as its input events (this is suggested by the dotted arrow). Suppose that block X produces an event with time stamp. That event is available at ports B and D, so the scheduler could choose to fire blocks Y or Z. Which should it fire? Intuition tells us it should fire the upstream one first, Y, because that firing may produce another event with time stamp at port D (which is presumably a multiport). It seems logical that if block Z is going to get one event on each input channel with the same time stamp, then it should see those events in the same firing. Thus, if there are simultaneous events at B and D, then the one at B will have higher priority.

The depths are determined by a topological sort of a directed acyclic graph (DAG) of the blocks. The DAG of blocks follows the topology of the graph, except when there are declared delays. Once the DAG is constructed, it is sorted topologically. This simply means that an ordering of blocks is assigned

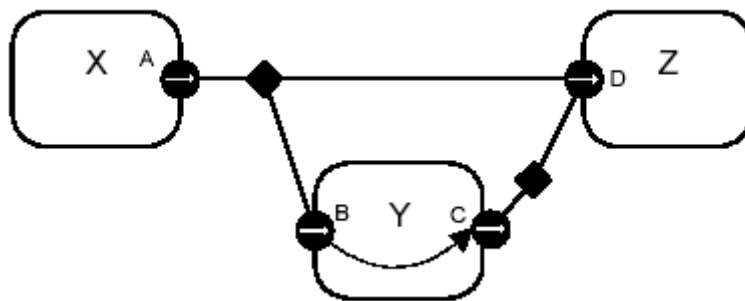


Figure 1 If there are simultaneous events at B and D, then the one at B will have higher priority because it may trigger another simultaneous event at D.

such that an upstream block in the DAG is earlier in the ordering than a downstream block. The depth of a block is defined to be its position in this topological sort, starting with zero. For example, in Figure 1, X will have depth 0, Y will have depth 1, and Z will have depth 2. In general, a DAG has several correct topological sorts. The topological sort is not unique, meaning that the depths assigned to blocks are somewhat arbitrary. But an upstream block will always have a lower depth than a downstream block, unless there is an intervening delay block. Thus, given simultaneous input events with the same micro step, an upstream block will always fire before a downstream block. Such a strategy ensures that the execution is *deterministic*, assuming the blocks only communicate via events. In other words, even though there are several possible choices that a scheduler could

make for an ordering of firings, all choices that respect the priorities yield the same results.

There are situations where constructing a DAG following the topology is not possible. Consider the topology shown in Figure 2. It is evident from the Figure that the topology is not acyclic. Indeed, Figure 2 depicts a *zero-delay loop* where topological sort cannot be done. The Simulator will refuse to run the model, and will terminate with an error message.

The TimedDelay block in DE is a simulator-specific block that asserts a delay relationship between its input and output. Thus, if we insert a TimedDelay block in the loop, as shown in Figure 3, then constructing the DAG becomes once again possible. The TimedDelay block breaks the precedence.

Note in particular that the TimedDelay block breaks the precedence even if its delay parameter is set to zero. Thus, the DE simulator is perfectly capable of modeling feedback loops with zero time delay, but the model builder has to specify the order in which events should be processed by placing a Timed-Delay block with a zero value for its parameter. When modeling multiple zero-delay feedback paths, simultaneity of the fed back signals is modeled by having the same number of TimedDelay blocks in each feedback path.

1.1.3 Iteration

At each iteration, after advancing the current time, the Simulator chooses all events in the global event queue that have the smallest time stamps, micro step, and depth (tested in that order). The chosen events are then removed from the global event queue and their data tokens are inserted into the appropriate

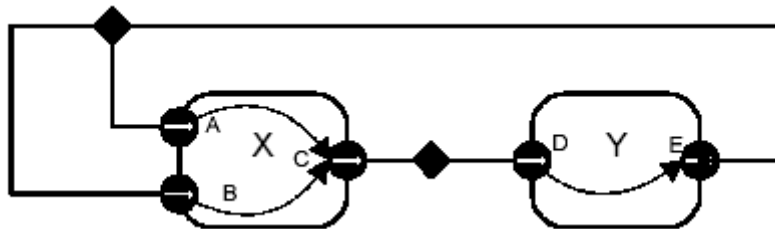


Figure 2 Example of a directed zero-delay loop.

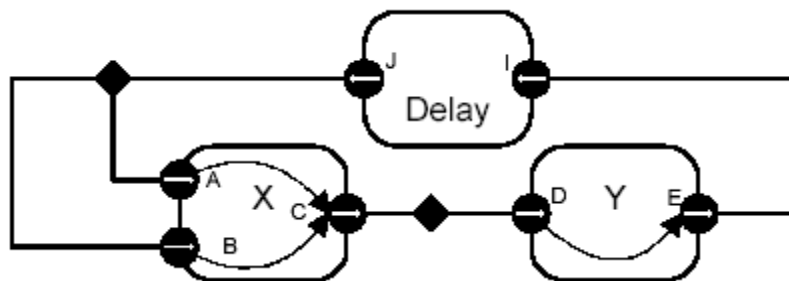


Figure 3 A Delay block can be used to break a zero-delay loop

input ports of the destination block. Then, the Simulator iterates the destination block; i.e. it invokes `prefire()`, `fire()`, and `postfire()`. All of these events are destined to the same block, since the depth is unique for each block.

A firing may produce additional events at the current model time (the block reacts instantaneously, or has zero delay). There also may be other events with time stamp equal to the current model time still pending on the event queue. The DE Simulator repeats the above procedure until there are no more events with time stamp equal to the current time. This concludes a single iteration of the model. Iteration processes all events on the event queue with the smallest time stamp.

1.1.4 Getting a Model Started

Before one of the iterations described above can be run, there have to be initial events in the global event queue. Blocks may produce initial pure events or regular output events in their `initialize()` method. Thus, to get a model started, at least one block must produce events. All the blocks described in the Block Libraries chapter that produce pure events can be used in DE. We can define the start time to be the smallest time stamp of these initial events.

1.1.5 Pure Events at the Current Time

A block calls `fireAt()` to schedule a pure event. The pure event is a request to the scheduler to fire the block sometime in the future. However, the block may choose to call `fireAt()` with the time argument equal to the current time. In fact, the preferred method for simulator-polymorphic source blocks to get started is to have code like the following in their `initialize()` method:

```
Director director = getDirector();
director.fireAt(this, director.getCurrentTime());
```

This will schedule a pure event on the event queue with micro step zero and depth equal to that of the calling block.

A block may also call `fireAt()` with the current time in its `fire()` method. This is a request to be refired later in the current iteration. This is managed by queuing a pure event with micro step one greater than the current micro step. In fact, this is the only situation in which the micro step is incremented beyond zero.

A pure event at the current time can also be scheduled by code like the following:

```
Director director = getDirector();
director.fireAtCurrentTime(this);
```

This code is equivalent to the previous example when used within standard block methods like `initialize()` and `fire()`. This is because the Simulator never advances model time while a block is being initialized or fired. However, when methods such as I/O callbacks queue events at the current time, they need to use the latter code. This is because the Simulator runs in a separate thread from the callback and, in the former code, will occasionally advance the model time between the call to `getCurrentTime()` and the call to `fireAt()`.

1.1.6 Stopping Execution

Execution stops when one of these conditions becomes true:

- The current time reaches the stop time, set by calling the `setStopTime()` method of the DE Simulator.
- The global event queue becomes empty and the `stopWhenQueueIsEmpty` parameter of the Simulator is true.

Events at the stop time are processed before stopping the model execution. The execution ends by calling the `wrapup()` method of all blocks. `Wrapup()` is called even when execution has been stopped due to an exception. Therefore, throwing an exception in the `wrapup()` method of a block is not recommended as this exception will mask the original exception, making the source of the original exception difficult to locate.

It is also possible to explicitly invoke `iterate()` method of the manager for some fixed number of iterations. Recall that an iteration processes all events with a given time stamp, so this will run the model through a specified number of discrete time steps.

Note that a block can prevent execution from stopping properly if it blocks in its `fire()` method. A block which blocks in `fire()` should have a `stopFire()` method which, when called, notifies the `fire()` method to cease blocking and return.

1.2 Writing DE Blocks

It is very common in DE modeling to include custom-built blocks. For the most part, writing blocks for the DE simulator is no different than writing blocks for any other simulator. Some blocks, however, need to exercise particular control over time stamps and block priorities. Such blocks use instances of `DEIOPort` rather than `TypedIOPort`.

The first section below gives general guidelines for writing DE blocks and simulator-polymorphic blocks that work in DE. The second section explains in detail the priorities, and in particular, how to write blocks that declares delays. The final section discusses blocks that operate as a Java thread.

1.2.1 General Guidelines

The points to keep in mind are:

- When a block fires, not all ports have tokens, and some ports may have more than one token. The time stamps of the events that contained these tokens are no longer explicitly available. The current model time is assumed to be the time stamp of the events.
- If the block leaves unconsumed tokens on its input ports, then it will be iterated again before model time is advanced. This ensures that the current model time is in fact the time stamp of the input events. However, occasionally, a block will want to leave

unconsumed tokens on its input ports, and not be fired again until there is some other new event to be processed. To get this behavior, it should return false from `prefire()`. This indicates to the DE Simulator that it does not wish to be iterated.

- If the block returns false from `postfire()`, then the Simulator will not fire that block again. Events that are destined for that block are discarded.
- When a block produces an output token, the time stamp for the output event is taken to be the current model time. If the block wishes to produce an event at a future model time, one way to accomplish this is to call the Simulator's `fireAt()` method to schedule a future firing, and then to produce the token at that time. A second way to accomplish this is to use instances of `DEIOPort` and use the overloaded `send()` or `broadcast()` methods that take a time delay argument.
- If an block contains a callback method or a private thread, and this callback or thread wishes to produce an event now or at a future model time, then a reliable way to achieve this is to call either the `fireAt-CurrentTime()` method or the `fireAtRelativeTime()` method. These methods may safely be called asynchronously, yielding real-time liveness. By contrast, `fireAt()` must be called from within a standard block method.
- The `DEIOPort` class can produce events in the future, but there is an important subtlety with using these methods. Once an event has been produced, it cannot be retracted. In particular, even if the block which produced the event (or the destination block of the event) is deleted before model time reaches that of the future event, the event will be delivered to the destination. If you use `fireAt()`, `fireAtCurrentTime()`, or `fireAtRelativeTime()` instead to generate delayed events, then if the block is deleted (or returns false from `postfire()`) before the future event, then the future event will not be produced.
- By convention in `VisualSim`, blocks update their state only in the `postfire()` method. In DE, the `fire()` method is only invoked once per iteration, so there is no particular reason to stick to this convention.

Nonetheless, Mirabilis Design recommends that this methodology is adopted to make the custom-block useful in other simulators. The simplest way to ensure this is to follow the following pattern. For each state variable, such as a private variable named `_count`,

```
private int _count;
```

Create a shadow variable

```
private int _countShadow;
```

Then write the methods as follows:

```
public void fire() {
    _countShadow = _count;
    ... perform some computation that may modify _countShadow ...
}
public boolean postfire() {
    _count = _countShadow;
    return super.postfire();
}
```

This ensures that the state is updated only in `postfire()`.

In a similar fashion, delayed outputs (produced by either mechanism) should be produced only in the `postfire()` method, since delayed outputs are persistent states. Thus, `fireAt()` should be called in `postfire()` only, as should the overloaded `send()` and `broadcast()` of `DEIOPort`.

1.2.2 Examples Simplified Delay Block.

An example of a simulator-specific block for DE is shown in Figure 4. This block delays input events by some amount specified by a parameter. The simulator-specific features of the block are shown in bold. They are:

- It uses `DEIOPort` rather than `TypedIOPort`.

- It has the statement:


```
input. delayTo( output);
```

 This statement declares to the Simulator that this block implements a delay from input to output. The block uses this to break the precedence's when constructing the DAG to find priorities.
- It uses an overloaded send() method, which takes a delay argument, to produce the output. Notice that the output is produced in the postfire() method, since by convention in VisualSim , persistent state is not updated in the fire() method, but rather is updated in the postfire() method.

Server Block: The Server block in the DE library uses a rich set of behavioral properties of the DE simulator. A server is a process that takes some amount of time to serve "customers."

While it is serving a customer, other arriving customers have to wait. This block can have a fixed service time (set via the parameter serviceTime, or a variable service time, provided via the input port newServiceTime). A typical use would be to supply random numbers to the newServiceTime port to generate random service times. These times can be provided at the same time as arriving customers to get an effect where each customer experiences a different, randomly selected service time.

The compacted code is shown in Figure 5. This block extends DETransformer, which has two public members, input and output, both instances of DEIOPort. The constructor makes use of the delayTo() method of these ports to indicate that the block introduces delay between its inputs and its output.

The block keeps track of the time at which it will next be free in the private variable `_nextTimeFree`. This is initialized to minus infinity to indicate that whenever the model begins executing, the server is free. The `prefire()` method determines whether the server is free by comparing this private variable against the current model time. If it is free, then this method returns true, indicating to the scheduler that it can proceed with firing the block. If the server is not free, then the `prefire()` method checks to see whether there is a pending input, and if there is, requests a firing when the block will become free. It then returns false, indicating to the scheduler that it does not wish to be fired at this time. Note that the `prefire()` method uses the methods `getCurrentTime()` and `fireAt()` of DEActor, which are simply convenient interfaces to methods of the same name in the Simulator.

The `fire()` method is invoked only if the server is free. It first checks to see whether the `newServiceTime` port is connected to anything, and if it is, whether it has a token. If it does, the token is read and used to update the `serviceTime` parameter. No more than one token is read, even if there are more in the input port, in case one token is being provided per pending customer.

The `fire()` method then continues by reading an input token, if there is one, and updating `_nextTimeFree`. The input token that is read is stored temporarily in the private variable `_currentInput`.

The `postfire()` method then produces this token on the output port, with an appropriate delay. This is done in the `postfire()` method rather than the `fire()` method in keeping with the policy in VisualSim that persistent state is not updated in the `fire()` method. Since the output is produced with a future time stamp, then it is persistent state.

Note that when the block will not get input tokens that are available in the `fire()` method, it is essential

```
package VisualSim.Simulators.de.lib.test;
import VisualSim.actor.TypedAtomicActor;
import VisualSim.Simulators.de.kernel.DEIOPort;
import VisualSim.data.DoubleToken;
import VisualSim.data.Token;
import VisualSim.data.expr.Parameter;
import VisualSim.actor.TypedCompositeActor;
import VisualSim.kernel.util.IllegalActionException;
```



```

import VisualSim.kernel.util.NameDuplicationException;
import VisualSim.kernel.util.Workspace;
public class SimpleDelay extends TypedAtomicActor {
public SimpleDelay( TypedCompositeActor container, String name)
throws NameDuplicationException, IllegalActionException {
super( container, name);
input = new DEIOPort( this, "input", true, false);
output = new DEIOPort( this, "output", false, true);
delay = new Parameter( this, "delay", new DoubleToken( 1.0));
delay.setTypeEquals( DoubleToken.class);
input.delayTo( output);
}
public Parameter delay;
public DEIOPort input;
public DEIOPort output;
private Token _currentInput;
public void fire() throws IllegalActionException {
_currentInput = input.get( 0);
}
public boolean postfire() throws IllegalActionException {
output.send( 0, _currentInput,
(( DoubleToken) delay.getToken()).doubleValue());
return super.postfire();
}
}
}

```

Figure 4 A simulator-specific blocks in DE

```

package VisualSim.Simulators.de.lib;
import statements ...
public class Server extends DETransformer {
public DEIOPort newServiceTime;
public Parameter serviceTime;
private Token _currentInput;
private double _nextTimeFree = Double.NEGATIVE_INFINITY;
public Server( TypedCompositeActor container, String name)
throws NameDuplicationException, IllegalActionException {
super( container, name);
serviceTime = new Parameter( this, "serviceTime", new
DoubleToken( 1.0));
serviceTime.setTypeEquals( BaseType.DOUBLE);
newServiceTime = new DEIOPort( this, "newServiceTime", true,
false);
newServiceTime.setTypeEquals( BaseType.Double);
output.setTypeAtLeast( input);
input.delayTo( output);
newServiceTime.delayTo( output);
}
...attributeChanged(), clone() methods ...
public void initialize() throws IllegalActionException {
super.initialize();
_nextTimeFree = Double.NEGATIVE_INFINITY;
}
}

```

```

public boolean prefire() throws IllegalArgumentException {
    DEDirector director = (DEDirector) getDirector(); DEDirector dir
    = (DEDirector) getDirector();
    if (director.getCurrentTime() >= _nextTimeFree) {
        return true;
    } else {
        // Schedule a firing if there is a pending token so it can be
        served.
        if (input.hasToken( 0)) {
            director.fireAt( this, _ nextTimeFree);
        }
        return false;
    }
}

public void fire() throws IllegalArgumentException {
    if (newServiceTime.getWidth() > 0 && newServiceTime.hasToken( 0))
    {
        DoubleToken time = (DoubleToken)( newServiceTime.get( 0));
        serviceTime.setToken( time);
    }
    if (input.getWidth()>0 && input.hasToken( 0)) {
        _currentInput = input.get( 0);
        double delay = (( DoubleToken)
        serviceTime.getToken()).doubleValue();
        _nextTimeFree = (( DEDirector) getDirector()).getCurrentTime() +
        delay;
    } else {
        _currentInput = null;
    }
}

public boolean postfire() throws IllegalArgumentException {
    if (_ currentInput != null) {
        double delay = (( DoubleToken)
        serviceTime.getToken()).doubleValue();
        output.send( 0, _currentInput, delay);
    }
    return super.postfire();
}
}
}

```

Figure 5 Code for the Server block

that `prefire()` return false. Otherwise, the DE scheduler will keep firing the block until the inputs are all consumed, which will never happen if the block is not consuming inputs! Like the `SimpleDelay` block in Figure 4, this one produces outputs with future time stamps, using the overloaded `send()` method of `DEIOPort` that takes a delay argument. There is a subtlety associated with this design. If the model mutates during execution, and the `Server` block is deleted, it cannot retract events that it has already sent to the output. Those events will be seen by the destination block, even if by that time neither the server nor the destinations are in the topology! This could lead to some unexpected results, but hopefully, if the destination block is no longer connected to anything, then it will not do much with the token.

1.2.3 Thread Blocks

In some cases, it is useful to describe a block as a thread that waits for input tokens on its input ports. The thread suspends while waiting for input tokens and is resumed when some or all of its input ports have input tokens. While this description is functionally equivalent to the standard

description explained above, it leverages on the Java multi-threading infrastructure to save the state information.

Consider the code for the ABRecognizer block shown in Figure 6. The two code listings implement two blocks with equivalent behavior. The left one implements it as a threaded block, while the right one implements it as a standard block. We will from now on refer to the left one as the threaded description and the right one as the standard description. In both descriptions, the block has two input ports, `inportA` and `inportB`, and one output port, `outport`. The behavior is as follows. Produce an output event at `outport` as soon as events at `inportA` and `inportB` occur in that particular order, and repeat this behavior.

Note that the standard description needs a state variable `state`, unlike the case in the threaded description. In general the threaded description encodes the state information in the position of the code, while the standard description encodes it explicitly using state variables. While it is true that the

```

public class ABRecognizer extends DThreadActor {
    StringToken msg = new StringToken("Seen AB");

    // the run method is invoked when the thread
    // is started.
    public void run() {
        while (true) {
            waitForNewInputs();
            if (inportA.hasToken(0)) {
                IOPort[] nextInport = {inportB};
                waitForNewInputs(nextInport);
                outport.broadcast(msg);
            }
        }
    }
}

public class ABRecognizer extends DEActor {
    StringToken msg = new StringToken("Seen AB");

    // We need an explicit state variable in
    // this case.
    int state = 0;

    public void fire() {
        switch (state) {
            case 0:
                if (inportA.hasToken(0)) {
                    state = 1;
                    break;
                }
            case 1:
                if (inportB.hasToken(0)) {
                    state = 0;
                    outport.broadcast(msg);
                }
        }
    }
}

```

Figure 6 Code listings for two style of writing the ABRecognizer block

context switching overhead associated with multi-threading application reduces the performance, we argue that the simplicity and clarity of writing blocks in the threaded fashion is well worth the cost in some applications.

To write a block in the threaded fashion, one simply derives from the `DThreadActor` class and implements the `run()` method. In many cases, the content of the `run()` method is enclosed in the infinite '`while(true)`' loop since many useful threaded blocks do not terminate.

The `waitForNewInputs()` method is overloaded and has two flavors, one that takes no arguments and another that takes an `IOPort` array as argument. The first suspends the thread until there is at least one input token in at least one of the input ports, while the second suspends until there is at least one input token in any one of the specified input ports, ignoring all other tokens.

In the current implementation, both versions of `waitForNewInputs()` clear all input ports before the thread suspends. This guarantees that when the thread resumes, all tokens available are new, in the sense that they were not available before the `waitForNewInput()` method call. The implementation also guarantees that between calls to the `waitForNewInputs()` method, the rest of the DE model is suspended. This is equivalent to saying that the section of code between calls to the `waitForNewInput()` method is a critical section. One immediate implication is that the result of the method calls that check the configuration of the model (e.g. `hasToken()` to check the receiver) will not be invalidated during execution in the critical section. It also means that this should not be viewed as a way to get parallel execution in DE.

It is important to note that the implementation serializes the execution of threads, meaning that at any given time there is only one thread running. When a threaded block is running (i.e. executing inside its run() method), all other threaded blocks and the Simulator are suspended. It will keep running until a waitForNewInputs() statement is reached, where the flow of execution will be transferred back to the

Simulator. Note that the Simulator thread executes all non-threaded blocks. This serialization is needed because the DE simulator has a notion of global time, which makes parallelism much more difficult to achieve.

The serialization is accomplished by the use of monitor in the DETHreadActor class. Basically, the fire() method of the DETHreadActor class suspends the calling thread (i.e. the Simulator thread) until the threaded block suspends itself (by calling waitForNewInputs()). One key point of this implementation is that the threaded blocks appear just like an ordinary DE block to the DE Simulator. The DETHreadActor base class encapsulates the threaded execution and provides the regular interfaces to the DE Simulator. Therefore the threaded description can be used whenever an ordinary block can, which is everywhere.

The code in Figure 7 implements the run method of a slightly more elaborate block with the following behavior:

Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs.

The DE Simulator supports parallel execution in the form of blocks containing private threads and callbacks.

1.3 Composing DE with Other Simulators

One of the major concepts in VisualSim is modeling heterogeneous systems through the use of hierarchical heterogeneity. Blocks on the same level of hierarchy obey the same set of semantics rules. Inside some of these blocks may be another simulator with a different model of computation. This mechanism is supported through the use of opaque composite blocks. An example is shown in Figure 8.

The outermost simulator is DE and it contains seven blocks, two of them are opaque and composite. The opaque composite blocks contain subsystems, which in this case are in the DE and CT Simulators.

1.3.1 DE inside another Simulator

The DE subsystem completes one iteration whenever the opaque composite block is fired by the

```

public void run() {
    try {
        while (true) {
            // In initial state..
            waitForNewInputs();
            if (R.hasToken(0)) {
                // Resetting..
                continue;
            }
            if (A.hasToken(0)) {
                // Seen A..
                IOPort[] ports = {B,R};
                waitForNewInputs(ports);
                if (!R.hasToken(0)) {
                    // Seen A then B..
                    O.broadcast(new DoubleToken(1.0));
                    IOPort[] ports2 = {R};
                    waitForNewInputs(ports2);
                } else {
                    // Resetting
                    continue;
                }
            } else if (B.hasToken(0)) {
                // Seen B..
                IOPort[] ports = {A,R};
                waitForNewInputs(ports);
                if (!R.hasToken(0)) {
                    // Seen B then A..
                    O.broadcast(new DoubleToken(1.0));
                    IOPort[] ports2 = {R};
                    waitForNewInputs(ports2);
                } else {
                    // Resetting
                    continue;
                }
            } // while (true)
        } catch (IllegalActionException e) {
            getManager().notifyListenersOfException(e);
        }
    }
}

```

Figure 7 The run() method of the ABRO block

outer simulator. One of the complications in mixing Simulators is in the synchronization of time. Denote the current time of the DE subsystem by t_{inner} and the current time of the outer simulator by t_{outer} . An iteration of the DE subsystem is similar to an iteration of a top-level DE model, except that prior to the iteration tokens are transferred from the ports of the opaque composite blocks into the ports of the contained DE subsystem, and after the end of the iteration, the Simulator requests a refire at the smallest time stamp in the event queue of the DE subsystem. This presumes that the DE subsystem knows at what time stamp, it or one of its contained blocks, will wish to be refired. Currently the DE simulator can handle such asynchronous events only if it is not inside another simulator.

The transfer of tokens from the ports of the opaque composite block into the ports of the contained DE subsystem blocks is done in the transferInputs() method of the DE Simulator. This method is extended from its default implementation in the Director class. The implementation in the DESimulator class advances the current time of the DE subsystem to the current time of the outer simulator, then calls `super.transferInputs()`. It is done in order to correctly associate tokens seen at the input ports of the opaque composite block with events at the current time of the outer simulator, t_{outer} , and put these events into the global event queue. This mechanism is, in fact, how the DE subsystem synchronize its current time, t_{inner} , with the current time of the outer simulator, t_{outer} . (Recall that

the DE Simulator advances time by looking at the smallest time stamp in the event queue of the DE subsystem). Specifically, before the advancement of the current time of the DE subsystem t_{inner} is less than or equal to the t_{outer} , and after the advancement t_{inner} is equal to the t_{outer} . Requesting a refiring is done in the postfire() method of the (inner) DE Simulator by calling the fireAt() method of the executive (outer) Simulator. Its purpose is to ensure that events in the DE sub-

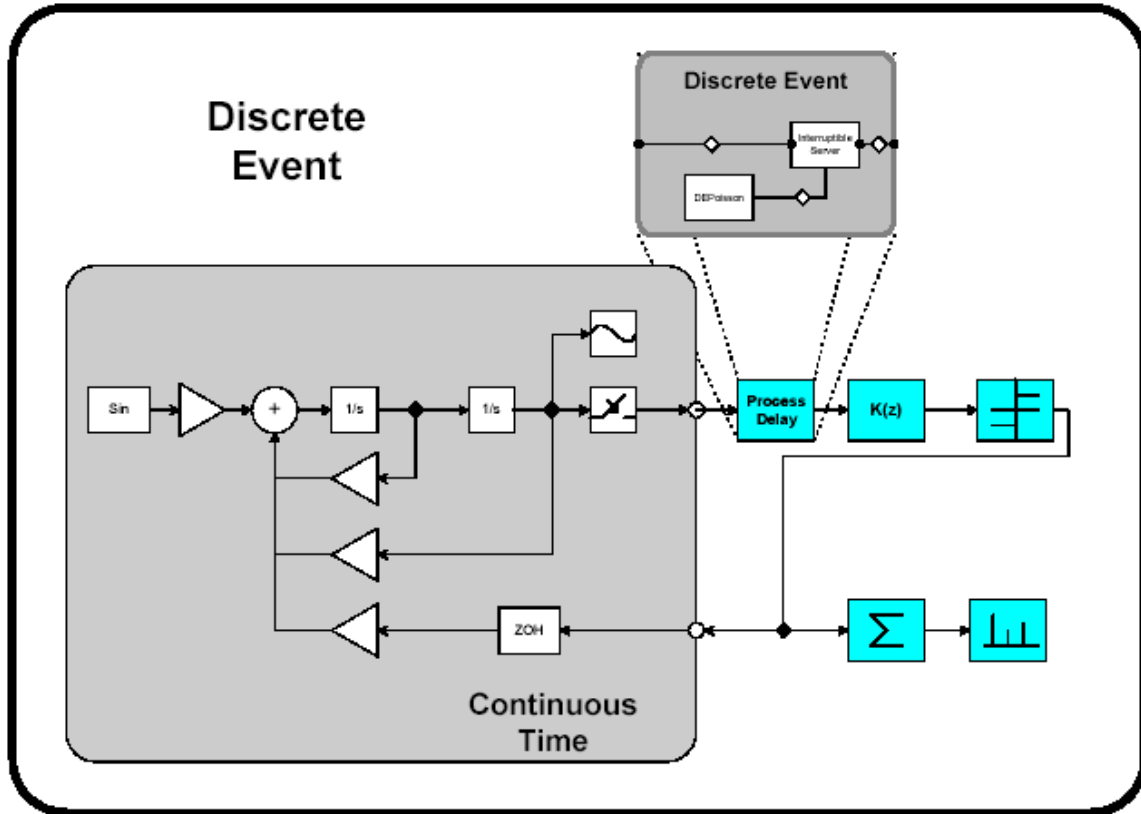


Figure 8 An example of heterogeneous and hierarchical composition. The CT subsystem and DE subsystem are inside an outermost DE system.

system are processed on time with respect to the current time of the outer simulator, t_{outer} . Note that if the DE subsystem is fired due to the outer simulator processing a refire request, then there may not be any tokens in the input port of the opaque composite block at the beginning of the DE subsystem iteration. In that case, no new events with time stamps equal to t_{outer} will be put into the global event queue. Interestingly, in this case, the time synchronization will still work because t_{inner} will be advanced to the smallest time stamp in the global event queue which, in turn, has to equal t_{outer} because we always request a refire according to that time stamp.

1.3.2 Another Simulator inside DE

Due to its nature, any opaque composite block inside DE is opaque and therefore, as far as the DE Simulator is concerned, behaves exactly like a simulator polymorphic block. Recall that simulator polymorphic blocks are treated as functions with zero delay in computation time. To produce events in the future, simulator polymorphic blocks request a refire from the DE Simulator and then produce the events when it is refired.

2 CT Simulator

2.1 Introduction

The continuous-time (CT) simulator in VisualSim aims to help the design and simulation of systems that can be modeled using ordinary differential equations (ODEs). ODEs are often used to model analog circuits, plant dynamics in control systems, lumped-parameter mechanical systems, lumped-parameter heat flows and many other physical systems.

Let's start with an example. Consider a second order differential system,

$$\begin{aligned} m\ddot{z}(t) + b\dot{z}(t) + kz(t) &= u(t) \quad (1) \\ y(t) &= c \cdot z(t) \\ z(0) &= 10, \dot{z}(0) = 0. \end{aligned}$$

The equations could be a model for an analog circuit as shown in Figure 9 (a), where z is the voltage

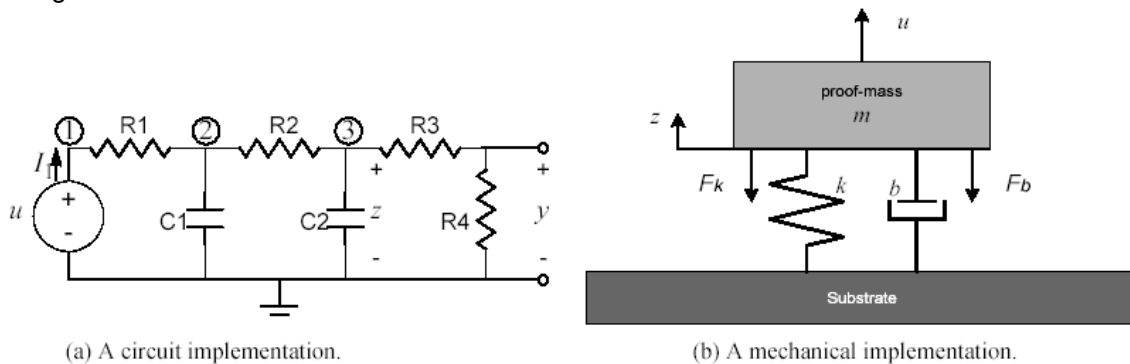


Figure 9 Possible implementations of the system equations.

of node 3, and

$$m = R1 \cdot R2 \cdot C1 \cdot C2 \quad (2)$$

$$k = R1 \cdot C1 + R2 \cdot C2$$

$$b = 1$$

$$c = \frac{R4}{R3 + R4}$$

Or it could be a lumped-parameter spring-mass mechanical model for the system shown in Figure 9(b), where z is the position of the mass, m is the mass, k is the spring constant, b is the damping parameter, and $c = 1$.

In general, an ODE-based continuous-time system has the following form:

$$\dot{x} = f(x, u, t) \quad (3)$$

$$y = g(x, u, t) \quad (4)$$

$$x(t_0) = x_0, \quad (5)$$

where, $t \in \mathcal{R}, t \geq t_0$, a real number, is continuous time. At any time t , $x \in \mathcal{R}^n$, an n -tuple of real numbers, is the state of the system; $u \in \mathcal{R}^m$ is the m -dimensional input of the system; $y \in \mathcal{R}^l$ is the l -dimensional output of the system; $\dot{x} \in \mathcal{R}^n$ is the derivative of with respect to time, i.e.

$$\dot{x} = \frac{dx}{dt}. \quad (6)$$

Equations (3), (4) and (5) are called the system dynamics, the output map, and the initial condition of the system, respectively. For example, in the mechanical system above, if we define a vector

$$x(t) = \begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix}, \quad (7)$$

then system (1) can be written in form of (3)-(5), like

$$\dot{x}(t) = \frac{1}{m} \begin{bmatrix} 0 & 1 \\ -k & -b \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u(t) \quad (8)$$

$$y(t) = \begin{bmatrix} c & 0 \end{bmatrix} x(t)$$

$$x(0) = \begin{bmatrix} 10 \\ 0 \end{bmatrix}.$$

The solution, $x(t)$, of the set of the ODE (3)-(5) is a continuous function of time, also called a wave-form, which satisfies the equation (3) and initial condition of (5). The output of the system is then defined as a function of $x(t)$ and $u(t)$, which satisfies (4). The precise solution of a set of ODEs is usually impossible to be found using digital computers. Numerical solutions are approximations of the precise solution. A numerical solution of ODEs is usually done by integrating the right-hand side of (3) on a discrete set of time points. Using digital computers to simulate continuous-time systems has been studied for more than three decades. One of the most well-known tools is Spice. The CT simulator differs from Spice-like continuous-time simulators in two ways — the system specification is somewhat different, and it is designed to interact with other models of computation.

2.1.1 System Specification

There are usually two ways to specify a continuous-time system, the conservation-law model and the signal-flow model. The conservation-law models, like the nodal analysis in circuit simulation and bond graphs in mechanical models, define systems by their physical components, which specify relations of cross and through variables, and conservation laws are used to compile the component relations into global system equations. For example, in circuit simulation, the cross variables are voltages, the through variables are currents, and the conservation laws are Kirchhoff's laws. This model directly reflects the physical components of a system, thus is easy to construct from a potential implementation. The actual mathematical representation of the system is hidden. In signal-flow models, entities in a system are maps that define the mathematical relation between their input and output signals. Entities communicate by passing signals. This

kind of models directly reflects the mathematical relations among signals, and is more convenient for specifying systems that do not have an explicit physical implementation yet.

In the CT simulator of VisualSim, the signal-flow model is chosen as the interaction semantics. The conservation-law semantics may be used within an entity to define its I/O relation. There are four major reasons for this decision:

1. The signal-flow model is more abstract. VisualSim focuses on system-level design and behavior simulation. It is usually the case that, at this stage of a design, users are working with abstract mathematical models of a system, and the implementation details are unknown or not cared about.
2. The signal flow model is more flexible and extensible, in the sense that it is easy to embed components that are designed using other models. For example, a discrete controller can be modeled as a component that internally follows a discrete event model of computation but exposes a continuous-time interface.
3. The signal flow model is consistent with other models of computation in VisualSim. Most models of computation in VisualSim use message-passing as the interaction semantics. Choosing the signal-flow model for CT makes it consistent with other simulators, so the interaction of heterogeneous systems is easy to study and implement. This also allows simulator polymorphic blocks to be used in the CT simulator.
4. The signal flow model is compatible with the conservation law model. For physical systems that are based on conservation laws, it is usually possible to wrap them into an entity in the signal flow model. The inputs of the entity are the excitations, like the current on ideal current sources, and the outputs are the variables that the rest of the system may be interested in.

The signal flow block diagram of the system (3)-(5) is shown in Figure 10. The system dynamics (3) is built using integrators with feedback. In this Figure, u , \dot{x} , x , and y , are continuous signals flowing from one block to the next. Notice that this diagram is only conceptual, most models may involve multiple integrators¹. Time is shared by all components, so it is not considered as an input. At any fixed time t , if the "snapshot" values $x(t)$ and $u(t)$ are given, then $y(t)$ can be found by evaluating f and g , which can be achieved by firing the respective blocks. The "snapshot" of all the signals at t called the behavior of the system at time.

The signal-flow model for the example system (1) is shown in Figure 11. For comparison purpose, the conservation-law model (modified nodal analysis) of the system shown in Figure 9(a) is shown in (9).

$$\begin{bmatrix} \frac{1}{R1} & -\frac{1}{R1} & 0 & 0 & -1 \\ -\frac{1}{R1} & \frac{1}{R1} + \frac{1}{R2} + C1 \frac{d}{dt} & -\frac{1}{R2} & 0 & 0 \\ 0 & -\frac{1}{R2} & \frac{1}{R2} + \frac{1}{R3} + C2 \frac{d}{dt} & -\frac{1}{R3} & 0 \\ 0 & 0 & -\frac{1}{R3} & \frac{1}{R3} + \frac{1}{R4} & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ y \\ I_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ u \end{bmatrix} \quad (9)$$

By doing some math, we can see that (9) and (8) are in fact equivalent. Equation (9) can be easily assembled from the circuit, but it is more complicated than (8). Notice that in (9) is the derivative operator, which is replaced by an integration algorithm at each time step, and the system equations reduce to a set of algebraic equations. Spice software is known to have a very good simulation engine for models in form of (9).

¹ VisualSim does not support vectorization in the CT simulator yet.

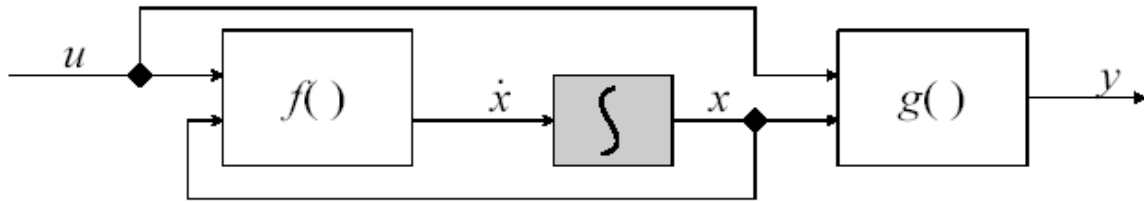


Figure 10 A conceptual block diagram for continuous time systems.

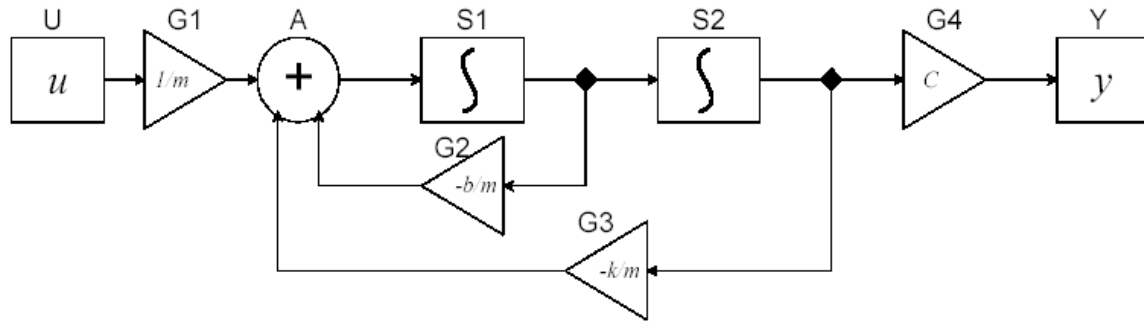


Figure 11 The block diagram for the example system.

2.1.2 Time

One distinct characterization of the CT model is the continuity of time. This implies that a continuous-time system have a behavior at any time instance. The simulation engine of the CT model should be able to compute the behavior of the system at any time point, although it may march discretely in time. In order to achieve an accurate simulation, time should be carefully discretized. The discretization of time, which appears as integration step sizes, may be determined by time points of interest (e. g. discontinuities), by the numerical error of integration, and by the convergence in solving algebraic equations. Time is also global, which means that all components in the system share the same notion of time.

2.2 Solving ODEs numerically

We outline some basic terminologies on numerical ODE solving techniques that are used in this chapter. This is not a summary of numerical ODE solving theory. For a detailed treatment for ODEs and their numerical solutions, please refer to books on numerical solutions for ODEs. Not all ODEs have a solution, and some ODEs have more than one solution. In such situations, we say that the solution is not well defined. This is usually a result of errors in the system modeling. We restrict our discussion to systems that have unique solutions. Theorem 1 in Appendix A states the conditions for the existence and uniqueness of solutions of ODEs. Roughly speaking, we denote by D a set in \mathcal{R} which contains at most a finite number of points per unit interval, and let u be piecewise-continuous on $\mathcal{R} - D$. Then, for any fixed $u(t)$, if f is also piecewise-continuous on $\mathcal{R} - D$, and f satisfies the Lipschitz condition, then the ODE (3) with the initial condition (5) has a unique solution. The solution is called the state trajectory of the system. The key to simulating a continuous-time system numerically is to find an accurate numerical approximation of the state trajectory.

2.2.1 Basic Notations

Usually, only the solution on a finite time interval $[t_0, t_f]$ is needed. A simulation of the system is performed on discrete time points in this interval. We denote by

$$Tc = \{t_0, t_1, t_2, \dots, t_n, \dots, t_f\}, Tc \subset [t_0, t_f], \quad (10)$$

where

$$t_0 < t_1 < t_2 < \dots < t_n < \dots < t_f, \quad (11)$$

the set of the discrete time points of interest. To explicitly illustrate the discretization of time and the difference between the precise solution and the numerical solution, we use the following notation in the rest of the chapter:

- t_n : the n -th time point, to explicitly show the discretization of time. However, t is specified, if the index n is not important.
- $X[t_i, t_j]$: the precise (continuous) state trajectory from time t_i to t_j ;
- $X(t_n)$ the precise solution of (3) at time t_n ;
- x_{t_n} : the numerical solution of (3) at time t_n ;
- $h_n = t_n - t_{n-1}$: step size of numerical integration. We also write h if the index n in the sequence is not important. For accuracy reason, h may not be uniform.
- $\|X(t_n) - x_{t_n}\|$: the 2-normed difference between the precise solution and the numerical solution at step n is called the (*global*) error at step n ; the difference, when we assume $x_{t_0} \dots x_{t_{n-1}}$ are precise, is called the local error at step n . Local errors are usually easy to estimate and the estimation can be used for controlling the accuracy of numerical solutions.

A general way of numerically simulating a continuous-time system is to compute the state and the output of the system in an increasing order of t_n . Such algorithms are called the *time-marching algorithms*, and, we only consider these algorithms. There are variety of time marching algorithms

that differ on how x_{t_n} is computed given $x_{t_0} \dots x_{t_{n-1}}$. The choice of algorithms is application dependent, and usually reflects the speed, accuracy, and numerical stability trade-offs.

2.2.2 Fixed-Point Behavior

Numerical ODE solving algorithms approximate the derivative operator in (3) using the history and the current knowledge on the state trajectory. That is, at time t_n , the derivative of x is

approximated by a function of $x_{t_0}, \dots, x_{t_{n-1}}, x_{t_n}$, i.e.

$$\dot{x}_{t_n} = p(x_{t_0}, \dots, x_{t_{n-1}}, x_{t_n}). \quad (12)$$

Plugging (3) in this, we get

$$p(x_{t_0}, \dots, x_{t_{n-1}}, x_{t_n}) = f(x_{t_n}, u(t_n), t_n) \quad (13)$$

Depending on whether x_{t_n} explicitly appears in (13), the algorithms are called explicit integration algorithms or implicit integration algorithms. That is, we end up solving a set of algebraic equations in one of the two forms:

$$x_{t_n} = F_E(x_{t_0}, \dots, x_{t_{n-1}}) \quad (14)$$

or,

$$F_I(x_{t_0}, \dots, x_{t_n}) = 0, \quad (15)$$

where F_E and F_I are derived from the time t_n , the input $u(t_n)$, the function f , and the history of x and \dot{x} . Solving (14) or (15) at a particular time is called an iteration of the CT simulation at t_n .

Equation (14) can be solved simply by a function evaluation and an assignment. But the solution of (15) is the fixed point of F_I , which may not exist, may not be unique, or may not be able to be found. The contraction mapping theorem [13] shows the existence and uniqueness of the fixed-point solution, and provides one way to find it. Given the map F_I that is a local contraction map (generally true for small enough step sizes) and let an initial guess σ_0 be in the contraction radius, then a unique fixed point exists and can be found by iteratively computing:

$$\sigma_1 = F_E(\sigma_0), \sigma_2 = F_E(\sigma_1), \sigma_3 = F_E(\sigma_2), \dots \quad (16)$$

Solving both (14) and (15) should be thought of as finding the fixed-point behavior of the system at a particular time. This means both functions F_E and F_I should be smooth w. r. t. time, during a single iteration of the simulation. This further implies that the topology of the system, all the parameters, and all the internal states that the firing functions depend on should be kept unchanged. We require that simulator polymorphic blocks to update internal states only in the `postfire()` method exactly for this reason.

2.2.3 ODE Solvers Implemented

The following solvers have been implemented in the CT simulator.

1. Forward Euler solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_n} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \end{aligned} \quad (17)$$

2. Backward Euler solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_{n+1}} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \end{aligned} \quad (18)$$

3. 2(3)-order Explicit Runge-Kutta solver

$$K_0 = h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \quad (19)$$

$$K_1 = h_{n+1} \cdot f(x_{t_n} + K_0/2, u_{t_n + h_{n+1}/2}, t_n + h_{n+1}/2)$$

$$K_2 = h_{n+1} \cdot f(x_{t_n} + 3K_1/4, u_{t_n + 3h_{n+1}/4}, t_n + 3h_{n+1}/4)$$

$$\tilde{x}_{t_{n+1}} = x_{t_n} + \frac{2}{9}K_0 + \frac{1}{3}K_1 + \frac{4}{9}K_2$$

with error control:

$$K_3 = h_{n+1} \cdot f(\tilde{x}_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \quad (20)$$

$$LTE = -\frac{5}{72}K_0 + \frac{1}{12}K_1 + \frac{1}{9}K_2 - \frac{1}{8}K_3$$

if $|LTE| < ErrorTolerance$, $x_{t_{n+1}} = \tilde{x}_{t_{n+1}}$, otherwise, fail. If this step is successful, the next integration step size is predicted by:

$$h_{n+2} = h_{n+1} \cdot \max(0.5, 0.8 \cdot \sqrt[3]{(ErrorTolerance)/|LTE|}) \quad (21)$$

4. Trapezoidal Rule solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + \dot{x}_{t_{n+1}}) \\ &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1})) \end{aligned} \quad (22)$$

Among these solvers, 1) and 3) are explicit; 2) and 4) are implicit. Also, 1) and 2) do not perform

step size control, so are called fixed-step-size solvers; 3) and 4) change step sizes according to error estimation, so are called variable-step-size solvers. Variable-step-size solvers adapt the step sizes according to changes of the system flow, thus are "smarter" than fixed-step-size solvers.

The existence and uniqueness of the solution of an ODE (Theorem 1 in Appendix A) allows the right-hand side of (3) to be discontinuous at a countable number of discrete points D , which are called the breakpoints (also called the discontinuous points in some literature). These breakpoints may be caused by the discontinuity of input signal u , or by the intrinsic flow of f . In theory, the solutions at these points are not well defined. But the left and right limits are. So, instead of solving the ODE at those points, we would actually try to find the left and right limits.

One impact of breakpoints on ODE solvers is that history solutions are useless when approximating the derivative of x after the breakpoints. The solver should resolve the new initial conditions and start the solving process as if it is at a starting point. So, the discretization of time should step exactly on breakpoints for the left limit, and start at the breakpoint again after finding the right limit. A breakpoint may be known beforehand, in which case it is called a *predictable breakpoint*. For example, a square wave source block knows its next flip time. This information can be used to control the discretization of time. A breakpoint can also be unpredictable, which means it is unknown until the time it occurs. For example, a block that varies its functionality when the input signal crosses a threshold can only report a "missed" breakpoint after an integration step is finished. How to handle break-points correctly is a big challenge for integrating continuous-time models with discrete models like DE and FSM.

2.2.4 Breakpoint ODE Solvers

Breakpoints in the CT simulator are handled by adjusting integration steps. We use a table to handle predictable breakpoints, and use the step size control mechanism to handle unpredictable breakpoints. Since the history information is useless at breakpoints, special ODE solvers are designed to restart the numerical integration process. In particular, we have implemented the following breakpoint ODE solvers.

1. DerivativeResolver: It calculates the derivative of the current state, i.e. $\frac{dx}{dt}$. This is simply done by evaluation the right-hand side of (3). At breakpoints, this solver is used for the first step to generate history information for explicit methods or one step methods.
2. ImpulseBESolver:

$$\tilde{x}_{t_{n+1}} = x_{t_n} + h_{n+1} \cdot \dot{x}_{t_{n+1}} \quad (23)$$

$$x_{t_n}^+ = \tilde{x}_{t_{n+1}} - h_{n+1} \cdot \dot{x}_{t_n}^+$$

The two time points and have the same time value. This solver is used for breakpoints at which a Dirac impulse signal appears.

Notice that none of these solvers advance time. They can only be used at breakpoints.

2.3 Signal Types

The CT simulator of VisualSim supports continuous time mixed-signal modeling. As a consequence, there could be two types of signals in a CT model: continuous signals and discrete events. Note that for both types of signals, time is continuous. These two types of signals directly affect the behavior of a receiver that contains them. A continuous CTReceiver contains a sample of a continuous signal at the current time. Reading a token from that receiver will not consume the token. A discrete CTReceiver may or may not contain a discrete event. Reading from a

discrete CTReceiver with an event will consume the event, so that events are processed exactly once². Reading from an empty discrete CTReceiver is not allowed.

Note that some blocks can be used to compute on both continuous and discrete signals. For example, an adder can add two continuous signals, as well as two sets of discrete events. Whether a particular link among blocks is continuous or discrete is resolved by a signal type system. The signal type system understands signal types on specific blocks (indicated by the interfaces they implement or the parameters specified on their ports), and try to resolve signal types on the ports of simulator polymorphic blocks.

The signal type system in the CT simulator works on a simple lattice of signal types, shown in Figure 12. A type lower in the lattice is more specific than a type higher in the lattice. A CT model is well-defined and executable, if and only if all ports are resolved to either CONTINUOUS or DISCRETE. Some blocks have their signal types fixed. For example, an *Integrator* has a CONTINUOUS input and a CONTINUOUS output; a *PeriodicSampler* has a CONTINUOUS input and a DISCRETE output; a *TriggeredSampler* has one CONTINUOUS input (the input), one DISCRETE

input (the trigger), and a DISCRETE output; and a *ZeroOrderHold* has a DISCRETE input and a CONTINUOUS output. For simulator polymorphic blocks that implement the *SequenceActor* interface, i.e. they operate solely on sequences of tokens, their inputs and outputs are treated as DISCRETE. For other simulator polymorphic blocks that can operate on both continuous and discrete signals, the signal type on their ports are initially UNRESOLVED. The signal type system will resolve and check signal types of ports according to the following two rules:

- ◆ If a port p is connected to another port q with a more specific type, then the type of p is resolved to that of the port q. If p is CONTINUOUS but q is DISCRETE, then both of them are resolved to NOT-A-TYPE.

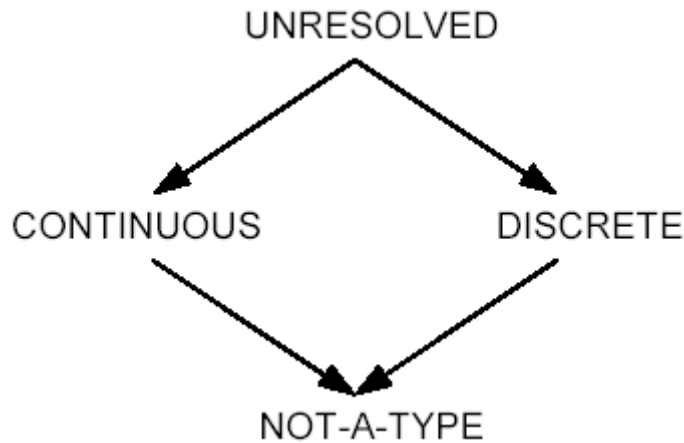


Figure 12 A signal type lattices

² This distinction of receivers is also called state and event semantics in some literatures

- ◆ Unless otherwise specified, the types of the input ports and output ports of a block are the same.

At the end of the signal-type resolution, if any port is of type UNRESOLVED or NOT-A-TYPE, then the topology of the system is illegal, and the execution is denied. The signal type of a port can also be forced by adding an parameter "*signalType*" to the port.

The signal type system will recognize this parameter and resolve other types accordingly. To add this parameter, right click on the port, select Configure, add a parameter with the name *signalType* and the value of a string of either "CONTINUOUS" or "DISCRETE", noting the quotation marks.

Signal types may be trickier at the boundaries of composite blocks than within a CT model. Because of the information hiding, it may not be obvious which port of another level of hierarchy is continuous and which port is discrete. In the CT simulator, we follow these rules to resolve signal types for composite ports:

- ◆ A *TypedCompositeActor* within a CT model is always treated as entirely discrete. Within a CT model, for any opaque composite block that may contain continuous dynamics at a deeper level, use the *CTCompositeActor* (listed in the block library as "continuous time composite block" in Control list) or the modal model composite block.
- ◆ For a *CTCompositeActor* or a modal model within a CT model, all its ports are treated as continuous by default. To allow a discrete event going through the composite block boundary, manually set the signal type of that port by adding the *signalType* parameter.
- ◆ For a *TypedCompositeActor* containing a CT model, all the ports of the *TypedCompositeActor* are treated as discrete, and the CT Simulator to use is the *CTMixedSignalSimulator* (listed as CTSimulator in the simulators library).
- ◆ For a *CTCompositeActor* or a modal model containing a CT model, all the signal types of the ports of the container are treated as continuous, and can be set by adding the *signalType* parameter. The *CTSimulator* to use in this situation is the *CTEmbeddedSimulator*.

2.4 CT Blocks

A CT system can be built up using blocks in the VisualSim Control library list and simulator polymorphic blocks that have continuous behaviors (i.e. all blocks that do not implement the *SequenceActor* interface). The key block in CT is the integrator. It serves the unique role of wiring up ODEs. Other blocks in a CT system are usually stateless. A general understanding is that, in a pure continuous-time model, all the information — the state of the system— is stored in the integrators.

2.4.1 CT Block Interfaces

In order to schedule the execution of blocks in a CT model and to support the interaction between CT and other Simulators (which are usually discrete), we provide the following interfaces:

- ◆ *CTDynamicActor*. Dynamic blocks are blocks that contains continuous dynamics in their I/ O path. An integrator is a dynamic block, and so are all blocks that have integration relations from their inputs to their outputs.
- ◆ *CTEventGenerator*. Event generators are blocks that convert continuous time input signals to discrete output signals.
- ◆ *CTStatefulActor*. Stateful blocks are blocks that have internal states. The reason to classify this kind of block is to support rollback, which may happen when a CT model is embedded in a discrete event model.
- ◆ *CTStepSizeControlActor*. Step size control blocks influence the integration step size by telling the Simulator whether the current step is accurate. The accuracy is in the sense of both tolerable numerical errors and absence of unpredictable breakpoints. It may also provide

information about refining a step size for an inaccurate step and suggesting the next step size for an accurate step.

- ◆ CTWaveformGenerator. Waveform generators are blocks that convert discrete input signals to continuous-time output signals.

Strictly speaking, event generators and waveform generators do not belong to any simulator, but the CT simulator is design to handle them intrinsically. When building systems, CT parts can always provide discrete interface to other Simulators.

Neither a loop of dynamic blocks nor a loop of non-dynamic blocks is allowed in a CT model.

They introduce problems about the order that blocks be executed. A loop of dynamic blocks can be easily broken by a Scale block with scale 1. A loop of non-dynamic blocks builds an algebraic equation.

2.5 CT Simulators

There are three CT Simulators CTMultiSolverDirector, CTMixedSignalDirector, and CTEEmbeddedDirector. The first one can only serve as a top-level Simulator, a CTMixedSignalDirector can be used both at the top-level and inside a composite block, and a CTEEmbeddedDirector can only be contained in a CTCompositeActor. In terms of mixing models of computation, all the Simulators can execute composite blocks that implement other models of computation, as long as the composite blocks are properly connected. Only CTMixedSignalDirector and CTEEmbeddedDirector can be contained by other Simulators. The outside simulator of a composite block with CTMixedSignalDirector can be the Discrete-Event simulator. The outside simulator of a composite block with CTEEmbeddedDirector must also be CT or FSM, if the outside simulator of the FSM model is CT.

2.5.1 ODE Solvers

There are six ODE solvers implemented in the Continuous package. Some of them are specific for handling breakpoints. These solvers are ForwardEulerSolver, BackwardEulerSolver, ExplicitRK23Solver, TrapezoidalRuleSolver, DerivativeResolver, and ImpulseBESolver. They implement the ODE solving algorithms in the ODE Solver section.

2.5.2 CT Simulator Parameters

The CTSimulator base class maintains a set of parameters which controls the execution. These parameters, shared by all CT Simulators, are listed in Table 1. Individual Simulators may have their own (additional) parameters, which will be discussed in the appropriate sections.

Name	Description	Type	Default Value
errorTolerance	The upper bound of local errors. Actors that perform integration error control (usually integrators in variable step size ODE solving methods) will compare the estimated local error to this value. If the local error estimation is greater than this value, then the integration step is considered inaccurate, and should be restarted with a smaller step sizes.	double	1e-4
initStepSize	This is the step size that users specify as the desired step size. For fixed step size solvers, this step size will be used in all non-breakpoint steps. For variable step size solvers, this is only a suggestion.	double	0.1
maxIterations	This is used to avoid the infinite loops in (implicit) fixed-point iterations. If the number of fixed-point iterations exceeds this value, but the fixed point is still not found, then the fixed-point procedure is considered failed. The step size will be reduced by half and the integration step will be restarted.	int	20
maxStepSize	The maximum step size used in a simulation. This is the upper bound for adjusting step sizes in variable step-size methods. This value can be used to avoid sparse time points when the system dynamic is simple.	double	1.0
minStepSize	The minimum step size used in a simulation. This is the lower bound for adjusting step sizes. If this step size is used and the errors are still not tolerable, the simulation aborts. This step size is also used for the first step after breakpoints.	double	1e-5
startTime	The start time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director follows the time of its executive director.	double	0.0
stopTime	The stop time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director follows the time of its executive director.	double	Double. MAX_ VALUE
synchronizeTo-RealTime	Indicate whether the execution of the model is synchronized to real time at best effort.	boolean	false
timeResolution	This controls the comparison of time. Since time in the CT domain is a double precision real number, it is sometimes impossible to reach or step at a specific time point. If two time points are within this resolution, then they are considered identical.	double	1e-10
valueResolution	This is used in (implicit) fixed-point iterations. If in two successive iterations the difference of the states is within this resolution, then the integration step is called converged, and the fixed point is considered reached.	double	1e-6

Table 1 CTSimulator Parameters

2.5.3 CTMultiSolverDirector

A *CTMultiSolverDirector* has two ODE solvers — one for ordinary use and one specifically for breakpoints. Thus, besides the parameters in the *CTSimulator* base class; this class adds two more parameters as shown in Table 2.

Name	Description	Type	Default Value
ODESolver	The fully qualified class name for the ODE solver class.	string	"ptolemy.domains.ct.kernel.solver.ForwardEulerSolver"
breakpointODESolver	The fully qualified class name for the breakpoint ODE solver class.	string	"ptolemy.domains.ct.kernel.solver.DerivativeResolver"

Table 2 Additional Parameters for CTMultiSolverDirector

A *CTMultiSolverDirector* can direct a model that has composite blocks implementing other models of computation. Simulation iteration is done in two phases: the continuous phase and the discrete phase. Let the current iteration be n . In the continuous phase, the differential equations are integrated from time t_{n-1} to t_n . After that, in the discrete phase, all (discrete) events which

happen at are processed.

The step size control mechanism will assure that no events will happen between t_{n-1} and t_n .

2.5.4 CTMixedSignalDirector

This simulator is designed to contain CT in an event-based system, like DE. When a CT subsystem is contained in the DE simulator, the CT subsystem should run ahead of the global time, and be ready for rollback. This Simulator implements this optimistic execution.

Since the outside simulator is event-based, each time the embedded CT subsystem is fired, the input data are events. In order to convert the events to continuous signals, breakpoints have to be introduced. So this Simulator extends *CTMultiSolverDirector*, which always has two ODE solvers. There is one more parameter used by this Simulator — the *runAheadLength*, as shown in Table 3.

Name	Description	Type	Default Value
runAheadLength	The maximum length of time for the CT subsystem to run ahead of the global time.	double	1.0

Table 3 Additional Parameter for CTMixedSignalDirector

When the CT subsystem is fired, the *CTMixedSignalDirector* will get the current time and the next iteration time from the outer simulator, and take the $\min(\tau - \tau', l)$ as the fire end time, where l is the value of the parameter *maxRunAheadLength*. The execution lasts as long as the fire end time is not reached or an output event is not detected.

This Simulator supports rollback; that is when the state of the continuous subsystem is confirmed (by knowing that no events with a time earlier than the CT current time will be present), the state of the system is marked. If an optimistic execution is known to be wrong, the state of the CT subsystem will roll back to the latest marked state.

2.5.5 CTEmbeddedSimulator

This Simulator is used when a CT subsystem is embedded in another continuous time system, either directly or through a hierarchy of finite state machines, like in the hybrid system scenario. This Simulator can pass step size control information up to its executive Simulator. To achieve this, the Simulator must be contained in a *CTCompositeActor*, which implements the *CTStepSizeControlActor* interface and can pass the step size control information from the inner simulator to the outer simulator.

This Simulator extends *CTMultiSolverDirector*, with no additional parameters. A major difference between this Simulator and the *CTMixedSignalDirector* is that this Simulator does not support rollback. In fact, when a CT subsystem is embedded in a continuous-time environment, rollback is not necessary.

2.6 Interacting with Other Simulators

The CT simulator can interact with other Simulators in VisualSim. In particular, we consider interaction among the CT simulator, the discrete event (DE) simulator and the finite state machine (FSM) simulator. Following circuit design communities, we call a composition of CT and DE a *mixed-signal model*; following control and computation communities, we call a composition of CT and FSM a *hybrid system model*.

There are two ways to put CT and DE models together, depending on the containment relation. In either case, event generators and waveform generators are used to convert the two types of signals. Figure 14 shows a DE component wrapped by an event generator and a waveform generator. From the input/ output point of view, it is a continuous time component. Figure 15 shows a CT subsystem wrapped by a waveform generator and an event generator. From the input/ output point of view, it is a discrete event component. Notice that event generators and

waveform generators always stay in the CT simulator.

A hierarchical composition of FSM and CT is shown in Figure 13. A CT component, by adopting the event generation technique, can have both continuous and discrete signals as its output. The FSM can use predicates on these signals, as well as its own

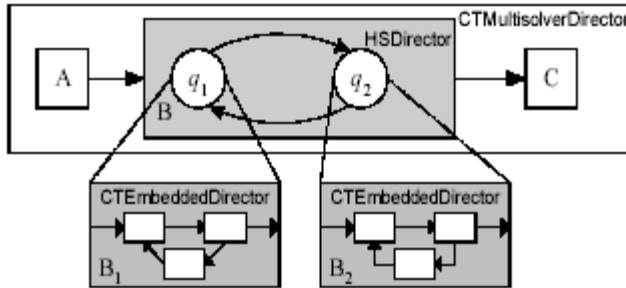


Figure 13 Hybrid system modeling

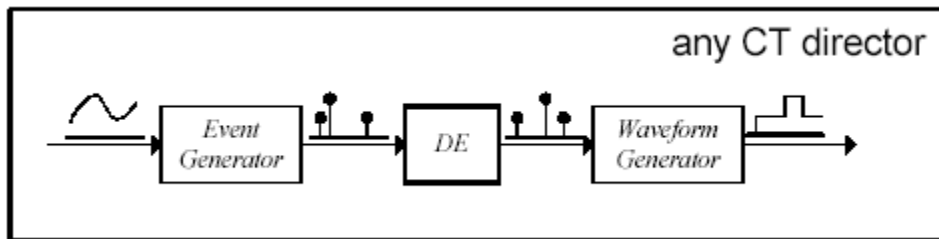


Figure 14 Embedding a DE component in a CT system

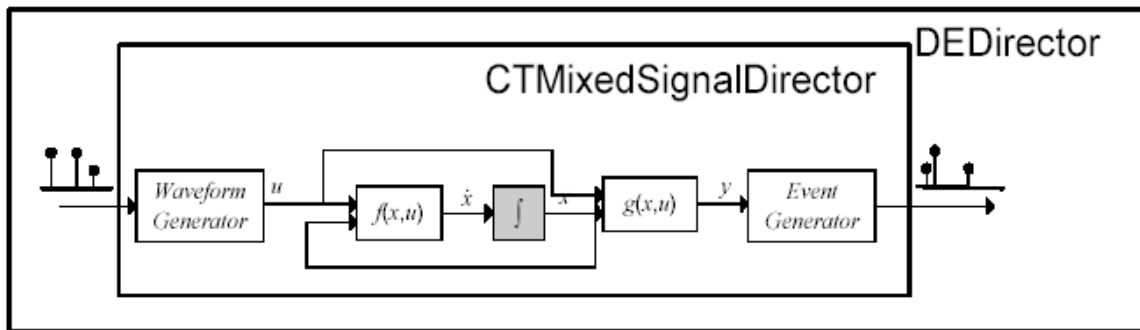


Figure 15 Embedding a CT component in a DE system

input signals, to build trigger conditions. The actions associated with transitions are usually setting parameters in the destination state, including the initial conditions of integrators.

2.7 Mixed-Signal Execution

DE inside CT.

Since time advances monotonically in CT and events are generated chronologically, the DE component receives input events monotonically in time. In addition, a composition of causal DE components is causal, so the time stamps of the output events from a DE component are always greater than or equal to the global time. From the view point of the CT system, the events produced by a DE component are predictable breakpoints.

Note that in the CT model, finding the numerical solution of the ODE at a particular time is

semantically an instantaneous behavior. During this process, the behavior of all components, including those implemented in a DE model, should keep unchanged. This implies that the DE components should not be executed during one integration step of CT, but only between two successive CT integration steps.

CT inside DE.

When a CT component is contained in a DE system, the CT component is required to be causal, like all other components in the DE system. Let the CT component have local time t , when it receives an input event with time stamp τ . Since time is continuous in the CT model, it will execute from its local time t , and may generate events at any time greater or equal to t . Thus we need

$$t \geq \tau \tag{29}$$

to ensure causality. This means that the local time of the CT component should always be greater than or equal to the global time whenever it is executed.

This ahead-of-time execution implies that the CT component should be able to remember its past states and be ready to rollback if the input event time is smaller than its current local time. The state it needs to remember is the state of the component after it has processed an input event. Consequently, the CT component should not emit detected events to the outside DE system before the global time reaches the event time. Instead, it should send a pure event to the DE system at the event time, and wait until it is safe to emit it.

2.7.1 Hybrid System Execution

Although FSM is an untimed model, its composition with a timed model requires it to transfer the notion of time from its external model to its internal model. During continuous evolution, the system is simulated as a CT system where the FSM is replaced by the continuous component refining the current FSM state. After each time point of CT simulation, the triggers on the transitions starting from the current FSM state are evaluated. If a trigger is enabled, the FSM makes the corresponding transition. The continuous dynamics of the destination state is initialized by the actions on the transition. The simulation continues with the transition time treated as a breakpoint.

2.8 Appendix F: Brief Mathematical Background

Theorem 1 [Existence and uniqueness of the solution of an ODE] Consider the initial value ODE problem

$$\begin{aligned}\dot{x} &= f(x, t) \\ x(t_0) &= x_0\end{aligned}\quad (30)$$

If f satisfies the conditions:

- ◆ [Continuity Condition] Let D be the set of possible discontinuity points; it may be empty. For each fixed $x \in \mathfrak{R}^n$ and $u \in \mathfrak{R}^m$, the function in (30) is continuous. And $\forall \tau \in D$, the left-hand and right-hand limit $f(x, u, \tau^-)$ and $f(x, u, \tau^+)$ are finite.
- ◆ [Lipschitz Condition] There is a piecewise continuous bounded function $k: \mathfrak{R} \rightarrow \mathfrak{R}^+$, where \mathfrak{R}^+ is the set of non-negative real numbers, such that $\forall t \in \mathfrak{R}, \forall \zeta, \xi \in \mathfrak{R}^n, \forall u \in \mathfrak{R}^m$

$$\|f(\xi, u, t) - f(\zeta, u, t)\| \leq k(t) \|\xi - \zeta\| \quad (31)$$

Then, for each initial condition $(t_0, x_0) \in \mathfrak{R} \times \mathfrak{R}^n$ there exists a unique continuous function $\psi: \mathfrak{R} \rightarrow \mathfrak{R}^n$ such that,

$$\psi(t_0) = x_0 \quad (32)$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t) \quad \forall t \in \mathfrak{R} \setminus D. \quad (33)$$

This function is called the solution through (t_0, x_0) of the ODE (30).

Theorem 2. [Contraction Mapping Theorem.] If $F: \mathfrak{R}^n \rightarrow \mathfrak{R}^n$ is a local contraction map at x with contraction radius ε , then there exists a unique fixed point of F within the ball ε centered at x .

I. e. there exists a unique $\sigma \in \mathfrak{R}^n$, $\|\sigma - x\| \leq \varepsilon$, such that $\sigma = F(\sigma)$.

And $\forall \sigma_0 \in \mathfrak{R}^n, \|\sigma_0 - x\| \leq \varepsilon$, the sequence

$$\sigma_1 = F(\sigma_0), \sigma_2 = F(\sigma_1), \sigma_3 = F(\sigma_2), \dots \quad (34)$$

converges to σ .

3 Untimed Digital or Synchronous Data Flow Simulator

3.1 Purpose of the Simulator

The synchronous dataflow (SDF) simulator is useful for modeling simple dataflow systems without complicated flow of control, such as signal processing systems. Under the SDF simulator, the execution order of blocks is statically determined prior to execution. This results in execution with minimal over-head, as well as bounded memory usage and a guarantee that deadlock will never occur. This simulator is specialized, and may not always be suitable.

3.2 Using SDF

There are four main issues that must be addressed when using the SDF simulator:

- ◆ Deadlock
- ◆ Consistency of data rates
- ◆ The value of the iterations parameter
- ◆ The granularity of execution

This section will present a short description of these issues. For a more complete description, see section on “Properties of SDF Simulator”.

3.2.1 Deadlock

Consider the SDF model shown in Figure 16. This block has a feedback loop from the output of the *AddSubtract* block back to its own input. Attempting to run the model results in the exception shown at the right in the Figure. The Simulator is unable to schedule the model because the input of the *AddSubtract* block depends on data from its own output. In general, feedback loops can result in such conditions.

The fix for such deadlock conditions is to use the *SampleDelay* block, shown highlighted in Figure 17. This block injects into the feedback loop an initial token, the value of which is given by the initial Outputs parameter of the block. In the Figure, this parameter has the value {0}. This is an array with a single token, an integer with value 0. A double delay with initial values 0 and 1 can be specified using a two element array, such as {0, 1}.

It is important to note that it is occasionally necessary to add a delay that is not in a feedback loop to match the delay of an in input with the delay around a feedback loop. It can sometimes be tricky to see exactly where such delays should be placed without fully considering the flow of the initial tokens described above.

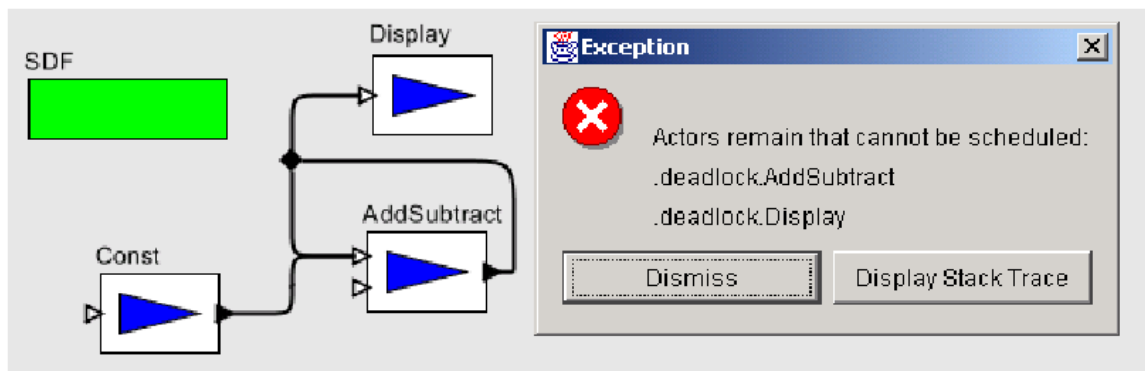


Figure 16 An SDF model that deadlocks

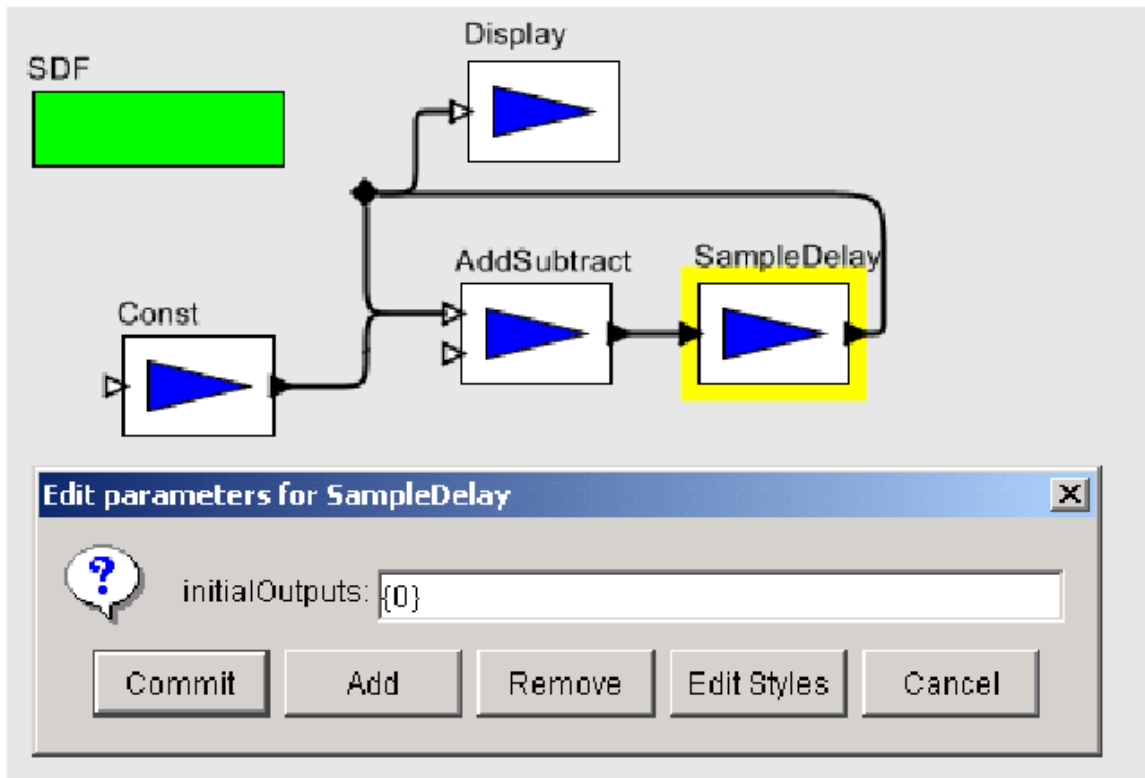


Figure 17 The model of Figure 17. 1 corrected with an instance of **SampleDelay** in the feedback loop

3.2.2 Consistency of data rates

Consider the SDF model shown in Figure 18. The model is attempting to plot a *sinewave* and its *downsampled* counterpart. However, there is an error because the number of tokens on each channel of the input port of the plotter can never be made the same. The *DownSample* block declares that it consumes 2 tokens using the *tokenConsumptionRate* parameter of its input port. Its output port similarly declares that it produces only one token, so there will only be half as many tokens being plotted from the *DownSample* block as from the *Sinewave*.

The fixed model is shown in Figure 19, which uses two separate plotters. When the model is executed, the plotter on the bottom will fire twice as often as the plotter on the top, since must consume twice as many tokens. Notice that the problem appears because one of the blocks (in this case, the *DownSample* block) produces or consumes more than one token on one of its ports. One easy way to

ensure rate consistency is to use blocks that only produce and consume one token at a time. This special case is known as homogeneous SDF. Note that blocks like the Sequence plotter which do not specify

rate parameters are assumed to be homogeneous. For more specific information about the rate parameters and how they are used for scheduling, see *Properties of the SDF Simulator-Scheduling*.

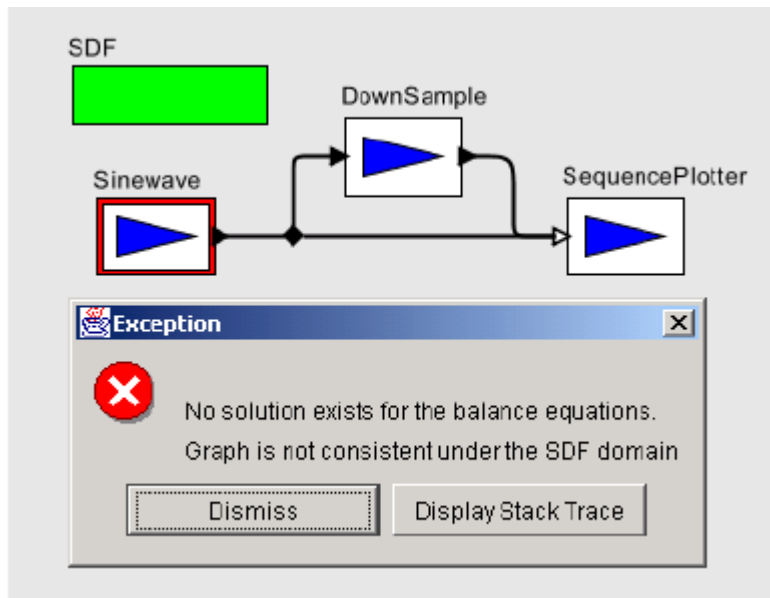


Figure 18 An SDF model with inconsistent rates.

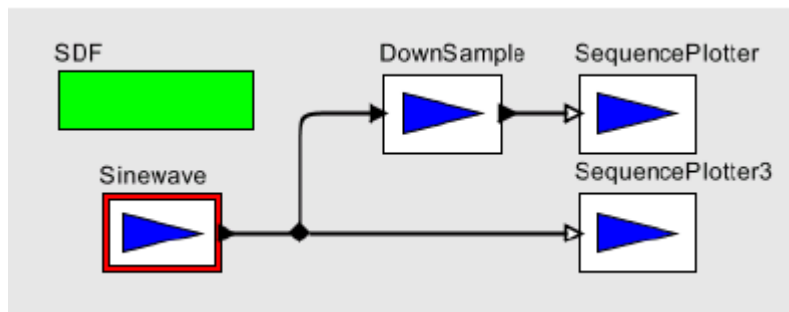


Figure 19 Figure 18 modified to have consistent rates.

3.2.3 How many iterations?

Another issue when using the SDF simulator concerns the value of the iterations parameter of the SDF Simulator. In homogeneous models one token is usually produced for every iteration. However, when token rates other than one are used, more than one interesting output value may be created for each iteration. For example, consider Figure 20 which contains a model that plots the Fast Fourier Transform of the input signal. The important point to realize about this model is that the FFT block declares that it consumes 256 tokens from its input port and produces 256 tokens from its output port, corresponding to an order 8 FFT. This means that only one iteration is required to produce all 256 values of the FFT. Contrast this with the model in Figure 21. This model plots the individual values of the signal. Here 256 iterations are necessary to see the entire input signal, since only one output value is plotted in each iteration.

3.2.4 Granularity

The granularity of execution of an SDF model is determined by the schedule as produced. As mentioned in the previous section, this schedule may involve a small or large number of firings of each block, depending on the data rates of the blocks. Generally, the smallest possible valid schedule, corresponding to the smallest granularity of execution, is the most interesting.

However, there some instances when this is not the case. In such cases the *vectorizationFactor* parameter of the SDF Simulator

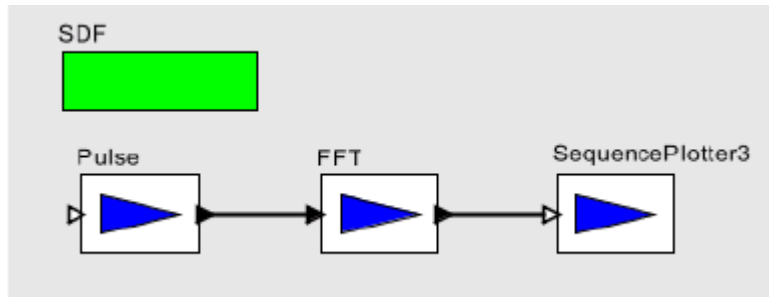


Figure 20 A model that plots the Fast Fourier Transform of a signal. A single iteration must be executed to plot all 256 values of the FFT, since the FFT block produces and consumes 256 tokens each firing.

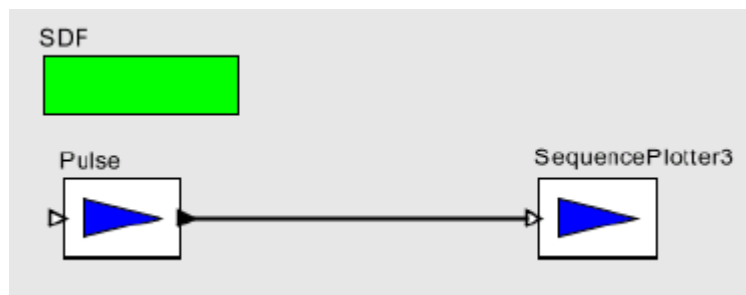


Figure 21 A model that plots the values of a signal. 256 iterations must be executed to plot the entire signal.

can be used to scale up the granularity of the schedule. A *vectorizationFactor* of 2 implies that each block is fired twice as many times as normal in the schedule.

One example when this might be useful is the modeling of block data processing. For instance, we might want to build a model of a signal processing system that filters blocks of 40 samples at a time using an FIR filter. Such a block could be written in Java, or it could be built as a hierarchical SDF model, using a single sample FIR filter, as shown in Figure 22. The *vectorizationFactor* parameter of the Simulator is set to 40. Here, each firing of the SDF model corresponds to 40 firings of the single sample FIR filter.

Another useful time to increase the level of granularity is to allow *vectorized* execution of blocks. Some blocks override the *iterate()* method to allow optimized execution of several consecutive firings.

Increasing the granularity of an SDF model can provide more opportunities for the SDF Simulator to perform this optimization, especially in models that do not have fine-grained feedback.

3.3 Properties of the SDF simulator

SDF is an untimed model of computation. All blocks under SDF consume input tokens, perform their computation and produce outputs in one atomic operation. If an SDF model is embedded within a timed model, then the SDF model will behave as a zero-delay block.

In addition, SDF is a statically scheduled simulator. The firing of a composite block corresponds to a single iteration of the contained model. SDF iteration consists of one execution of the pre-calculated SDF schedule. The schedule is calculated so that the number of tokens on each relation is the same at the end of each iteration as at the beginning. Thus, an infinite number of iterations can be executed, without deadlock or infinite accumulation of tokens on each relation. Execution in SDF is extremely efficient because of the scheduled execution. However, in order to execute so efficiently, some extra information must be given to the scheduler. Most importantly,

the data rates on each port must be declared prior to execution. The data rate represents the number of tokens produced or consumed on a port during every firing³. In addition, explicit data delays must be added to feedback loops to prevent deadlock. At the beginning of execution, and any time these data rates change, the schedule must be recomputed. If this happens often, then the advantages of scheduled execution can quickly be lost.

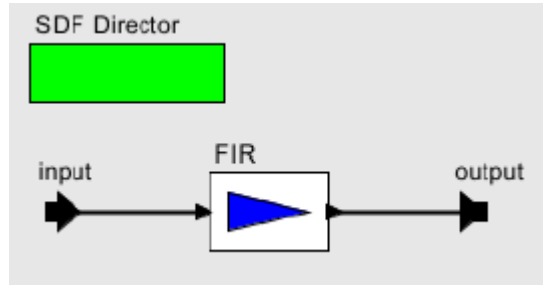


Figure 22 A model that implements a block FIR filter. The *vectorizationFactor* parameter of the Simulator is set to the size of the block.

3.3.1 Scheduling

The first step in constructing the schedule is to solve the balance equations. These equations determine the number of times each block will fire during iteration. For example, consider the model in Figure 23. This model implies the following system of equations, where *ProductionRate* and *ConsumptionRate* are declared properties of each port, and *Firings* is a property of each block that will be solved for:

$$Firings(A) \times ProductionRate(A1) = Firings(B) \times ConsumptionRate(B1)$$

$$Firings(A) \times ProductionRate(A2) = Firings(C) \times ConsumptionRate(C1)$$

$$Firings(C) \times ProductionRate(C2) = Firings(B) \times ConsumptionRate(B2)$$

These equations express constraints that the number of tokens created on a relation during iteration is equal to the number of tokens consumed. These equations usually have an infinite number of linearly dependent solutions, and the least positive integer solution for *Firings* is chosen as the firing vector, or the repetitions vector.

The second step in constructing an SDF schedule is dataflow analysis. Dataflow analysis orders the firing of blocks, based on the relations between them. Since each relation represents the flow of data, the block producing data must fire before the consuming block. Converting these data dependencies to a sequential list of properly scheduled blocks is equivalent to topologically sorting the SDF Block Diagram, if the graph is acyclic⁴. Dataflow graphs with cycles cause somewhat of a problem, since such graphs cannot be topologically sorted. In order to determine which block of the loop to fire first, a data

delay must be explicitly inserted somewhere in the cycle. This delay is represented by an initial token created by one of the output ports in the cycle during initialization of the model. The presence of the delay allows the scheduler to break the dependency cycle and determine which block in the cycle to fire first. In VisualSim, the initial token (or tokens) can be sent from any port, as long as the port declares an *initProduction* property. However, because this is such a common operation in SDF, the Delay block is provided that can be inserted in a feedback loop to break the

³ This is known as multirate SDF, where arbitrary rates are allowed. Not to be confused with homogeneous SDF, where the data rates are fixed to be one.

⁴ Note that the topological sort does not correspond to a unique total ordering over the blocks. Furthermore, especially in multirate models it may be possible to interleave the firings of blocks that fire more than once. This can result in many possible schedules that represent different performance trade-offs. We anticipate that future schedulers will be implemented to take advantage of these tradeoffs.

cycle. Cyclic graphs not properly annotated with delays cannot be executed under SDF. An example of a cyclic graph properly annotated with a delay is shown in Figure 24. In some cases, a non-zero solution to the balance equations does not exist. Such models are said to

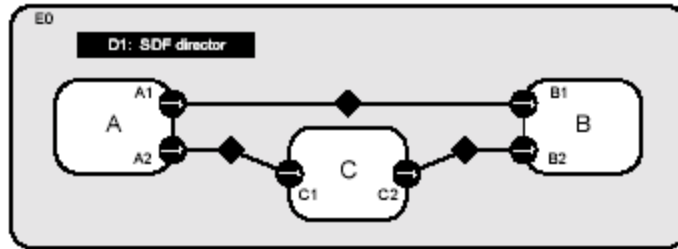


Figure 23 An example SDF model.

be inconsistent, and cannot be executed under SDF. Inconsistent graphs inevitably result in either deadlock or unbounded memory usage for any schedule. As such, inconsistent graphs are usually bugs in the design of a model. Examples of consistent and inconsistent graphs are shown in Figure 25.

3.3.2 Hierarchical Scheduling

So far, we have assumed that the SDF graph is not hierarchical. The simplest way to schedule a hierarchical SDF model is flatten the model to remove the hierarchy, and then schedule the model as

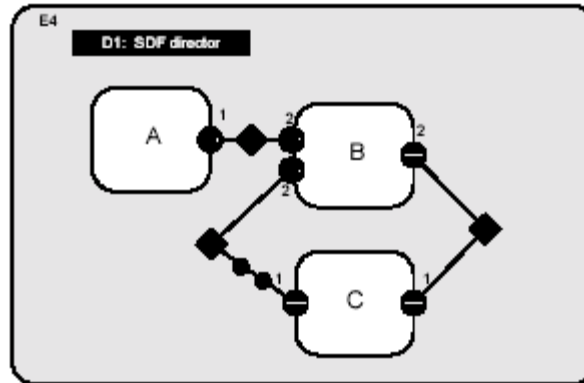


Figure 24 A consistent cyclic graph, properly annotated with delays. A one token delay is represented by a black circle. Block C is responsible for setting the *tokenInitProduction* parameter on its output port, and creating the two tokens during initialization. This Block Diagram can be executed using the schedule A, A, B, C, C.

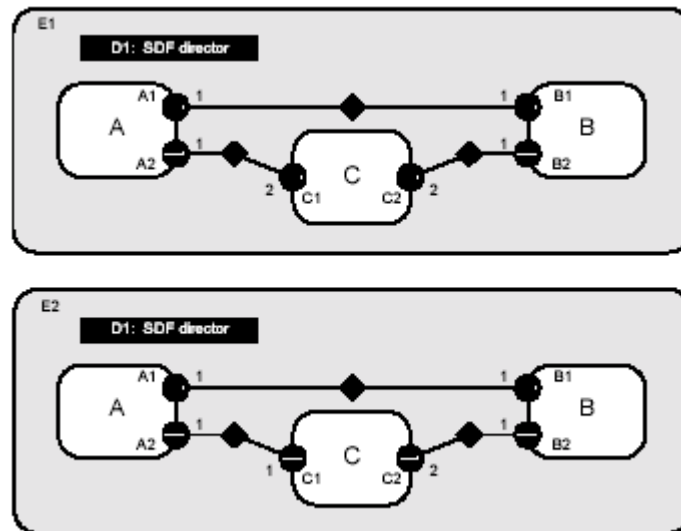


Figure 25 Two models, with each port annotated with the appropriate rate properties. The model on the top is consistent, and can be executed using the schedule A, A, C, B, B. The model on the bottom is inconsistent because tokens will accumulate between ports C2 and B2.

usual. This technique allows the most efficient schedule to be constructed for a model, and avoids certain composing problems when creating hierarchical models. In VisualSim, a model created using a transparent composite block to define the hierarchy is scheduled in exactly this way. VisualSim also supports a stronger version of hierarchy, in the form of opaque composite blocks. In this case, the hierarchical block appears to be no different from the outside than an atomic block with no hierarchy. The SDF simulator does not have any information about the contained model, other than the rate parameters that may be specified on the ports of the composite block. The SDF simulator is designed so that it automatically sets the rates of external ports when the schedule is computed. Most other Simulators are designed (conveniently enough) so that their models are compatible with default rate properties assumed by the SDF simulator.

3.3.3 Hierarchically Heterogeneous Models

An SDF model can generally be embedded in any other simulator. However, SDF models are unlike most other hierarchical models in that they often require multiple inputs to be present. When building one SDF model inside another SDF model, this is ensured by the containing SDF model because of the way the data rate parameters are set as described in the previous section. For most other Simulators, the SDF Simulator will check how many tokens are available on its input ports and will refuse firing (by returning false in `prefire()`) until enough data is present for an entire iteration to complete.

4 FSM Simulator

4.1 Introduction

Finite state machines (FSMs) have been used extensively in designing sequential control logic. There are two major reasons behind their use. First, FSMs are a very intuitive way to capture control logic and make it easier to communicate a design. Second, FSMs have been the subject of a long history of research work. Many formal analysis and verification methods have been developed for them.

In their simple flat form, FSM models have a key weakness: the number of states in an FSM model can get quite large even for a moderately complex system. Such models quickly become chaotic and incomprehensible when one tries to model a system having many concurrent activities. The problem can be solved by introducing hierarchical organization into FSM models and using them in combination with concurrency models. David Harel first used this approach when he introduced the Statecharts formalism. The Statecharts formalism extends the conventional FSM model in three aspects: hierarchical decomposition of states, concurrent composition of FSMs in a synchronous-reactive fashion, and a broadcast communication mechanism between concurrent components. While how these extensions fit together was not completely specified, Harel's work stimulated a lot of interest in the approach.

Consequently, there is a proliferation of variants of the Statecharts formalism, each proposing a different way to make the extensions fit into a monolithic model. Unfortunately, in all these variants FSM is combined with a particular concurrency model. The applicability of the resulting models is often limited.

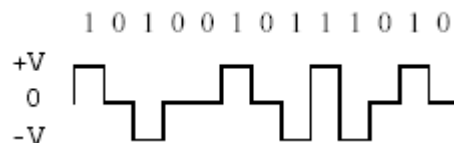
Based on the VisualSim philosophy of hierarchical composition of heterogeneous models of computation, the *charts⁵ formalism allows embedding hierarchical FSMs within a variety of concurrency models. If tight synchronization is possible and desirable, then FSMs can be composed by the synchronous-reactive model. If the system has a global notion of time and components communicate by time-stamped events, then FSMs can be composed by the discrete-event model. The rest of this chapter focuses on how the FSM simulator in VisualSim supports the *charts formalism.

4.2 Building FSMs in ModelBuilder

An FSM model is contained by an instance of FSM-Controller block, located in the FSM directory. The FSM model reacts to inputs to the FSM block by making state transitions. Actions such as sending tokens to the output ports of the FSM block can be associated with state transitions. In this section, we show how to construct and run a model with an FSM block in ModelBuilder.

4.2.1 Alternate Mark Inversion Coder

Alternate Mark Inversion (AMI) is a simple digital transmission technique that encodes a bit stream on a signal line as shown below:



The 0 bits are transmitted with voltage zero. The 1 bits are transmitted alternately with positive and negative voltages. On average, the resulting waveform will have no DC component.

⁵ Pronounced "starcharts." The star represents a wildcard that can be interpreted as matching multiple concurrency models.

We can model an AMI coder with a two-state FSM shown in Figure 27. To construct a VisualSim model containing this coder, follow these steps:

1. Start ModelBuilder; open a Block Diagram editor by selecting File -> New -> Block Diagram Editor.
2. From utilities in the palette on the left, drag an FSM block to the Block Diagram. Rename the FSM block *AMICoder*.
3. Right click on *AMICoder*, select Configure Ports. Add an input port with name *in* and an output port with name *out* to *AMICoder*.
4. Right click on *AMICoder*, select Look Inside. This will open an FSM editor for *AMICoder*. Note that the ports of *AMICoder* are placed at the upper left corner of the Block Diagram panel.
5. From the palette on the left, drag a state to the Block Diagram, rename it Positive. Drag another state to the Block Diagram, rename it Negative.
6. Control-drag from the Positive state to the Negative state to create a transition.
7. Double click on the transition. This will bring up the dialog box shown in Figure 26 for editing the parameters of the transition.
8. Set *guardExpression* to *in == 1*, and *outputActions* to *out = 1*.
9. Create a transition from the Positive state back to itself with guard expression *in == 0* and output action *out = 0*.
10. Create a transition from the Negative state back to itself with guard expression *in == 0* and output action *out = 0*.

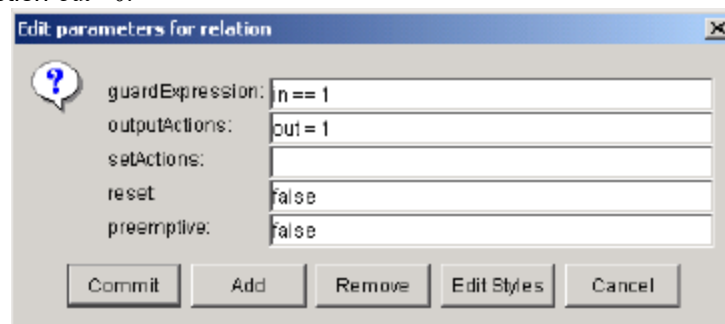


Figure 26 The dialog box for editing parameters of a transition.

11. Create a transition from the Negative state to the Positive state with guard expression *in == 1* and output action *out = -1*.
12. Right click on the background of the Block Diagram panel. Select Edit Parameters from the context menu. This will bring up the dialog box for editing parameters of *AMICoder*. Set *initialStateName* to *Positive*.
13. The construction of *AMICoder* is complete. It will look like what is shown in Figure 27.
14. Return to the Block Diagram Editor opened in step 1.
15. Drag a Pulse block (from block library, basic/sources), a SequencePlotter (from model library, display), and an SDF Simulator (from Simulator library) to the Block Diagram.
16. Connect the blocks as shown in Figure 28.
17. Edit parameters of the Pulse block: set indexes to {0, 1, 2, 3, 4, 5}; set values to {0, 1, 1, 1, 0, 1}.

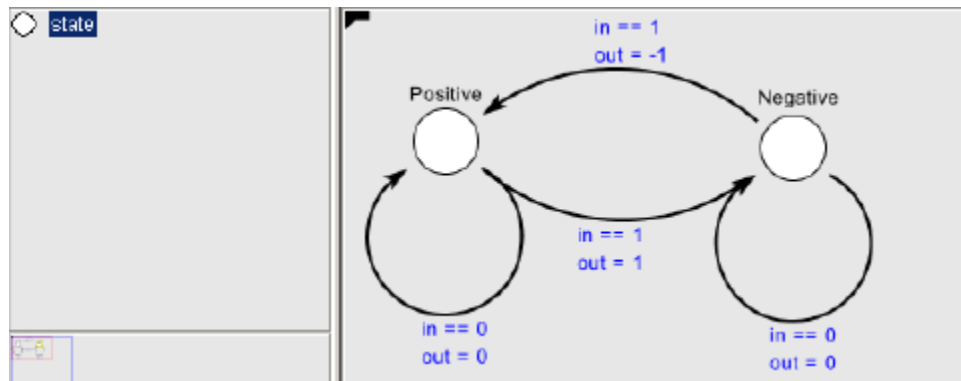


Figure 27 ModelBuilder FSM editor showing the AMICoder.

18. The model construction is complete.
19. Select View -> Run Window from the menu. Set Simulator iterations to 6 and execute the model. For a better display of the result, open the set plot format dialog box, unselect connect and use various marks.

4.3 The Implementation of FSMActor

The *FSMActor* class extends the *CompositeEntity* class and implements the *TypedActor* interface. An FSM block contains states and transitions. The State class is a subclass of *ComponentEntity*. A State has two ports: *incomingPort*, which links to incoming transitions to the state, and *outgoingPort*, which links to transitions going out from the state. The Transition class is a subclass of *ComponentRelation*. A transition links to exactly two ports: the outgoing port of its source state, and the incoming port of its destination state.

4.3.1 Guard Expressions

The guard of a transition is specified by its *guardExpression* string attribute. Guard expressions are parsed and evaluated using the VisualSim expression language (see the Expressions chapter for details). Guard expressions should evaluate to a *boolean* value. A transition is enabled if its guard expression evaluates to true. Parameters of the FSM block and input variables (defined below) can be used in guard expressions.

Input variables represent the status and input value for each input port of the FSM block. If the input port is a single port, two variables are used: a status variable named *portName_isPresent*, and a value variable named *portName*. If the input port is a multiport of width *n*, 2*n* variables are used, two for each channel: a status variable named *portName_channelIndex_isPresent*, and a value variable named *portName_channelIndex*. The status variables will have boolean value true if there is a token at the corresponding input, or false otherwise. The value variables have the same type as the corresponding input, and contain the token received from the input, or null if there is no token. All input variables are contained by the FSM block. In the following examples (and the examples in the next section), we assume that the FSM block has two input ports: a single port *in1* and a *multiport* *in2* of width 2; an output port *out* that is a multi-port of width 2; and a parameter *param*.

- ◆ Guard expression: $in2_0 + in2_1 > 10$. If the inputs from the two channels of port *in2* have a total greater than 10, the transition is enabled. Note that if one or both channels of port *in2* do not have a token when this expression is evaluated, an exception will be thrown.

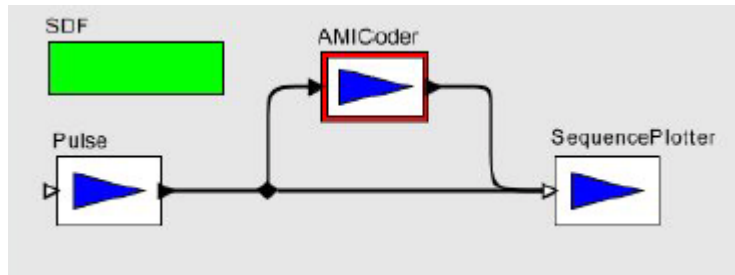


Figure 28 An SDF model with the AMICoder.

- ◆ Guard expression: `in1_isPresent && in1 > param`. If there is input from port `in1` and the value of the input is greater than `param`, the transition is enabled.

4.3.2 Actions

A transition can have a set of actions that produce output tokens or set parameters of the FSM block. To make FSM blocks simulator polymorphic, especially for them to be operational in Simulators having fixed-point semantics, two kinds of actions are defined: choice actions and commit actions. Choice actions do not modify the extended state 1 of the FSM block. They are executed when the FSM block is fired and the containing transition is enabled. Commit actions may modify the extended state of the FSM block. They are executed in *postfire()* if the containing transition was enabled in the last firing of the FSM block. Two marker interfaces are defined in the FSM kernel package:

- ◆ *ChoiceAction*, which is implemented by all choice action classes, and *CommitAction*, implemented by all commit action classes.

A transition has an *outputActions* attribute which is an instance of *OutputActionsAttribute*. The *OutputActionsAttribute* class allows the user to specify a list of semicolon separated output actions of the form `destination = expression`. The expression can use parameters of the FSM block and input variables. The destination is either a port name, in which case the result token from evaluating the expression is broadcast to all channels of the port, or of the form `portName(channelIndex)`, in which case the result token is sent to the specified channel. Output actions are choice actions.

- ◆ `outputActions: out = in1_isPresent ? in1 : 0`. Broadcast the input from port `in1`, or 0 if there is no input from `in1`, to the two channels of `out`.
- ◆ `outputActions: out(0) = param; out(1) = param + 1`. Send the value of `param` to the first channel of `out`, and the value of `param` plus 1 to the second channel.

A transition has a *setActions* attribute which is an instance of *CommitActionsAttribute*. The *CommitActionsAttribute* class allows the user to specify a list of semicolon separated commit actions of the form `destination = expression`. The expression can use parameters of the FSM block and input variables. The destination is a parameter name.

- ◆ `setActions: param = param + (in1_isPresent ? in1 : 0)`. The input values from port `in1` are accumulated in `param`.

It is worth noting that parameter values are persistent. If not properly initialized, the parameter *t* in the above example will retain its accumulated value from previous model executions. A useful approach is to build the FSM model such that the initial state has an outgoing transition with guard expression `true`, and use the set actions of this transition for parameter initialization.

Execution

The methods that define the execution of an FSM block are implemented as follows:

- ◆ `preinitialize()`: create receivers and input variables for each input port; set current state to the initial state as specified by the `initialStateName` attribute.

- ◆ `initialize()`: perform simulator-specific initialization by calling the `initialize (Actor)` method of the Simulator. Note that in the example given in AMI Coder, the Simulator will be the SDF Simulator.
- ◆ `prefire()`: always return true. An FSM block is always ready to fire.
- ◆ `fire()`: set the values of input variables; choose the enabled transition among the outgoing transitions of the current state; execute the choice actions of the chosen transition.
- ◆ `postfire()`: execute the commit actions of the last chosen transition; change state to the destination state of that transition.

Non-deterministic FSMs are currently not allowed. The `fire()` method checks whether there is more than one enabled transition from the current state. An exception is thrown if there is. In the case when there is no enabled transition, the FSM will stay in its current state.

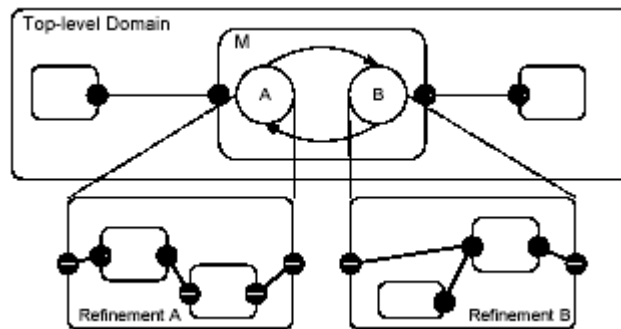


Figure 29 A Hierarchical FSM example.

4.4 FSM-Hierarchical

The FSM simulator supports the *charts formalism with FSM-Hierarchical. The concept of FSM-Hierarchical is illustrated in Figure 29. M is a FSM-Hierarchical with two operation modes. The modes are represented by states of an FSM that controls mode switching. Each mode has a refinement that specifies the behavior of the mode. In VisualSim, a FSM-Hierarchical⁶ is constructed in a typed composite block having the FSM Simulator as local Simulator. The composite block contains a mode controller (an FSM block) and a set of blocks that model the refinements. The FSM Simulator mediates the interaction with the outside simulator, and coordinates the execution of the refinements with the mode controller.

4.4.1 A Schmidt Trigger Example

In this section, we will illustrate how to build a modal model in VisualSim with a simple Schmidt trigger example. The output from the Schmidt trigger will move from -1.0 to 1.0 when its input becomes greater than 0.3, and will move back to -1.0 once its input becomes less than -0.3.

- ◆ Open a ModelBuilder Block Diagram Editor. From utilities, drag a typed composite block to the Editor, rename it SchmidtTrigger. Add an input port named `in` and an output port named `out` to it.
- ◆ Look inside SchmidtTrigger. This will open a Block Diagram editor for it. In this Block Diagram editor, drag an FSM block to the Block Diagram, rename it Controller. Drag a typed composite block to the Block Diagram, rename it RefinementP. Drag another typed composite block to the Block Diagram, rename it RefinementN.
- ◆ Add an input port named `in` to Controller. Add an output port named `out` for both RefinementP and RefinementN.

⁶ The current software architecture that supports modal models is experimental. A new approach based on higher order functions is in progress.

- ◆ Look inside Controller. This will open an FSM editor for it. In this FSM editor, construct a two-state FSM as shown in Figure 30. Set the reset parameter of both transitions to true. Set refinement name of state P to `RefinementP`. Set refinement name of state N to `RefinementN`. Set initial state name of Controller to N.

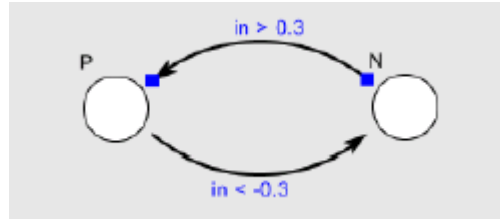


Figure 30 The mode controller for SchmidtTrigger.

Back to the Block Diagram editor for SchmidtTrigger. Look inside RefinementP. Build a model for it as shown in Figure 31. Set the value of Const to 1.0. Edit parameters of Pulse: set indexes to {0, 1, 2, 3, 4}, and values to {-2.0, -1.6, -1.2, -0.8, -0.4}. Back to the Block Diagram editor for SchmidtTrigger. Look inside RefinementN. Build a model for it as shown in Figure 31. Set the value of Const to -1.0. Edit parameters of Pulse: set indexes to {0, 1, 2, 3, 4}, and values to {2.0, 1.6, 1.2, 0.8, 0.4}. Back to the Block Diagram editor for SchmidtTrigger. Drag an FSM Simulator to the Block Diagram. Set its controller-Name to `Controller`. Connect the blocks as shown in Figure 32. Back to the Block Diagram editor opened in step 1. Build the model as shown in Figure 33. The model generates an input signal (a sinusoid plus Gaussian noise) for the SchmidtTrigger and plots its output. Edit parameters of Ramp: set init to $-\pi/2$, and step to $\pi/20$. Edit parameters of Gaussian: set standardDeviation to 0.2. Run the model for 200 iterations. A sample result is shown in Figure 34.

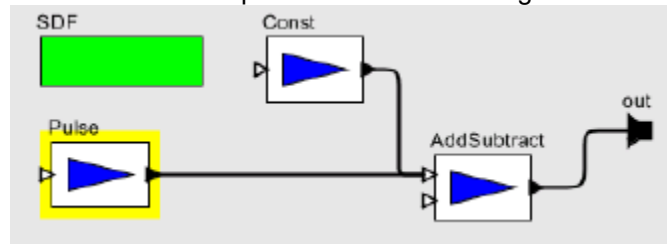


Figure 31 Model for the refinements in SchmidtTrigger.

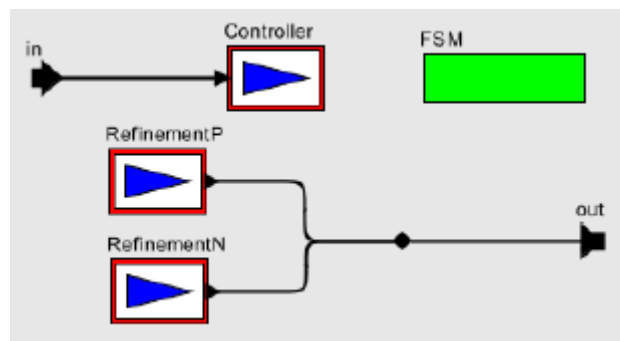


Figure 32 The SchmidtTrigger modal model.

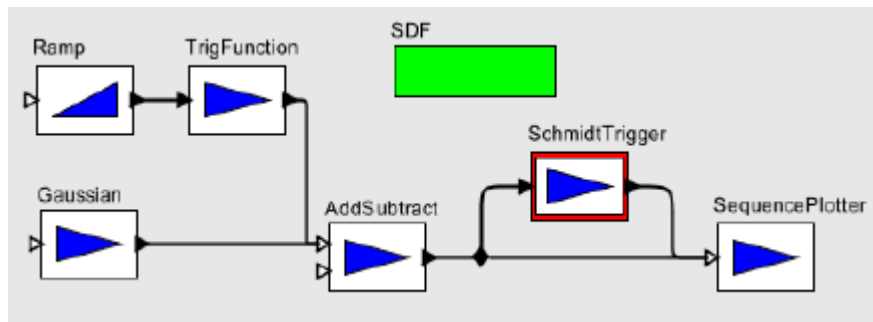


Figure 33 The top-level model with the SchmidtTrigger.

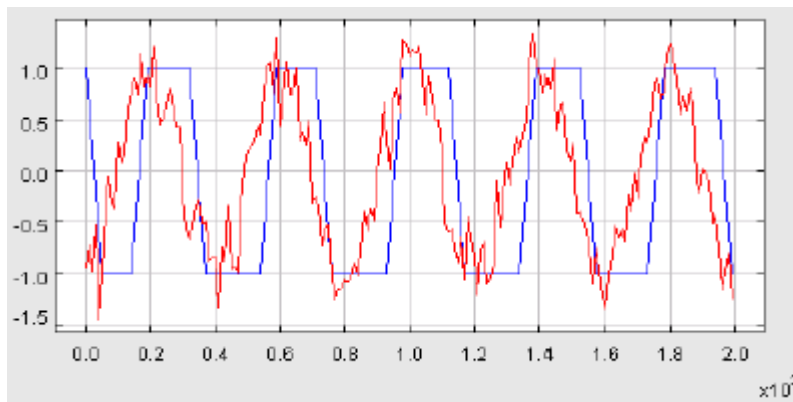


Figure 34 Sample result of the model shown in Figure 33.

4.4.2 Applications

Hybrid System Modeling. An *HSSimulator* class that extends the *FSMSimulator* class is created for modeling hybrid systems with FSMs and continuous-time (CT) models. An example and Execution control is presented in the Continuous Time Simulator section.

Communication Protocol Modeling. Hierarchical FSMs are used to model protocol control logic. The timing characteristics of the communication channel are captured by discrete-event (DE) models. We have applied this approach to the alternating bit protocol.

Chapter 2 Modeling Libraries

Introduction to Resources

Resources are blocks that are used to define the physical entities of a system. These can be used Queue in an M/M/1 definition. Examples of these can be a Bank Teller, a conveyer belt or a call center agent. In product setting, this can be RTOS Queue, RTOS scheduling, multiple virtual machines for a middleware, processor, memories, peripherals, network nodes or wireless channel. In a distributed system, a aircraft or a set of ammunition can be a resource.

There are two types of resources- Active which consumes time and passive which consumes resources. A processor is an active resource while a parking lot with spaces for different types of cars is a passive resource. A memory is a passive resource with a active controller and a active time to get access to the passive content.

Active Resources

The **Event Queue** blocks, **Timed Queue** blocks, the advanced capabilities of the **System Resources (a.k.a Schedulers)** and the extendable **Channel** blocks provide tiered resource modeling options in the Block Diagram.

The Event and Timed Queues blocks consist of the blocks shown in Figure 1. The Scheduler blocks are shown separately in Figure 2. One will notice that the Event and Timed Queues blocks have a lot of inputs and outputs that are required for proper operation of these blocks.

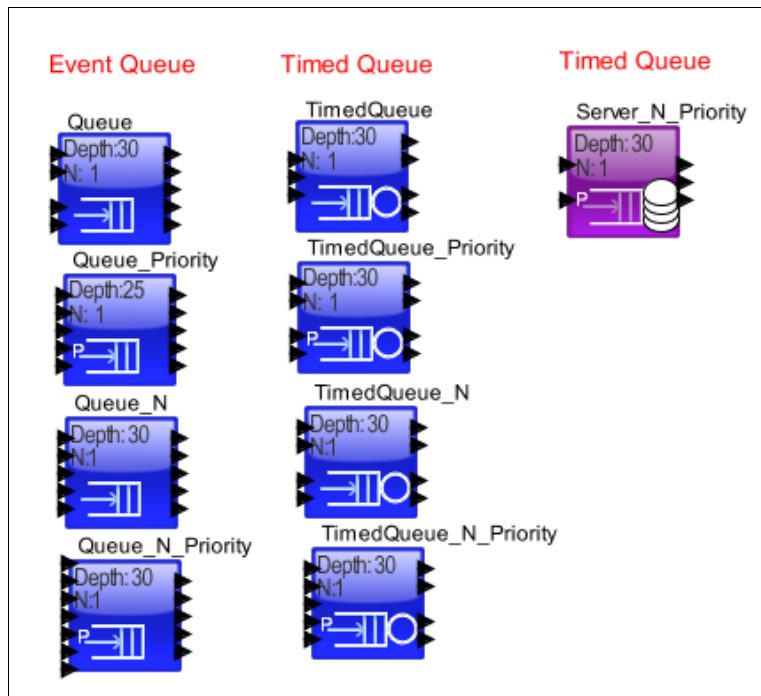


Figure 35 VisualSim Resource Library

Event Queue Blocks

The operation of the queue is straightforward. The Event **Queue** takes the input tokens and stores it in a queue. When a token is received in 'pop_input' port, the token at the head of the queue is sent on the output port.

All the Data Structure or numerical Token input enter and exit on the top input (upper-left) and output ports (upper-right) respectively. Each block has a 'stats_input' on the lower left and a 'stats_output' on the lower right. Each queue block also has a 'reject_queue' port, just below the output port on the upper right side. An incoming composite Data Structure can be rejected if the queue length has been exceeded. In the case of the two priority queues, one can select a reject mechanism that will reject the lowest priority Data Structure.

Each queue also has a queue length output port, above the 'stats_output' port on the lower right side. Each time a queue input actions is initiated, then the queue length will be sent to the 'queue_length' output port. Each queue has an output port for obtaining copies of queue elements, the middle output port named 'copy_queue_n_output'.

Each queue has a 'pop_queue' input port that will send the next Data Structure in the queue, including the 'queue_number_input' for multi-dimensional queues. There is also a 'copy_queue_n_input' for selecting a particular queue element. The two priority queues have a priority input that must arrive for a new queue element.

The only asymmetrical item for these blocks is the 'set_queue_input' port, which allows one to modify an existing queue element. This port only appears on the **Queue-Priority_N** Block.

To better understand the use of these Queues blocks, view the Queue Usage Page [here](#). This uses the Queue-Priority block. Pay special attention on the arrival of the data at the Queue input. The required inputs must all exist before the Data Structure will enter the queue.

Timed Queue Resource Blocks

The Timed Queue Resource Blocks, namely **Timed_Queue**, **Timed_Queue-N**, **Timed_Queue-Priority**, and **Timed_Queue-Priority_N**, are logically the combination of two fundamental components: a queue and a server. The key difference between the event Queue blocks and time-based Queue blocks is that the Timed pops the queued Data Structure out based on an associated time value. The time value is entered along with the input token. When the token reaches the front of the queue, it is delayed by the associated time value and then popped on the output port.

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 simplest block, **Timed_Queue**, can implement an M/M/1 queue by providing exponential inter-arrival time to the "time_input" port, for example. If the internal queue is at the capacity set by the block level parameters, then the incoming Data Structure will exit via the "FCFS_reject_output" port.

A typical modeling use for the basic **Timed_Queue** Block is to implement a bus queue, where the "time_input" represents a value that combines the bus transaction size divided by the data rate of the bus to generate seconds. If the **Timed_Queue-Priority** Block is used, then the underlying bus can process the bus transactions with priorities.

A user defined Data Structure enters the **Timed_Queue** Blocks on one port, the server time on a second port, the priority (optional) for the incoming Data Structure, and the specific queue

(optional) that the Data Structure is destined. The **Timed_Queue** Blocks has parameters to change the type of internal queue (FIFO/LIFO), the length of the internal queue, and the dimension of multiple queue blocks (denoted by “N”). There is also a “stats” input and output port for each Resource Block to collect resource pertinent statistics internal to the block.

All the Data Structure or numerical Token input enter and exit on the top input (upper-left) and output ports (upper-right) respectively. Each block has a 'stats_input' on the lower left and a 'stats_output' on the lower right. Each queue block also has a 'reject_queue' port, just above the stats port on the lower right side. An incoming Data Structure is rejected if the queue length has been exceeded.

Each queue also has a queue length output port, below the ds_output' port on the upper right side. Each time a queue input actions is initiated, then the queue length will be sent to the 'queue_length' output port.

Unlike the **Event_Queue-Priority_N** Block, the **Timed_Queue-Priority_N** Block lacks the capability to overwrite an element in the queue.

Server_N_Priority

There is a special block called the **Server_N_Priority** that has an additional dispatch queue in front of the Servers within the block. The individual server queues have been fixed to 1 unit. This is useful where a shared resource need to schedule the DS to an array of servers.

To better understand the use of these **Timed_Queue** blocks, view the Usage Page here. This shows the use of all the **Timed_Queue** blocks. Pay special attention on the arrival of the data at the input. The required inputs must all exist before the Data Structure will enter the queue.

System Resource blocks(a.k.a Scheduler Blocks)

System Resource Blocks model the execution of a general timing resource at an architectural level. While the **TimedQueue** blocks are used at the queuing-level of abstraction, the System Resource blocks are used at the architectural-level with more complex scheduling algorithms. Complex scheduling algorithms that are not simply time-based and accept request from multiple model points require the System Resource blocks. These blocks separate the behavior and architectural aspects of a system, and provide a way for separate design groups to implement different parts of a model, if needed.

The **Mapper_Adv/ Mapper** is used to make the request (behavior) to the **System Resource_Extended (a.k.a Scheduler_HW)** and **System_Resource (a.k.a Scheduler_SW)** that model the processing (architecture) side. The System Resource blocks, namely **SystemResource_Extended** and **System Resource blocks**, perform a variety of scheduling algorithms, including First-Come-First-Serve, First-Come-First-Serve plus pre-emption, Round Robin, or User defined schedulers. In addition, the **SystemResource** Block can be arranged as a hierarchical set of schedulers. The **SystemResource_Extended** Block supports detailed external modeling of the task execution, whereas the software scheduler only executes internal to the scheduler block itself. Scheduler execution is in terms of the 'Task_Time' field of the incoming data structure, and the user can set this to be interpreted as either relative time or clock cycles by a parameter entry of the **SystemResource** or **System_Resource_Extended** Blocks.

The Task data structures enter the SystemResource via a **Mapper_Adv/ Mapper** Block. The **Mapper_Adv/ Mapper** Block can contain values in the parameter fields that effect scheduler operation, or the names of data structure fields entering the **Mapper_Adv/ Mapper_Basic** Block.

The ability to use parameter entries or fields of incoming data structures provides for the concept of static and dynamic mapping within the model topology.

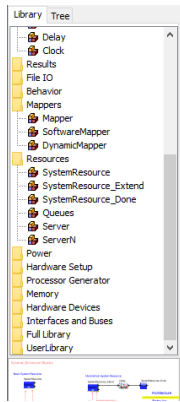
The 'Parent_Scheduler' field of the **Mapper_Adv/ Mapper** Block determines the scheduler to be used, whether **SystemResource** or **SystemResource_Extended** Blocks. This form of internal mapping is akin to a virtual node concept, where any **Mapper_Adv/ Mapper_Basic** Block in the model can address any scheduler in the model. Multiple **Mapper_Adv/ Mapper** blocks can make requests to a single SystemResource block thus establishing multiple request points in the behavior flow. The **SystemResource** and **SystemResource_Extended** do not require an input trigger but are virtually mapped from the **Mapper_Adv/ Mapper** block that is in the behavior flow through dynamic name matching of the 'Scheduler_Name' parameter.

After the Task_Time is complete within the **SystemResource_Extended**, the task is sent on the output port for additional processing. When the task encounters a **SystemResource_Done** (a.k.a **Scheduler_Release**) it immediately returns the task as complete this Scheduler and also down the lower-level Schedulers/Mapper blocks. The task can flow out of the **SystemResource_Done** and continue with additional logic. There can be multiple execution sequences after the output port. Each can have a **SystemResource_Done** anywhere in the flow.

Usage: The **SystemResource** is used to model software and hardware elements that do not require 'refinement'. This block supports pre-emption and can be called hierarchically. The **SystemResource_Extended** model the top-level architecture element that also requires 'refining'. This block does not support pre-emption. The SystemResource_Extended must always be the top-level of the hierarchy and provides for an extension of the block for refinement of the architecture element.

Practical Application: Consider the case where an application makes a request to a driver that request a processing event on an RTOS. This would be modeled with the **Mapper_Adv/ Mapper** (Behavior) sending a request to a top-level SystemResource (Driver), which would send the request to a next-level SystemResource (RTOS) that would send it to the Top-level SystemResource_Extended (CPU). The request will be sent up the flow and then back down before being returned to the Task_Issue to be processed further.

Examples: In an automotive system, one might wish to inject intentional errors into the system to see the effect of backup, or redundant, processors on overall system recovery response. Without **virtual connection** scheduler mapping, this becomes a more complex task. VisualSim also provides an internal addressing infrastructure that can be used in conjunction with the Scheduler Resource Blocks. Other examples of systems that would need the scheduler blocks are Real Time Operating Systems (RTOS), hardware interrupt processing or advanced switch fabric. Most hardware and software elements such as CPU, BUS, Cache, Memory, ROTS and drivers can be modeled using this group of blocks.



System_Resource Blocks

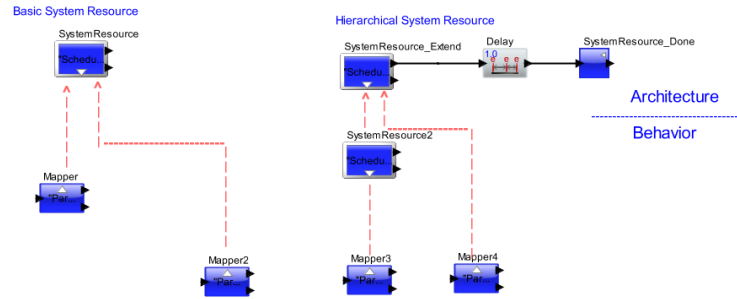


Figure 36 VisualSim Scheduler Library

Channel Blocks

The Channel blocks models fixed number of channels. The key difference between the **Channel** and the **Channel_Priority** is the additional priority parameter that is used to determine the next task to be scheduled on the channel. The Channel_N block has a separate queue for each of the Channels while the other two blocks have a single input channel. The Channel blocks can be used to model a multiple communication channel, DMA channel to memory, software task sequence based on channel and the type of transaction, and virtual channels.

The Channel blocks provide the infrastructure to extend the ability of the Queue and TimedQueue. Here the implementation details can be logic plus delay. The '**Channel_Release**' Block can model re-transmissions through it's two inputs: 'accept_input' for no re-transmission and 'reject_input' that calls back the Channel block to retransmit a packet that is rejected due to noise (communication channel) or buffer overflow (memory).

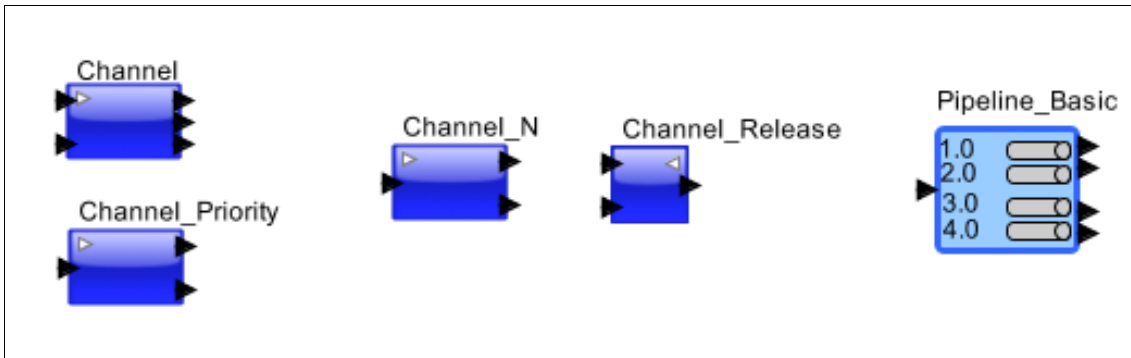
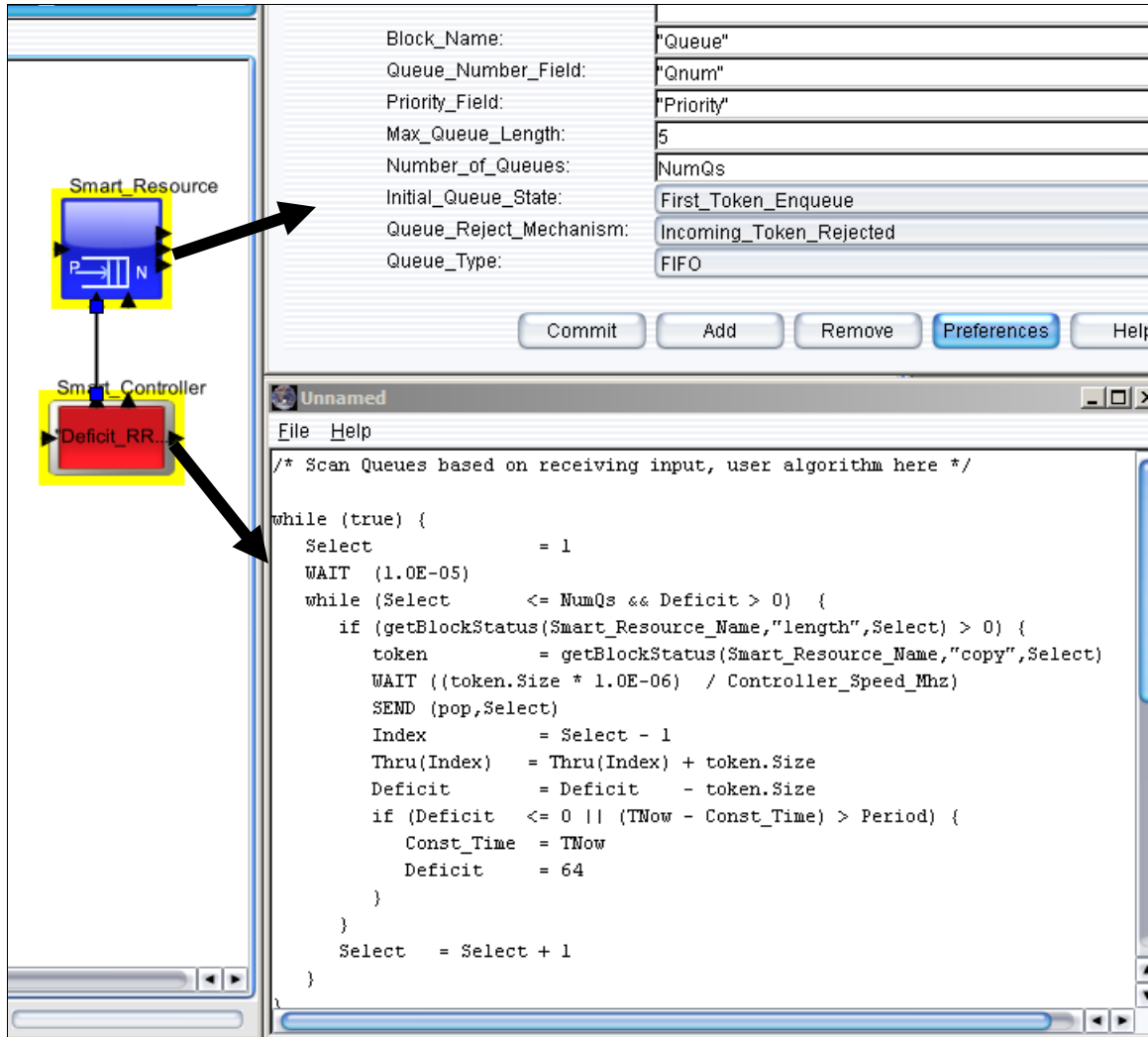


Figure 37 VisualSim Channel and Pipeline blocks

Queues (a.k.a Smart_Resource)



Block_Name:	"Queue"
Queue_Number_Field:	"Qnum"
Priority_Field:	"Priority"
Max_Queue_Length:	5
Number_of_Queuees:	NumQs
Initial_Queue_State:	First-Token_Enqueue
Queue_Reject_Mechanism:	Incoming-Token_Rejected
Queue_Type:	FIFO

```

/* Scan Queues based on receiving input, user algorithm here */
while (true) {
  Select          = 1
  WAIT (1.0E-05)
  while (Select   <= NumQs && Deficit > 0) {
    if (getBlockStatus(Smart_Resource_Name,"length",Select) > 0) {
      token          = getBlockStatus(Smart_Resource_Name,"copy",Select)
      WAIT ((token.Size * 1.0E-06) / Controller_Speed_Mhz)
      SEND (pop,Select)
      Index          = Select - 1
      Thru(Index)    = Thru(Index) + token.Size
      Deficit         = Deficit - token.Size
      if (Deficit <= 0 || (TNow - Const_Time) > Period) {
        Const_Time   = TNow
        Deficit       = 64
      }
    }
  }
  Select          = Select + 1
}
  
```

Figure 38 Controller Library blocks

The "Queues" library combines a named "Queues" and a script block called Smart Controller. This library is used to describe functional models of protocols, flow control algorithms, schedulers and custom hardware components such as arbiters. The Resource is a multi-dimension event queue that enqueue incoming data structures based on priority. The block has a name that is used to track statistics information. The associated Controller determines the scheduling and dequeuing of the data structures. The Controller can be associated with the Resource using the Pop and Stats_in ports. The Controller block describes the algorithm using a C-like scripting language. This language uses the full power of the RegEx; commands such as Goto, While and if-else; and special instructions such as WAIT, TIMEQ and SEND. The Controller can be triggered on the input port or self-triggered. Self-triggered simply means that the block tarts execution at TNow=0.0. The script can get information about the Resource content using the "getBlockStats" command with keywords "stats", "length" and "copy".



The Controller script is similar to the Virtual Machine with some features including the ability to create queues, schedulers, timed queues and virtual connections unavailable.

Server (a.k.a Smart_Timed_Resource)

This is a combination of a queue plus a server resource. This is a multi-dimension resource, which means that multiple queue plus server resource can be defined using this single block. The processing time is known in advance and provided along with the transaction to this block. This can be used to replace a TimedQueue when we need to enhance the block with Power and extract the queue depth information for activities such as flow control and credit policy. It is used to model schedulers, processors and RTOS with scheduling slots arrangements.

Passive or Quantity-Shared Resource Blocks

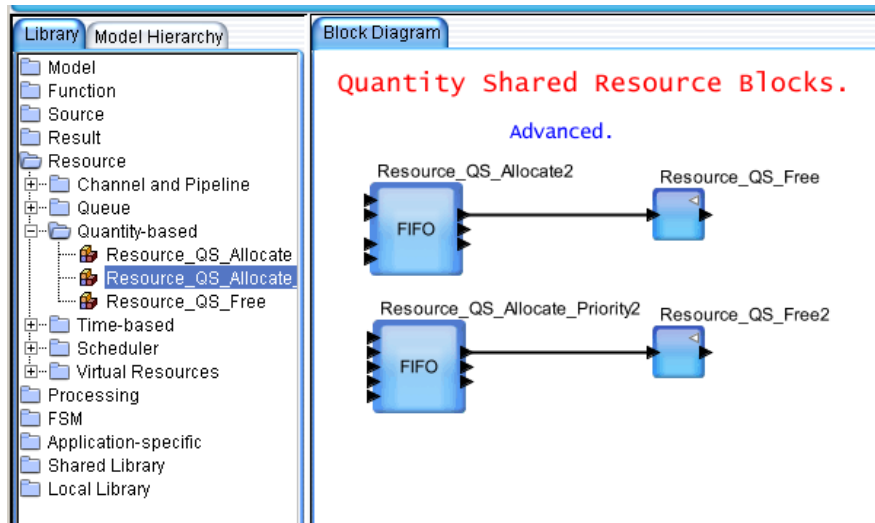


Figure 39 VisualSim Quantity-Shared Resource Library

The Quantity-Shared Resource Blocks, namely **Resource_QS_Allocate**, **Resource_QS_Allocate_Priority**, and **Resource_QS_Free**, represent discrete elements that are stored in a resource pool when not in use and are released as Data Structures when the active Data Structure or resource completes operation. A Quantity-Shared resource completes operation by entering the **Resource_QS_Free** Block, which internally signals the appropriate allocation Resource Block that it has completed its process.

Resource units can represent items such as pages in cache, virtual circuits in a communication channel, or available disk space. The resource units can be indistinguishable, as in a pool, or distinguishable by providing a specific address for the request of units. Requests for addressed resource units can be allocated using a first-fit or best-fit policy, settable as a block parameter. For example, if one requested three cache lines using a first-fit policy, then the first address with three, or more cache lines would be allocated to this request and processed, until the **Resource_QS_Free** Block is executed.

The **Resource_QS_Allocate**, **Resource_QS_Allocate_Priority**, and **Resource_QS_Free** Blocks constitute a different resource modeling capability. Instead of using time as the resource allocation, as in the FCFS blocks, the Quantity Shared (QS) blocks allocate resources based on requests for units from a central resource. Once the requested units are allocated, the model can process the units granted, and when complete, free the resources back to the central resource. Some common applications for the Quantity Shared resource blocks are the allocation of memory, or channels in a networking model pages in cache, virtual circuits in a communication channel, or available disk space. These blocks also can take into account the addresses of the units requested, performing best fit, or first fit algorithm to mimic cache memory, for example.

To better understand the use of these Resource_QS blocks, view the Resource_QS Usage Page [here](#). This shows the usage of and their proposed/intended use models. This also covers what are the required inputs and outputs.

Virtual Connection Blocks

Virtual Connection blocks introduce a new dimension to architectural modeling by simplifying model creation, and introducing a more powerful interconnects methodology that complements spatial topologies. One can create model connections with uniquely named blocks without port to port wires, either local or global, the same concept as local or global memories. It is similar to bubble references in mechanical engineering drawings, except Tokens are actually being sent through the modeling space. To better understand the use of Virtual Connection blocks, view the Virtual Connection Usage Page here.

The basic virtual flow consists of **OUT** to **IN** Blocks. If there are multiple **IN** Blocks with the same name, then a single **OUT** Block is virtually connected to all the like-named **IN** Blocks. A more advanced virtual connection block is the **DEMUX** Block, which can send to a number of destinations, based on the incoming composite Data Structure field (integer or string). If a valid destination is not found with the **DEMUX** Block, then it will drop the composite Data Structure. The **MUX** Block can accept **OUT** or **DEMUX** Block DSs, if the name and type of connection (local, global) match.

Typically, one may wish to access a memory at the output of the **IN** Blocks for plotting, statistics, or model functionality. Similarly, the **IN** Block is used to select a particular field of the incoming Data Structure for plotting, statistics, or model functionality.

The **OUT** can also select a destination based on a memory or Data Structure field of type string. This block is similar to the **DEMUX** Block, except that the destination decoding is one-to-one.

Virtual Connection communication between blocks provides better simulation performance than port to port connections. In addition, **virtual connection** communications provide a dynamic mapping capability within the modeling space, speeding system analysis.

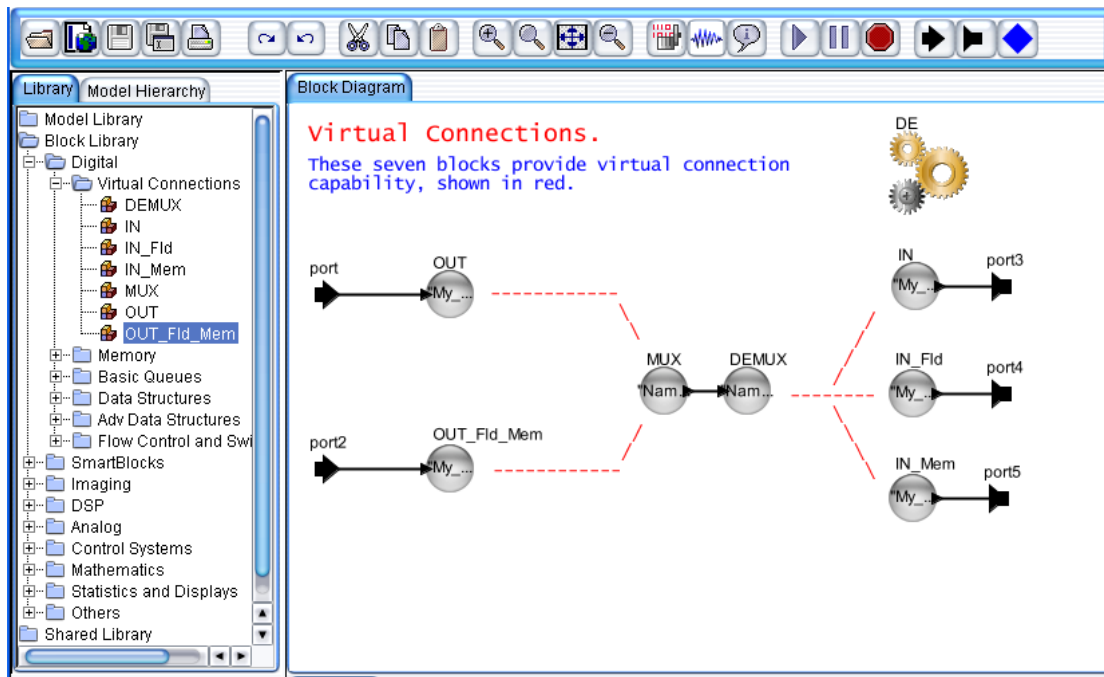


Figure 40 VisualSim Virtual Connection blocks (Virtual Connections Library)



Typically, one would not use the virtual connection blocks as shown in the figure. This is for illustration purposes to demonstrate how they might interact. One can actually go from the **OUT** and **DEMUX** blocks to **IN** and **MUX** blocks.

Architecture Modeling Toolkit

The VisualSim - Hardware Architecture Generator Toolkit generates transaction-level and cycle-accurate models of complex, vendor-specific processor, DSP and application-specific processors. Using this generator and the associated hardware architecture library, platform architecture can be defined graphically without the need to write significant C code or create complex spreadsheets of the instruction sets. These platform models can execute performance-aspects of the software or a specification of the software. The virtual platform can be used to select components, optimize component size and speed, and define arbitration algorithms. This virtual platform can be saved as a library for use by processor architect, systems engineer and software architect within an organization.

The following Figure shows the blocks in the Hardware Architecture Generator Toolkit: Architecture_Setup, Instruction_Set, Processor, Bus_Controller, Bus_Port, Cache, DRAM, and DeviceInterface (a.k.a I_O) blocks.

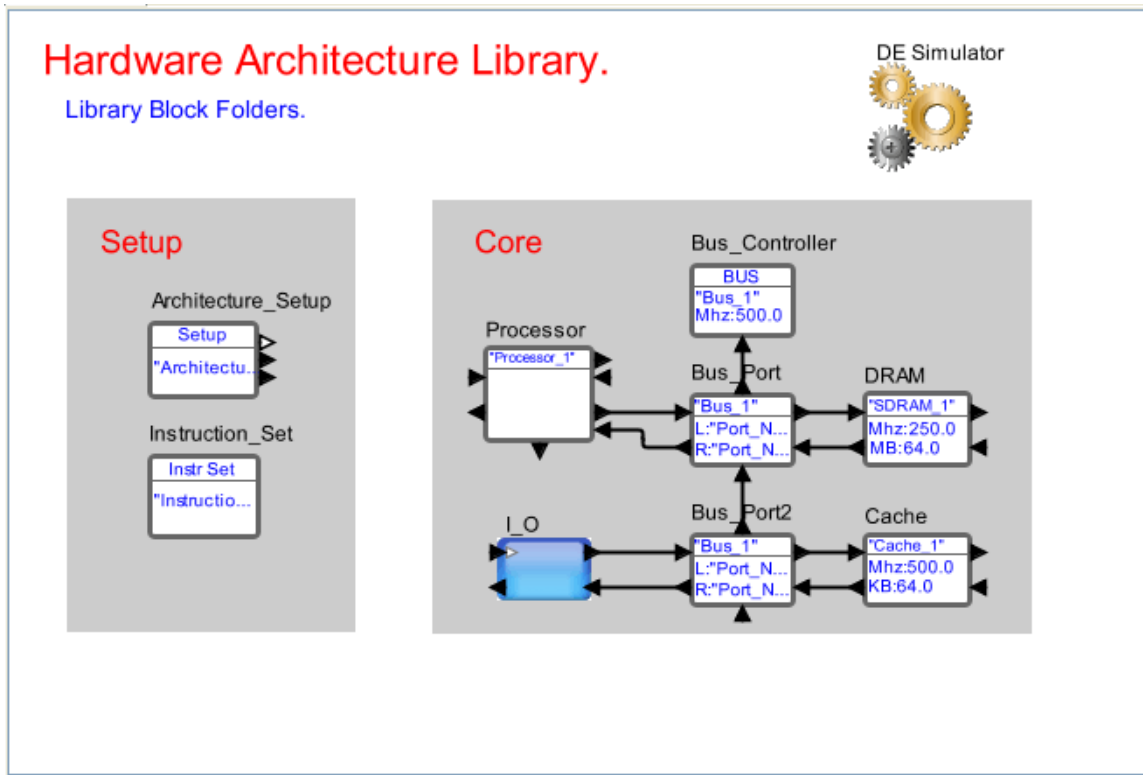


Figure 41 Hardware Architecture Library Blocks

<u>Processor</u>	<u>Cache</u>
<ul style="list-style-type: none"> ○ Pipeline ○ External calls from pipeline for advanced operation creation ○ Registers 	<ul style="list-style-type: none"> ○ Speed ○ Size ○ Hit-ratio ○ Pre-fetch

<ul style="list-style-type: none"> ○ Separate Instruction and Data Cache ○ L2 and L3 ○ Interrupt queue ○ Priority ○ Instruction Set- Integer, Vector, Floating-Point, SIMD, Branch ○ Pre-fetch ○ Stalls ○ Context switch ○ Interrupt Service Routine (internal and external from DMA, etc.) ○ Hit-miss ratio ○ Out of order execution ○ Load Store Unit ○ Branch Instruction ○ Branch prediction ○ Statistics - Throughput, utilization, latency ○ Processor width 	<ul style="list-style-type: none"> ○ Buffer ○ Line width ○ Cache width <p><u>DRAM</u></p> <ul style="list-style-type: none"> ○ Speed ○ Size ○ Buffer ○ Line width ○ Memory width ○ Access Time - Read, Refresh, Write, Erase, Read/Write, ○ Banks ○ Refresh ○ Controllers - SDR, DDR, DDR-2, DDR-3, QDR, RDR, custom
<p><u>Bus Controller/Bus Port</u></p> <ul style="list-style-type: none"> ○ Control message ○ Data message ○ Speed ○ Buffer ○ Word width ○ Switch ○ Arbitration <p><u>DeviceInterface (a.k.a I O)</u></p> <ul style="list-style-type: none"> ○ External In/Out Port ○ Bridge Functionality ○ External Routing 	<p><u>Instruction Set</u></p> <ul style="list-style-type: none"> ○ List of instructions by Execution Units ○ Cycle time by instructions ○ Associate instructions by groups ○ Setup uniform distribution for instruction cycle time <p><u>Architecture Setup</u></p> <ul style="list-style-type: none"> ○ Routing ○ Post processing ○ Statistics aggregation ○ Pipeline activity tracing

Table 4 Hardware Architecture Library Features

This technology radically reduces the time it now takes architects and engineers to create electronic system level (ESL) models. With VisualSim Hardware Architecture Generation Library, a user can create processor, instruction set, cache, memory, and bus models for specific commercial processors and new custom processors. A number of examples are provided in the example section. The VisualSim processor models more valuable, the processor models incorporate the unique ability to separate the architecture and the behavior within the same model. As an example, this allows an ad-hoc change in the number of bits or the number of taps for a fixed-point FIR filter so performance can be evaluated for quality and performance in a single model. This new technology approach enables architects and high-performance computing system designers to validate their architecture selection in less than one week using simulation and accurate application-specific transactions. This becomes even more important when leading edge multi-processors or multi-core architectures are used. A software developer can validate assumptions, thread distributions and best load scheduling techniques in a very short period of

time, even before the code development has commenced. The evaluation can be for performance and power consumption.

Cache

Block Description

This is a hardware cache that determines whether a memory request is a read or writes (instruction), has the requested data (hit), or must go to the next level of the memory hierarchy (miss) to complete the request. The Cache does not make instruction/data distinction.

Block Usage

This is a dual ported Cache. It can be made to depict different cache configurations by manipulating the Cache hit expression. A Cache hit expression is further defined in terms of Cache_Size_Kbytes, Words_per_Cache_Line, A_Instruction (array length) and Unique_Task_ID (assigned by the Processor).

Functionality

The cache processes cache 'Read', 'Write' and 'Prefetch' instructions.

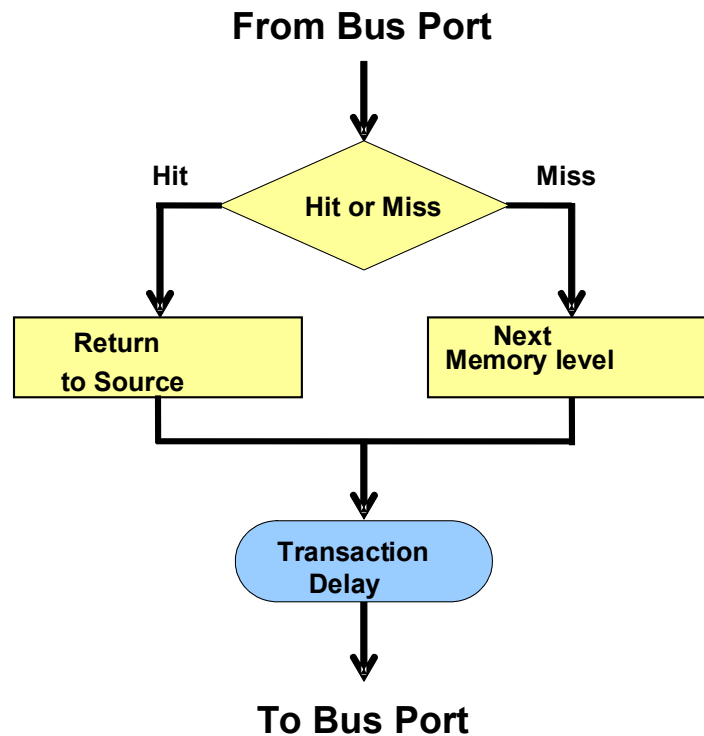


Figure 42 Cache Flow Diagram

The Cache picks up the 'Read' or 'Write' instruction and a cache hit or a miss is determined using a cache-hit expression that depends on several parameters. The cache hit expression is formulated using parameters like cache size, Words_per_cache_line, activity in the cache based on task size and number of tasks etc.

The cache-hit expression thus determines a cache hit or a miss; and if it is a cache hit the token is sent out on the bus port with the destination as the requested source. If a cache miss occurs the next level of memory is looked up. The next level of memory if it is a DRAM, returns a token with the destination as the requested source.

A Cache 'Read' request is processed and the data is returned to the source. If the instruction is Cache 'Write' then it is processed and the token is just dropped after a transaction delay to indicate a cache write.

Cache Pre-fetch

If the number of cache transactions exceeds the words_per_cache_line a cache prefetch is initiated. To simulate a prefetch the cache sends out a token to the next level of memory.

Computations

Hit/Miss Calculation

The Cache Hit expression parameter decides the ratio of hits or misses. This regular expression is evaluated and if true then it resolves to a Cache hit and if false it resolves to a Cache Miss.

Cache latency

Cache delay time = Cache cycle time * number of words

Where number of words = bytes sent / width bytes and cache cycle time = $1.0E-06$ / cache speed in Mhz.

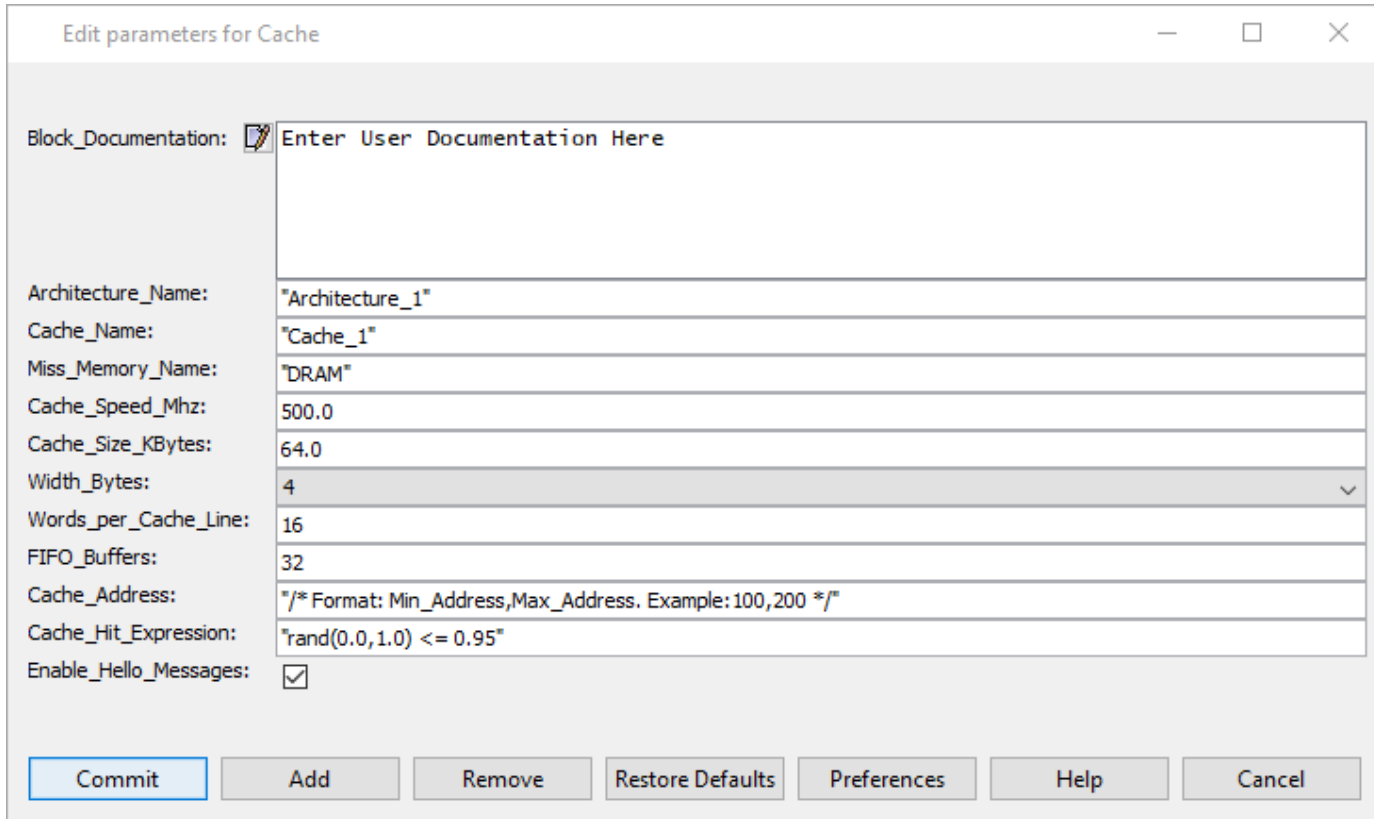
State Plots

The plot shows the cache IDLE and BUSY states (0, 1).

Statistics

Statistics collected are Cache utilization, throughput, transaction delay time, Hit Ratio and Pre-fetch count. Cache utilization depends on sum of all the transaction delays occurring when the cache is processing the request. Cache throughput depends on the sum of bytes sent/requested.

Configuration of Parameters



Block_Documentation:	<input type="checkbox"/> Enter User Documentation Here
Architecture_Name:	"Architecture_1"
Cache_Name:	"Cache_1"
Miss_Memory_Name:	"DRAM"
Cache_Speed_Mhz:	500.0
Cache_Size_KBytes:	64.0
Width_Bytes:	4
Words_per_Cache_Line:	16
FIFO_Buffers:	32
Cache_Address:	"/* Format: Min_Address,Max_Address. Example: 100,200 */"
Cache_Hit_Expression:	"rand(0.0,1.0) <= 0.95"
Enable_Hello_Messages:	<input checked="" type="checkbox"/>

Figure 43 Cache Parameter Window

Architecture_Name

Name of the common architecture to which blocks in the model belong to, Type is String. There can be different architectures existing; hence the unique Architecture name is defined with each block.

Cache_Name

Name of the Cache, Type is String. The Cache name has to be unique within the same architecture. Another architecture can exist with the same Cache name.

Cache_Address

Address of the cache, Type is integer or string.

Cache_Speed_Mhz

Speed of the Cache in Mega Hertz, Type is double. This is the rate at which the Cache operates. The transaction delay times, Cache hit or miss ratio depends on this parameter. Cache speed is used to calculate the Cache cycle time; Cache cycle time = $1.0E-06 / \text{cache speed in Mhz}$. The cache cycle time together with the number of words (chunks of data) determines the cache transaction delay time.

Cache_Size_KBytes

Size of the cache in kilobytes, Type is double. Since this determines the storage capacity of the cache, Cache hit or miss ratio depends on this parameter.

Width_Bytes

The Cache word width in bytes shown as a pull-down with values 2, 4, 8
The number of words (the chunk of data used for processing) depends upon the bytes sent divided by the width bytes.

Words_per_Cache_Line

The total number of words that fit into one cache line, Type is integer.
The cache Pre-fetch is initiated based on the number of cache requests processed and the words per cache line. If the cache request exceeds the words per cache line, then a cache pre-fetch request is initiated.

FIFO_Buffers

The size of the Cache word buffer; i.e. the size of the input queue, Type is integer.

Hit_Expression

An expression formulated using the various parameters that impacts a cache hit, Type is String. It could be as simple as a random number generated based on a condition. Example: "rand (0.0,1.0) < 0.95" or could be advanced hit expression involving several parameters like Cache_Size_Kbytes, Words_per_Cache_Line, A_Instruction (array length), Unique_Task_ID (assigned by the Processor).

Miss_Memory_Name

Name of the next level of memory to access when there is a cache miss, Type is String.

Memory

Block Description

The VisualSim Statistical DRAM executes a memory request, read or write (instruction), and returns the request to the source. In addition, the DRAM model supports memory banks, a new design to improve system performance. DRAM has been selected as the name of the block, as this block can model different dynamic random access memory technologies.

Block Usage

A simple DRAM with a single input and output port can be depicted. It can be made to depict different DRAM technologies and designs including:

- Synchronous Dynamic (SDRAM)
- Double Data Rate (DDR, DDR-2, DDR-3)
- Quad Data Rate (QDR) SRAM
- Direct Rambus (DRDRAM)
- Video DRAM (VRAM)
- Synchronous Graphics RAM (SGRAM)
- Pseudo Static RAM (PSRAM)

Functionality

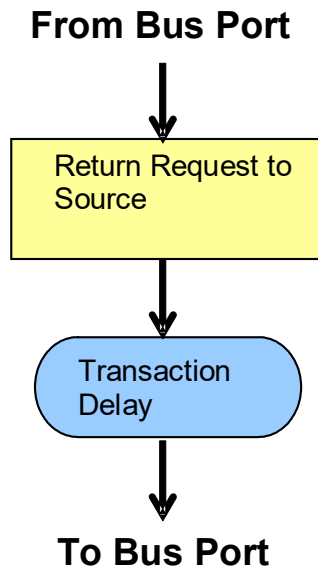


Figure 44 Memory Flow Diagram

All memory requests are queued and processed on First come first served basis. Transactions arriving at the DRAM will either be a "Read" or "Write" request. If the incoming command is a "Read", after a transaction delay computed based on factors like bytes sent, speed of memory, size of memory etc, sends out a token with a destination set to the requested source for ex: the Processor block. If the incoming command is a "Write", after a transaction delay, drops the transaction or data structure. If the incoming command is a "Read_Write", then treats as a "Read" command. A DRAM refresh event is scheduled periodically. When the DRAM is in refresh state it adds the requests on to a queue, this is processed later when it comes back to working state.

Computations

DRAM Access Time

Access Time is the time taken since a request is made and when the data is made available from the DRAM. Access time is defined in nano seconds. DRAM Access Time is defined by the user for each operation like Read, Write or a Read-Write as an Access Time parameter.

DRAM latency

DRAM latency i.e. the total transaction delay time depends on both Access time and the memory cycle time.

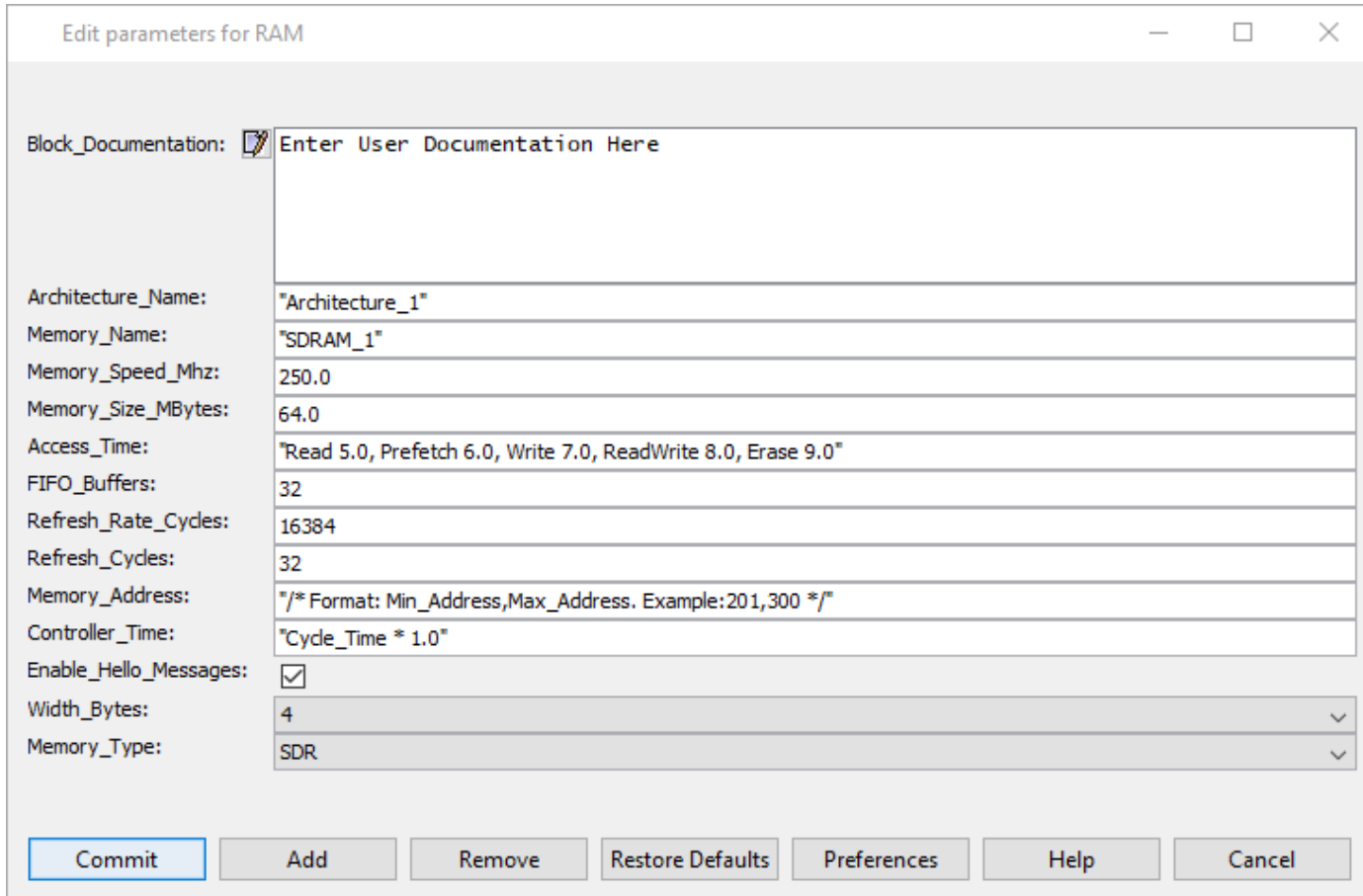
DRAM Access Cycles = Access Time/Memory cycle time

Total Delay Time or Latency = (Number of words + Access cycles - 1) * Memory cycle time,
Where number of words = bytes sent / width bytes and memory cycle time = 1.0E-06 / Memory speed in Mhz.

State Plots

The plot shows the DRAM IDLE and BUSY states (0, 1) when in working state. The plot also shows the DRAM REFRESH states (0, 1) when in refresh state and back to working state.

Configuration of Parameters



Parameter	Value
Block_Documentation:	Enter User Documentation Here
Architecture_Name:	"Architecture_1"
Memory_Name:	"SDRAM_1"
Memory_Speed_Mhz:	250.0
Memory_Size_MBytes:	64.0
Access_Time:	"Read 5.0, Prefetch 6.0, Write 7.0, ReadWrite 8.0, Erase 9.0"
FIFO_Buffers:	32
Refresh_Rate_Cycles:	16384
Refresh_Cycles:	32
Memory_Address:	"/* Format: Min_Address,Max_Address. Example:201,300 */"
Controller_Time:	"Cycle_Time * 1.0"
Enable_Hello_Messages:	<input checked="" type="checkbox"/>
Width_Bytes:	4
Memory_Type:	SDR

Figure 45 Memory Configuration Window

Architecture_Name

Name of the common architecture to which blocks in the model belong to, Type is String. There can be different architectures existing; hence the unique Architecture name is defined with each block.

Memory_Name

Name of the memory, Type is String. The memory name has to be unique within the same architecture. Another architecture can exist with the same memory name.

Memory_Address

Address of the memory, Type is integer or string.

Memory_Speed_Mhz

Speed of the memory in Mega hertz, Type is double. This is the rate at which the memory operates. Memory speed is used to calculate the memory cycle time; Memory cycle time = $1.0E-06 / \text{memory speed in Mhz}$. The memory cycle time together with the access time determines the DRAM transaction delay time.

Memory_Size_MBytes

Size of the memory in megabytes, Type is double. This determines the storage capacity of the memory.

Width_Bytes

The memory word width in bytes shown as a pull-down with values 2, 4, 8

The number of words (the chunk of data used for processing) depends upon the bytes sent divided by the width bytes.

FIFO_Buffers

The size of the Cache word buffer; i.e. the size of the input queue, Type is integer.

Refresh_Rate_Cycles

This is the time between refresh. This value is an average of all the rows in the memory bank.

The refresh rate in memory cycles, Type is integer. This decides the rate at which a DRAM refresh event is triggered during a simulation run.

Refresh_Cycles

The number of memory refresh cycles, Type is integer. This is the number of refresh events that is triggered during a simulation run.

Access_Time

This is the time taken to process requests. The memory access time differs for each type of request like 'Read', 'Write', 'Read_write' or 'Erase'. Any number of commands can be added to the list. The incoming Data Structure field called A_Command determines which Access Time value should be used. The Access Time is the time taken to do the operation in nano-seconds on one memory width within the memory cell. This does not include overhead such as transaction-specific cycles or the controller time. Type is String.

Ex: "Read 90.0, Write 70.0, RdWr 80.0, Erase 60.0"

Controller_Time

This is a RegEx that calculates the latency for the controller. This will be applied to the first word in a transaction request.

Memory_Type

The memory design can be chosen as SDR, DDR, DDR2, DDR3, QDR or RAMBUS.

Type is a pulldown.

Processor Block

Description

The VisualSim processor block approaches the capabilities of an ISS while still retaining the graphical nature of VisualSim to model common processors. The processor block provide parallel execution on different processor cores, or execution on different instruction streams within a single processor core. An Instruction Set Simulators (ISSs) has detailed implementations to the bit level of individual instructions, often called that can process processor specific instructions, sometimes including bus, cache and RAM activity. A typical ISS instruction might include:

Instruction Address

Instruction Mnemonic

Block Usage

A linear state machine engine forms the basis for processor model execution, whether an instruction pipeline, or individual instruction stream. Instructions are processed, based on arrays that execute in the processor model pipeline:

Processor Configuration

Execution Units (Cache, Integer Units, Floating Point Units)

Pipeline Execution

Instruction Cycles (integer or uniform random range of integers)

The length of the arrays can vary based on the processor instruction set, instruction variants, instruction resources, and instruction conditions or constraints. The concept is to keep the processor block as simple as possible for anyone familiar with a processor's instruction set and the internal resources used during execution. One can also group instructions based on cycle count, external memory references, co-processor transactions, etc. for purposes of architectural modeling. The Linear State Machine allows this flexibility in specifying either a complete instruction set, or a "grouped" instruction set.

The VisualSim processor block could be initially constructed with a "grouped" instruction set, and then later refined to a more detailed instruction set, as part of the design process, as required. The VisualSim processor block is essentially a programmable processor in the sense that the configuration and pipeline execution can be altered to fit most common processors, custom ASICs, FPGAs, or other micro-controllers.

The Processor Block is a high performance, multi-instruction-per-cycle, superscalar processor for executing mnemonic sequences of instructions, including key software loops. Or, the Processor Block can represent a simpler ASIC or micro-controller, depending on how the parameters, pipeline is setup. A user can configure the pipeline to match the number of pipeline stages, execute internal or external to the pipeline, vary the cache pipeline pre-fetch width to first level caches, and share the same cache among more than one Processor Block. A user can configure N integer or M floating point execution units, based on instruction set groups setup in the Instruction Set block. Out of order execution can be performed utilizing an Instruction Reorder array that designates if one instruction is dependent on a prior instruction to execute, the default being all instructions are dependent on the prior instruction.

In addition, the Processor Block performs a context switch for each new task, or thread, which begins to execute. The user can specify the number of context switch cycles, during which the Processor Block will perform a pre-fetch for the base cache memories, modeling the pre-fetching of instruction (I1), data (D1) and registers. Each Processor task consists of a composite data structure that contains the mnemonic instruction sequence array, instruction reorder array, and internal routing with external bus, DMA, or memory.

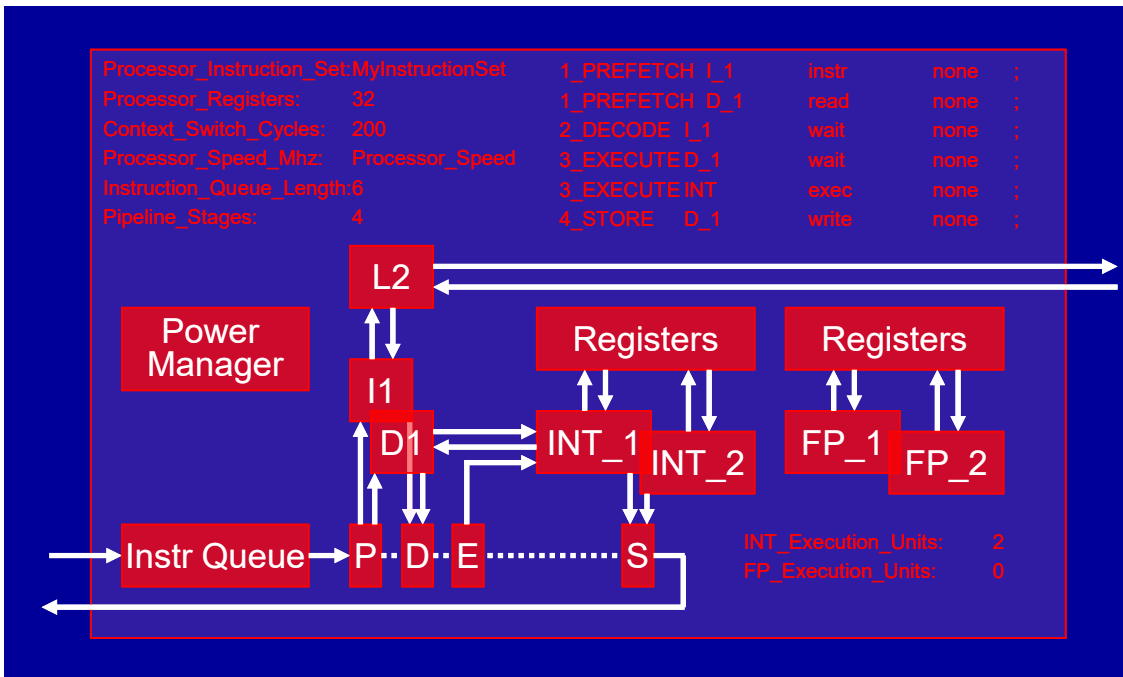


Figure 46 Processor Block

Setup

The Processor Block requires certain Model Parameters and a specific Data Structure (Processor_DS) for proper use, once the Processor Block has been dragged into a model window. Each Processor Block used in a model must designate the Processor Name and Architecture Name associated with the Architecture_Setup Block, which also needs to be dragged into a model, if one does not exist.

Edit parameters for Processor

Block_Documentation: Enter User Documentation Here

Architecture_Name: "Architecture_1"

Processor_Name: "Processor_1"

Processor_Setup:

```

/* First row contains Column Names.
Parameter_Name      Parameter_Value      ;
Processor_Instruction_Set:  MyInstructionSet      ;
Number_of_Registers:      32                      ;
Processor_Speed_Mhz:      1000.0          ;
Context_Switch_Cycles:    100             ;
Instruction_Queue_Length:  6              ;
Number_of_Pipeline_Stages: 4             ;
Number_of_INT_Execution_Units: 3         ;
Number_of_FP_Execution_Units: 2         ;
Number_of_Cache_Execution_Units: 3      ;
I_1:                  {Cache_Speed_Mhz=1000.0, Size_KBytes=16.0, Words_per_Cache_Line=16
D_1:                  {Cache_Speed_Mhz=1000.0, Size_KBytes=16.0, Words_per_Cache_Line=16
L_2:                  {Cache_Speed_Mhz=500.0,  Size_KBytes=64.0, Words_per_Cache_Line=16

```

Pipeline_Stages:

```

/* First row contains Column Names.
Stage_Name  Execution_Location  Action  Condition ;
1_PREFETCH I_1                instr   none      ;
1_PREFETCH D_1                read    none      ;
2_DECODE    I_1                wait    none      ;
3_EXECUTE   D_1                wait    none      ;
3_EXECUTE   INT_1              exec    none      ;
4_STORE     INT_1              wait    none      ;
4_STORE     D_1                write   none      ;

```

Enable_Hello_Messages:

Processor_Bits: 32

Commit Add Remove Restore Defaults Preferences Help Cancel

Figure 47 Processor Block Configuration Parameters

The key Processor parameter settings:

- Architecture_Name: "Architecture_1"
- Processor_Name: "Processor_1"
- Processor_Setup: Condensed List of Processor Parameters
- Pipeline_Stages: Pipeline Script
- Processor_Bits: Pulldown Menu, 16 to 64 Bits

Connecting to Architectural Blocks

The Processor Block has some default ports:

instr_in	– Instruction In Port
instr_out	– Instruction Out, or Complete, Port
bus_in	– Bus In, typically from an architectural bus, Port
bus_out	– Bus Out, typically to an architectural bus, Port
bus_in2	– Bus In 2, typically from an architectural bus, Port
bus_out2	– Bus Out 2, typically to an architectural bus, Port
reject	– Reject Port

The reject port will reject incoming data structures if A_Destination, A_Hop fields do not match the Processor_Name, or if the Instruction Queue is full. The Instruction Queue size can be set by the parameter Instruction_Queue_Length in the Processor_Setup window. This queue is for task, or thread, level data structures. The field A_Instruction defines the mnemonic instruction sequence, where A_IDX, A_IDY define the pointers to the specific instruction, or instructions, if a multi-instruction processor in the execute phase of the pipeline.

The user can also add ports to the Processor Block, such as bus_in3, bus_out3 using the right-mouse click “Configure Ports”, add capability. In this fashion, the Processor can have many more bus ports than the default configuration. The bus_in, bus_out ports can be connected directly to the cache, or DRAM blocks, for example.

Processor Block added to a Model

The Processor Block is shown below; note the Architecture_Setup (Arch_Setup) and Instruction_Set blocks supporting the Processor Block. In this example, the Processor Block is connected to a Linear Bus Port, which in turn is connected to the cache and DRAM blocks, forming a memory hierarchy from the Processor I_1, D_1, L_2 to Cache_1, SDRAM_1 via the Linear Bus. Instructions are entering the instr_in port, and exiting when complete via the instr_out port. A Traffic block is generating the task level data structures for the Processor in this test model. The Architecture_Setup and Instruction_Set blocks are described in more detail in other portions of the documentation, search for Architecture_Setup and Instruction_Set block information.

It is recommended to run a very simple task through the Processor Block to verify that the Architecture_Setup, Instruction_Set, and Processor Block parameters are in sync with each other. The model below runs 500 ADD mnemonic instructions through the test model.

Architecture Library

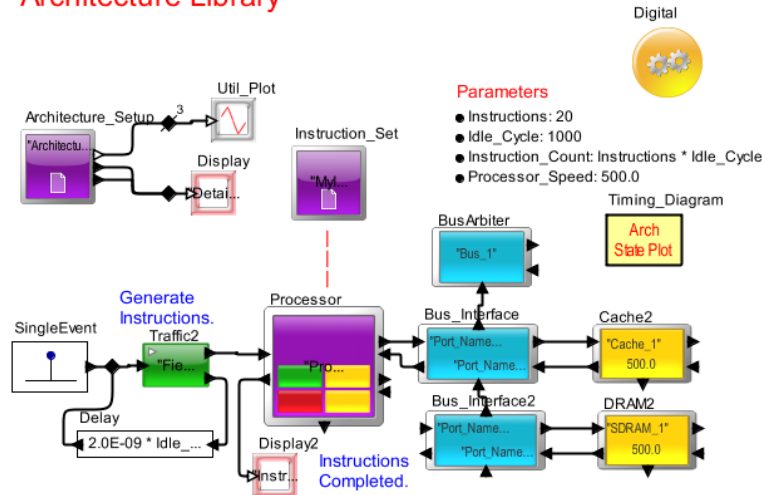


Figure 48 Typical Processor Model

Configuration of Parameters

The Processor_Setup text window includes the following default entries:

```

/* First row contains Column Names.                               */
Parameter_Name           Parameter_Value                       ;
Processor_Instruction_Set: PPC_Instr                          ;
Number_of_Registers:     32                                   ;
Processor_Speed_Mhz:     cpu_speed                            ;
Context_Switch_Cycles:   100                                 ;
Instruction_Queue_Length: 6                                   ;
Instructions_per_Cycle:   2                                   ;
Number_of_Pipeline_Stages: 4                                 ;
Number_of_INT_Execution_Units: 3                             ;
Number_of_FP_Execution_Units: 1                             ;
Memory_Database_Reference: none                               ;
Number_of_Cache_Execution_Units: 2                           ;
I_1: {Processor_Name=Processor_1, Cache_Speed_Mhz=cache_speed,
Size_KBytes=16.0, Words_per_Cache_Access=2, Words_per_Cache_Line=16,
Cache_Miss_Name=SDRAM_1}
D_1: {Processor_Name=Processor_1, Cache_Speed_Mhz=cache_speed,
Size_KBytes=16.0, Words_per_Cache_Access=2, Words_per_Cache_Line=16,
Cache_Miss_Name=SDRAM_1}

```

The Processor supports individual instruction (I_1) and data (D_1) caches like the Harvard architecture, or a single L_1 cache can also be configured. Each processor setup parameter is described below:

Processor_Instruction_Set

This is the name of the Instruction_Set block referenced by this Processor Block, a string.

Processor_Registers

The number of registers in the processor, an integer. The model uses this value to compare against the number of variables (A_Variables, see next) in determining if the data is in a register, or the first level cache.

Context_Switch_Cycles

This is the time that the processor takes to switch to a new thread, or task, load the registers, and issue a cache prefetch. Typically, this will be related to the Processor_Registers, three times is the default.

Processor_Speed_Mhz

The processor speed in Mhz, coupled with the Processor_Bits determines the processor throughput.

Instruction_Queue_Length

Length of the instruction input queue, if each A_Instruction is a string (individual instruction) or a string array (task).

Instructions_Per_Cycle

Number of instructions per cycle in the pipeline execution stage, optional integer parameter. This is an optional parameter.

Pipeline_Stages

Number of stages in the pipeline, should coincide with Pipeline_Stages text window line entries. If the left hand side has the values 1_, 2_, 3_, 4_ (any number of rows), then there are 4 Pipeline_Stages. Processor block uses this information to create internal representation of the pipeline in advance of the pipeline detail, type integer.

INT_Execution_Units

Number of internal integer execution units, can be any value 0 to N, type integer.

FP_Execution_Units

Number of internal floating point execution units, can be any value 0 to M, type integer.

Cache_Execution_Units

Number of internal cache execution units, can be any value 1 to P, type integer. This entry pre-configures the next cache description detail.

Memory_Database_Reference

Memory_Database_Reference for DMA type of instructions, type string. This parameter defaults to "None". This is an optional parameter.

Cache Structure

The Processor Block can setup a variety of cache structures. A common one is instruction (I_1), data (D_1) to on-chip level two (L_2) cache. These three on-chip caches would be described as follows in the Processor_Setup text window:

Cache_Execution_Units 3

```
I_1 {Processor_Name=Processor_Name, Cache_Speed_Mhz=500.0, Size_KBytes=64.0,
     Words_per_Cache_Access=1, Words_per_Cache_Line=16, Cache_Miss_Name=L_2}
D_1 {Processor_Name=Processor_Name, Cache_Speed_Mhz=500.0, Size_KBytes=64.0,
     Words_per_Cache_Access=1, Words_per_Cache_Line=16, Cache_Miss_Name=L_2}
L_2 {Processor_Name=Processor_Name, Cache_Speed_Mhz=500.0, Size_KBytes=64.0,
     Words_per_Cache_Access=1, Words_per_Cache_Line=16, Cache_Miss_Name=Cache_1}
```

The Processor_Name is set to the current Processor. If this field names another Processor Block, then it means the cache used by the current Processor is shared with the named



processor. The `Cache_Speed_Mhz` is typically the speed of the processor, or it could be a multiple slower, such as 2X slower. The `Size_KBytes` sets the internal size of the cache. The `Words_Per_Cache_Access` can be set to N words, depending on the number of instructions executed in one cycle. If two instructions are executed per "exec" of the pipeline, then `Words_per_Cache_Access` would typically be set to two. The `Words_per_Cache_Line` determines how the Processor will perform pre-fetch from the baseline cache to the next level cache. The `Cache_Miss_Name`, is the name of the internal, or external, memory that will be accessed if a single miss occurs, note the `Cache_Miss_Name` in red.

Other possibilities for the cache structure may be to have the next level memory be off-chip, such as L_3 cache (`Cache_1`) or SDRAM. Since each on-chip cache has a miss possibility, it must point to a higher level memory that is either another cache, or SDRAM.

Instruction Stack

The instruction stack is a queue of pending instructions to be processed by the processor model, organized as tasks sent by the RTOS or a behavioral element directly. Each task consists of an array of sequential processor instructions needed for execution on the processor, represented as instruction data structures containing:

- A_Task_Name*** (string for task name, optional)
- A_Task_ID*** (integer for task ID, optional)
- A_Instruction*** (string or array of instructions for task)
- A_Variables*** (number of variables associated with the task)
- A_Source*** (string of source requestor, optional)
- A_Hop*** (string of hop address, same as processor if *instr_in*)
- A_Destination*** (string of destination, processor)
- A_Priority*** (integer of task priority)

The instruction stack can also execute several tasks simultaneously, including accepting higher priority tasks in the instruction stack queue.

Linear State Machine

The Linear State Machine is the basis for the statistical processor block and consists of a variable length pipeline, up to N stages, that can accept task Data Structures as an input, and process that instruction based on the Pipeline_Stages text setup. In the simplest form an instruction is a sequence of events that involves:

- Pre-fetching Instruction Information***
- Decoding Instruction Information***
- Executing Instruction, or parts of Instruction***
- Storing Completed Instruction results***

This is a typical four stage pipeline execution sequence that can be modeled in the VisualSim statistical processor block. Think of the Linear State Machine as a super scheduler that coordinates internal flow of instructions, based on processor speed (Mhz), bus widths (Bytes), etc. The Linear State Machine can also model instruction stream processing, as an instruction specific pipeline, here is the default pipeline:

```

/* First row contains Column Names.          */
Stage_Name Execute_Location Action Condition ;
1_PREFETCH I_1          instr    none    ;
1_PREFETCH D_1          read     none    ;
2_DECODE   I_1          wait     none    ;
3_EXECUTE  D_1          wait     none    ;
3_EXECUTE  INT          exec     none    ;
4_STORE    D_1          write    none    ;

```

The only restriction on Stage_Names, is that they start with 1_, 2_, etc. depicting the pipeline stage, the text after the numbering can be any text that describes the stage. If the above default has 1_Prefetch_I1 and 1_Prefetch_D1 as stage names, the block will process this as the first pipeline stage. The Execution_Location identifies the internal execution unit instruction group, or external execution unit (requires Action = "task"). Valid actions include:

“none” for equivalent of pipeline decode
 “instr” for cache instruction fetch
 “read” for cache data fetch
 “write” for cache write results to register/cache
 “exec” for INT_1, INT_2, FP_1, FP_2 execution on internal execution unit
 “wait” for compliment to cache “read”, or “exec” to integer, floating point unit
 “task” for external execution, Execute_Location (name of external execution unit), while
 Condition contains name of Instruction group, both required.

The shorthand notation of the listen to architecture where an integer appears, is the first integer represents the pipeline stage, and if a second integer appears ((3, 4)), it represents the pipeline execution line number. For the pipeline execution used, here is the stage and line representations:

Stage_Number	Line#	Stage_Name	Execute_Location	Action	Condition ;
1	1	1_PREFETCH	I_1	instr	none ;
1	2	1_PREFETCH	D_1	read	none ;
2	3	2_DECODE	I_1	wait	none ;
3	4	3_EXECUTE	D_1	wait	none ;
3	5	3_EXECUTE	INT	exec	none ;
4	6	4_STORE	D_1	write	none ;

The Processor Block can model 2 to N stage pipelines, simply by increasing the number of stages in the pipeline script; note the number on the left-hand-side of the Stage_Name column. Each stage of the pipeline can perform explicit actions. A classic four stage pipeline that pre-fetches, decodes, executes, and stores results:

Stage_Name	Execute_Location	Action	Condition ;
1_PREFETCH	I_1	instr	none ;
1_PREFETCH	D_1	read	none ;
2_DECODE	I_1	wait	none ;
3_EXECUTE	D_1	wait	none ;
3_EXECUTE	INT	exec	none ;
4_STORE	D_1	write	none ;

A processor with a longer 13 stage pipeline:

Stage_Name	Execution_Location	Action	Condition ;
1_FETCH1	I_1	instr	none ; /* Fetch */
2_FETCH2	D_1	read	none ; /* Fetch */
3_DATA0	I_1	wait	none ; /* Data */
4_DATA1	D_1	wait	none ; /* Data */
5_DATA2	none	exec	none ; /* Data Pipeline */
6_DATA3	none	exec	none ; /* Data Pipeline */
7_DATA4	none	exec	none ; /* Data Pipeline */
8_EXEC0	ARM8	exec	none ; /* Execute Instr */
9_EXEC1	none	exec	none ; /* Processing */
10_EXEC2	none	exec	none ; /* Processing */
11_EXEC3	none	exec	none ; /* Processing */
12_EXEC4	ARM8	wait	none ; /* Wait Execute */
13_EXEC5	D_1	write	none ; /* Store */

One will notice that there are similarities between the two pipelines. In both cases, there is a Harvard cache architecture that separates the instruction (I_1) and data caches (D_1) on the instruction/data pre-fetch and decode stages of the pipeline. One may elect to have a single

cache containing both instructions and data, simply by defining one first level cache. The 13 stage pipeline has some stages that execute (Action = "exec") in the pipeline to retain the proper timing, while the Execution_Location is "none". These stages are not executing instructions, as longer pipelines are using these extra stages for superscalar style setup for executing the instruction, in this example Stage_Name = 8_EXEC_0. Similarly, stages 8 through 12 are executing internal sequences to obtain the result in stage 12_EXEC4, for example. The last stage, 13_EXEC5, stores results like the classic four stage pipeline.

Some processors will execute a variable number of pipeline stages to gain some local instruction advantage, however, the change in the length of the pipeline for certain instructions may introduce non-superscalar effects to the pre-fetch/store portions of the pipeline that may offset certain instruction efficiencies. If the variable instruction execution is based on a strong baseline pipeline execution to maintain superscalar properties, and the variable execution is an extension of the pipeline to the execution unit, then variable pipeline cycles will be accommodated as pipeline waits or stalls, resulting in more efficient overall operation. The Architecture Library assumes a fixed pipeline execution length, as a result.

Processing Flow

Instructions enter the Instruction Stack and proceed through the pipeline. The last instruction in a task triggers the next task for execution, emptying the Instruction Stack of the task that has just completed.

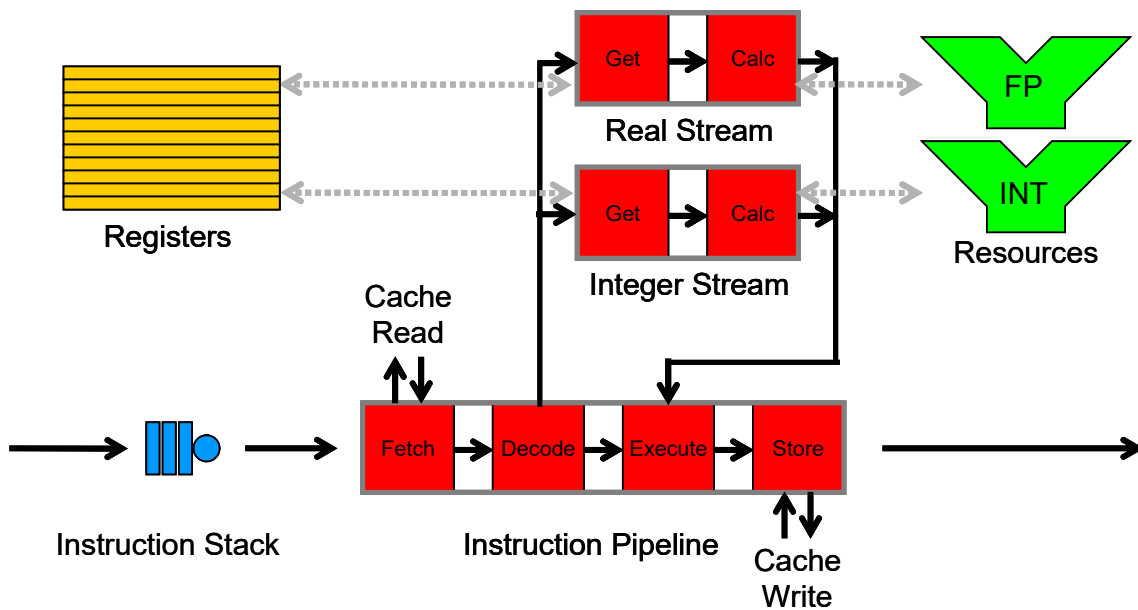


Figure 49 Statistical Processor Block with Pipeline

Figure 1 illustrates a typical flow in the statistical processor block using an advanced pipeline for a four stage pipeline and two instruction streams: integer and real. The integer and real instruction streams resemble an instruction pipeline at a secondary level with essentially the same capabilities. One could have sub-instruction streams for special instructions, such as AltiVec SIMD in the PowerPC, if needed. Registers contain pseudo data, and resources (execution units) can perform operations based on the cycles defined in the Instruction_Set block.

Execution of the Linear State Machine or processor pipeline can take into account a finite set of operations:

- No Action (Clock Delay)**
- Non-Blocking Call**
- Blocking Call**

Each pipeline stage executes with available resources, else an “idle” cycle will be inserted for a pipeline stage if the resource is busy. If a pipeline state is waiting for a resource to complete, and the execution unit has not completed, then the pipeline will “stall” waiting for the result. These pipeline execution mechanisms indirectly resolve availability and buffering restrictions. The processor statistics provide buffering information for execution units based on the tasks/instructions executed on the processor.

More on Processor Flow

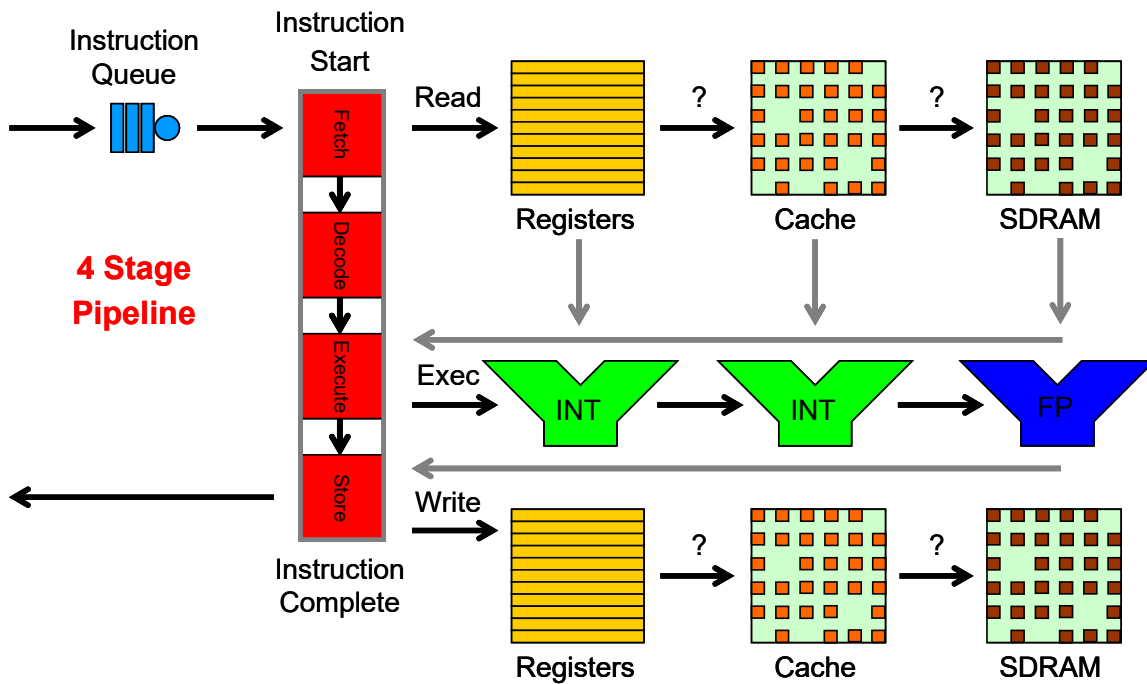


Figure 50 Statistical Processor Block Instruction flow

The statistical processor flow in the model starts with a new task starting to execute in the pipeline. Before the new task starts to execute instructions, the pipeline executes the Context_Switch_Cycles set for the processor, while at the same time prefetching the first cache lines for the lowest level caches, such as I_1 (instruction cache) and D_1 (data cache). In the second stage of the pipeline, the Processor waits for the I_1 instruction and register/D_1 data, and then decodes the instruction. In the third stage of the pipeline, the instruction is executed either internally (“exec”) or externally (“task”). The last stage of the pipeline waits (“wait”) for the finished instruction execution, and stores the result in either registers or cache, depending on the number of variables the task contains.

The processor state plot diagram shows this at the beginning of the simulation:

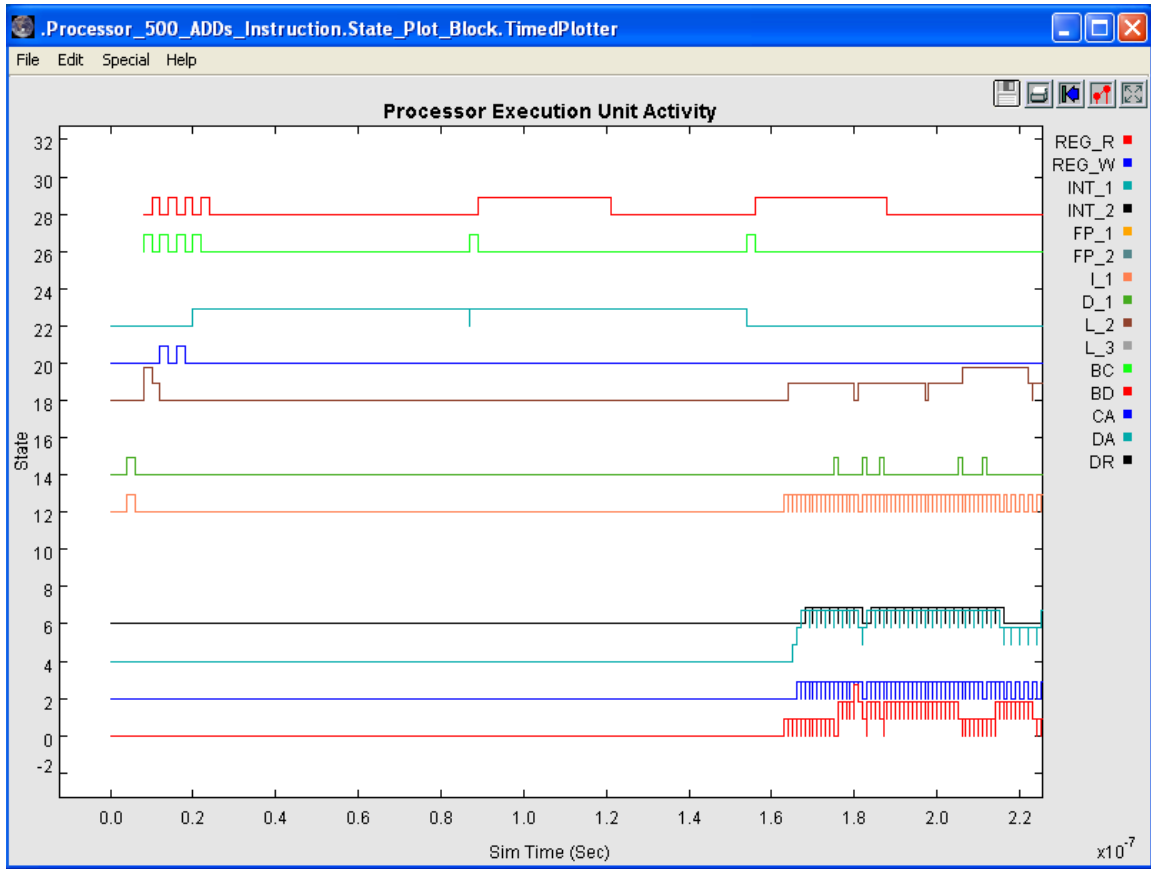


Figure 51 Statistical Processor Block Startup Cache Prefetch

The light blue trace (DRAM Access: DA) in this figure show the initial cache pre-fetch starting at the beginning of the context switch cycles for the processor for the I_1 (orange trace) and D_1 (dark green trace) first level caches. Both the I_1 and D_1 caches then go to the L_2 (brown trace), which in turn goes to the external blue trace (Cache Access:CA). The external cache then goes to the DRAM for the first cache pre-fetch for I_1 and D_1. The top light green trace (Bus Controller: BC) and red trace (Bus Data: BD) show the external bus transactions.

The first cache pre-fetches arrive about the same time as the Processor begins to execute the instructions, 500 ADD instructions in this example model. Internally, the I_1, D_1, and L_2 caches maintain dynamic pre-fetch counts of lines fetched, as compared to instruction task lines executed, as the key determinant of cache hits or misses. If the cache cannot pre-fetch a cache line in sufficient time for the pipeline instruction execution, based on the pre-fetch words versus instructions executed, then the instruction will perform a miss sequence. In addition, on a cache line crossing, assuming the pre-fetched instruction, data is available, the cache algorithm will examine if the current tasks running in the cache have sufficient space, such that they have not been swapped with other executing tasks.

Instruction_Set

Description

The Instruction_Set block can be used to create instruction references for the Processor using a shorthand notation with indirect, user-defined naming references. Here is a PowerPC example of how the Instruction_Set block can be configured:

```

/* Instruction Set or File Path. */
Mnew Ra Rb Rc Rd Re ; /* Label */
PPC IU BPU FPU VPU ;
IU INT_1 INT_2 ;
BPU INT_3 ;
FPU FP_1 ;
VPU FP_2 ;

begin INT_1 ; /* Group */
IU_add 1 ;
IU_shift 1 ;
IU_rotate 1 ;
IU_logical 1 ;
end INT_1 ;

begin INT_2 ; /* Group */
IU_mux 6 ;
IU_div 19 ;
end INT_2 ;

begin INT_3 ; /* Group */
*b 1 ;
l_s 1 ;
end INT_3 ;

begin FP_1 ; /* Group */
FPU_s_add 3 ;
FPU_s_mul 3 ;
FPU_s_madd 3 ;
FPU_s_div 17 ;
FPU_d_add 3 ;
FPU_d_mul 3 ;
FPU_d_madd 3 ;
FPU_d_div 31 ;
end FP_1 ;

begin FP_2 ; /* Group */
vpu_sim_int 1 ;
vpu_com_int 3 ;
vpu_fp 4 ;
end FP_2 ;

```

Notice how the very first line after the column labels, which starts with “PPC” references internal execution units that appear below in two stages:

```

Mnew Ra Rb Rc Rd Re ; /* Label */
PPC IU BPU FPU VPU ;
IU INT_1 INT_2 ;
BPU INT_3 ;
FPU FP_1 ;

```

```
VPU FP_2 ;
```

Since BPU, FPU, and VPU have direct execution unit references, they could have named them directly. In this case, the first line could read “PPC IU INT_3 FP_1 FP_2”.

The IU (integer units) in the PPC line refers to INT_1 and INT_2 execution units, which are in turn defined below this line as:

```
begin INT_1 ; /* Group */
  IU_add 1 ;
  IU_shift 1 ;
  IU_rotate 1 ;
  IU_logical 1 ;
end INT_1 ;

begin INT_2 ; /* Group */
  IU_mux 6 ;
  IU_div 19 ;
end INT_2 ;
```

The IU line defines two execution units that are available, based on the instructions they execute. If the instructions are unique between INT_1 and INT_2, then the instruction being processed by the pipeline will select the execution unit, based on the instruction uniqueness. If the instructions are shared between INT_1 and INT_2, then the available processor will be selected, in essence out of order execution based on who is currently available (not true in this example).

If the instructions are unique for all execution units, typically true for smaller processors, then the first line can just list the execution units directly. As the above execution unit instruction lists indicate, one can start with:

```
begin INT_1 ; /* Group */
```

and finish with:

```
end INT_1 ;
```

These keywords append INT_1 to each instruction in the group, so they actually appear as “INT_1 IU_add 1” internal to the Instruction_Set block. One can use the Instruction_Set block to create arrays that could be referenced by other blocks in the model (Instruction_Set name is the memory reference, set to global), for example.

Each instruction can have a single integer to represent the execution time (IU_add and IU_logical), or two values that allow the model to statistically select from a uniform random distribution between the two values (IU_shift and IU_rotate):

```
IU_add 1 ;
IU_shift 1 8 ;
IU_rotate 1 8 ;
IU_logical 1 ;
```

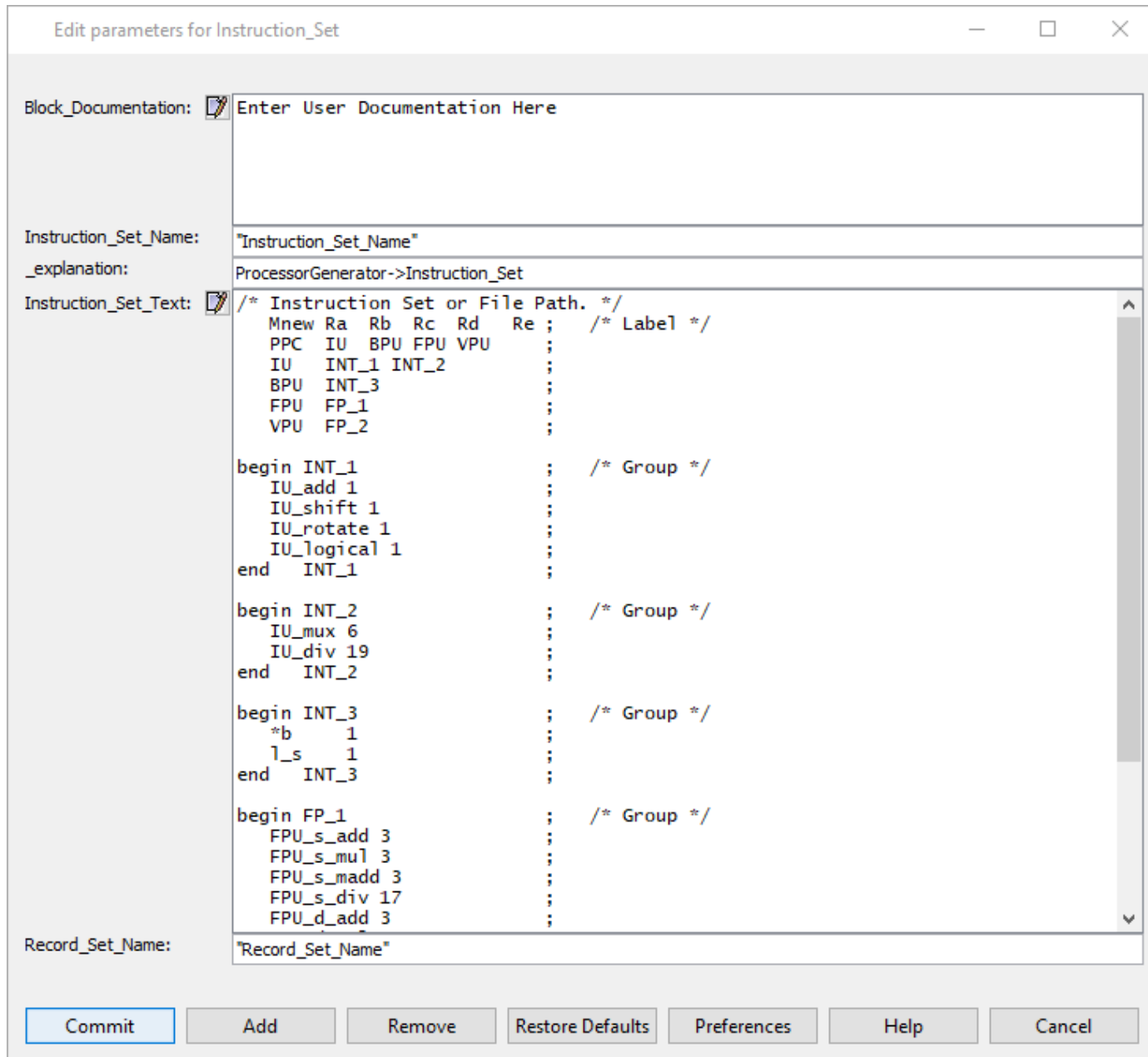


Figure 52 Instruction_Set Parameter Settings

Instruction_Set_Name

This must match the Processor parameter Processor_Instruction_Set, or the model will throw an exception indicating that it cannot find the instruction set. The memory type can be local or global, depending on the individual processors actually accessing the same instructions, whether in a single model window, or if processor blocks are inside hierarchical blocks accessing the same instruction set. Local is the default.

Instruction_Set_Text

This is the listing for the instruction units, groups already discussed on prior pages.

Architecture Setup

Description

The architecture setup block configures the complete set of blocks linked to a single Architecture_Name parameter found in most blocks. The Architecture_Setup has a Field_Name_Mapping text window, Routing_Table text window, Number_of_Samples, Statistics_to_Plot, Internal_Plot_Trace_Offset, and Listen_to_Architecture entries.

Edit parameters for ArchitectureSetup
— □ ×

Block_Documentation:	<input checked="" type="checkbox"/> <pre>Source_Node Destination_Node Hop Source_Port ; Processor_1 Cache_1 Port_1 bus_out2 ; Cache_1 Processor_1 Port_2 output ; Cache_1 SDRAM_1 Port_2 output ; SDRAM_1 Cache_1 Port_4 output ; SDRAM_1 Processor_1 Port_4 output ;</pre>
Architecture_Name:	"Architecture_1"
Routing_Table:	<input checked="" type="checkbox"/> <pre>/* First row contains Column Names. */</pre>
Number_of_Samples:	2
Statistics_to_Plot:	"Processor_1_PROC_Utilization_Min, Processor_1_PROC_Utilization_Mean, Processor_1_PROC_Utilization_Max"
Internal_Plot_Trace_Offset:	2
Listen_to_Architecture_Options:	None ▾
Field_Name_Mapping:	<input checked="" type="checkbox"/> <pre>/* First row contains Column Names. */ External_Field_Name Internal_Field_Name ; A_Address A_Address ; A_Bytes A_Bytes ; A_Data A_Data ; A_IDX A_IDX ; A_Instruction A_Instruction ; A_Priority A_Priority ; A_Source A_Source ; A_Destination A_Destination ; A_Task_ID A_Task_ID ; A_Time A_Time ;</pre>

Architecture_Setup Parameter Settings

Configuration of Parameters

Field Mapping

```

/* First row contains Column Names. */
External_Field_Name      Internal_Field_Name      ;

ID                       A_Address                ;
A_Bytes                  A_Bytes                  ;
A_Data                   A_Data                   ;
A_IDX                    A_IDX                    ;
A_Instruction            A_Instruction            ;
A_Priority               A_Priority               ;
A_Source                 A_Source                 ;
A_Destination            A_Destination            ;
A_Task_ID                A_Task_ID                ;
A_Time                   A_Time                   ;

```

This Field_Name_Mapping allows one to map and existing Data Structure fields to architecture specific fields for common variables, such as routing, address, instructions, priority, etc. There is also a standard Processor_DS that one can use with these fields pre-defined. The Processor_DS:

```

/*
    Processor Data Structure

Field Name                Type      Value      Comment                                     */
A_Address                 int       100        ; /* Starting Instruction, Bus Addr */
A_Branch                  boolean   false      ; /* Instruction Branch */
A_Bytes                   int       8          ; /* Bus Bytes */
A_Bytes_Remaining         int       4          ; /* Bus Bytes Remaining */
A_Bytes_Sent              int       4          ; /* Bus Bytes Sent */
A_Command                 String    Read       ; /* Routing Command */
A_Data                    String    MyData    ; /* User Data */
A_Instruction              String    ADD       ; /* Instr (String, or ArrayToken) */
A_Interrupt               boolean   false     ; /* Instruction Interrupt */
A_Prefetch                boolean   false     ; /* Instruction Prefetch Flag */
A_IDX                     int       0          ; /* Instruction Index */
A_Priority                int       0          ; /* Instruction, Bus Priority */
A_Proc_Return             int       -1         ; /* Processor Return ID */
A_Return                  int       -1         ; /* Return ID */
A_Task_Name               String    Name     ; /* Unique Task Name */
A_Task_ID                 long      1          ; /* Unique Task ID */
A_Source                  String    Src     ; /* Routing Source */
A_Hop                     String    Hop     ; /* Routing Hop */
A_Status                  String    Status  ; /* Routing Status */
A_Destination             String    Dest    ; /* Routing Destination */
A_Time                    double    0.0       ; /* Internal Timestamp */
A_Variables               int       16         ; /* Number of Software Variables */

```

Usage of Data Structure fields in the Model

A_Address
Holds the bus address, maintained internally and not shown to the user

A_Branch
Instruction Branch, used by the processor

A_Bytes
Total Bus Bytes of a transaction

A_Bytes_Remaining
Bus Bytes Remaining after a transaction through the bus is made



A_Bytes_Sent
Bus Bytes sent in a transaction through the bus

A_Command
Holds the Request made by the Master, destined for a slave device

A_Data
User Data

A_Instruction
Processor Instruction

A_Interrupt
Processor Instruction Interrupt either true or false.

A_Prefetch
Processor Instruction Prefetch Flag either true or false

A_IDX
Processor Instruction Index

A_Priority
Bus Priority of the Processor Instruction

A_Proc_Return
Processor Return ID

A_Return
Return ID

A_Task_Name
Unique Processor Task Name

A_Task_ID
Unique Processor Task ID

A_Source
Source that generates a request

A_Hop
Hop port name for routing a request

A_Status
Routing Status manipulated internally the Bus Controller

A_Destination
Routing Destination, destination where the request has to be sent

A_Time
Internal Timestamp

A_Variables
Number of Software Variables

Data Structure Example

```
{A_Address      = 16,  
A_Branch       = false,  
A_Bytes        = 8,  
A_Bytes_Remaining = 4,  
A_Bytes_Sent   = 4,  
A_Command      = "write",  
A_Data         = "MyData",  
A_Destination  = "Processor_1",  
A_Hop          = "Processor_1",  
A_IDX          = 0,  
A_Instruction  = "ADD",  
A_Interrupt    = false,
```

```

A_Pipeline           = {0, 0, 0, 0, 0, 0},
A_Prefetch           = false,
A_Priority           = 0,
A_Proc_Return        = -1,
A_Return             = -1,
A_Source             = "RTOS",
A_Status             = "Status",
A_Task_ID            = 1L,
A_Task_Name          = "Name",
A_Time               = 0.0,
A_Variables          = 16,
BLOCK                = "DS_Gen",
DELTA                = 1.464E-6,
DS_NAME              = "Processor_DS",
ID                   = 16,
INDEX                = 732,
TIME                 = 1.5E-6}

```

Routing_Table

This can route instructions within the processor, bus, cache, SDRAM topology modeled. One advantage of this simplified routing table is that the external bus creates its own internal routing map, so bus input port to output bus port, for a connected cache, or SDRAM does not need to be added to the routing table, thereby saving many manual entries. Here is a typical routing table:

```

/* First row contains Column Names.                */
Source_Node   Destination_Node   Hop           Source_Port ;
Processor_1   Cache_1           Port_1       bus_out     ;
Processor_1   SDRAM_1          Port_1       bus_out     ;
Cache_1       Processor_1      Port_2       output      ;
Cache_1       SDRAM_1          Port_2       output      ;
SDRAM_1       Processor_1      Port_4       output      ;
SDRAM_1       Cache_1          Port_4       output      ;

```

The organization of the routing table is source, destination to obtain the next hop in the routing table, and there is also a Source_Port for the port name to exit if there is more than one output port on the block, such as the Processor, Cache, or DRAM. One will notice that the default routing table does not have any bus ports, Port_1, Port_2, Port_3, or Port_4 as sources or destinations, since the bus itself creates an internal bus specific routing table. Thus, one just needs to add entries for Processor, Cache, DRAM, I_O blocks, meaning to/from destination nodes in the topology.

The use of the routing table is straight forward, in the sense that any entry missing during processor execution will throw an exception saying this source, destination pair is "missing" in the routing table, one then needs to add source destination, and the next hop in the routing. The Hop column are typically bus ports, such as going from the Processor to Cache_1, using Port_1 of the bus, and using the bus_out port of the Processor. The Hop column is used in the model to check that the next destination is correct, meaning each block A_Hop field must match the node name, else the model will throw an exception. This is to confirm the desired routing, and if a change has been made to make simple checks at each point in the overall topology. To determine the next Hop in the routing table simply look at which port one wishes to use, and the source port to leave the Processor, for example.

This routing table can be expanded to include multiple busses in the same topology by just adding more rows to the routing table, based on advance knowledge, or routing exceptions, when the model is running. The Source_Port column becomes more important in multiple bus topologies, as one can connect a dual port cache, dual port SDRAM to two different processors, by having the source (Cache for example), destination (Processor_1 or Processor_2 – two entries

in routing table), and Hop with different port names, and the Source_Port will be output or output2 on the Cache, depending on the destination processor.

Here is an example of a dual processor, dual-port cache, dual-port SDRAM routing table:

```

/* First row contains Column Names.
Source_Node Destination_Node Hop Source_Port ;
ARM7_1 Cache_L2 Port_1 bus_out ;
ARM7_2 Cache_L2 Port_8 bus_out ;
Cache_L2 ARM7_1 Port_2 output ;
Cache_L2 ARM7_2 Port_5 output2 ;
Cache_L2 SDRAM_1 Port_2 output ;
SDRAM_1 Cache_L2 Port_4 output ;
SDRAM_1 ARM7_1 Port_4 output ;
SDRAM_1 ARM7_2 Port_7 output2 ;

```

Here is what the above topology looks like, and shows how a relatively complex topology requires only a few routing table entries. Each to/from node in the topology requires two entries in the table, in the model below the to/from nodes are ARM7_1, ARM7_2, Cache_L2, and SDRAM_1. One will also observe that there are no entries for the eight bus ports as source or destinations, since they contain their own mini-routing tables calculated when "hello" messages are sent during model initialization. The bus ports do appear as Hop column entries, used for dynamic routing.

Dual ARM7 + L2 Cache + SDRAM

Detailed processor portion of model.

Parameters.

- Processor_Speed_Mhz: Processor_Speed_Mhz
- L_Cache_KB: "16"
- D_Cache_KB: "8"
- Bus_Speed_Mhz: Processor_Speed_Mhz
- Cache_Speed_Mhz: Processor_Speed_Mhz
- Cache_Size_KB: 32
- RAM_Speed_Mhz: Processor_Speed_Mhz / 2.0
- RAM_Size_MB: 8
- RAM_Access_Time: "Read 8.0, Prefetch 8.0, Refresh 8.0, Write 7.5"

Digital Simulator

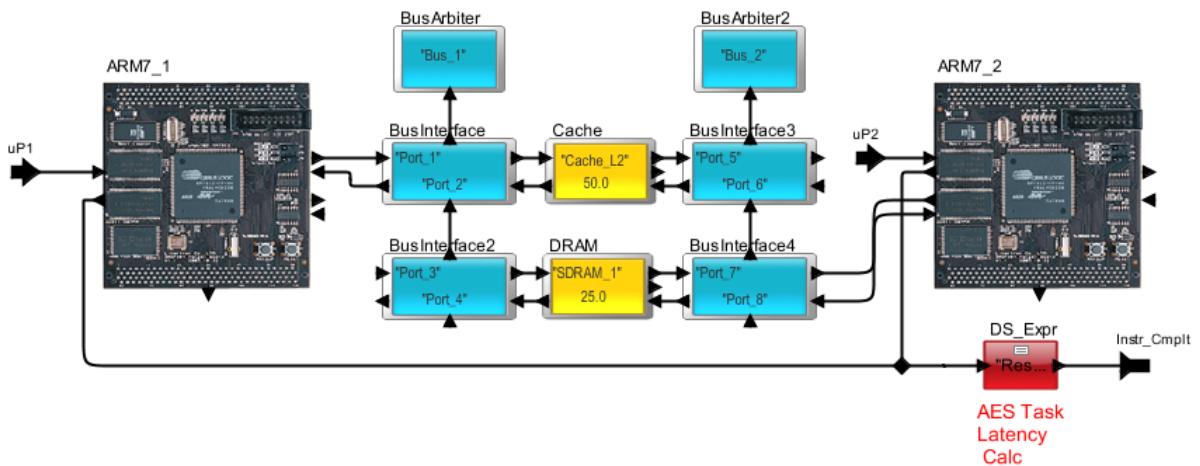


Figure 53 Dual Processor, Dual-Port Cache, Dual-Port SDRAM Routing

Purpose of Hello Messages

Every block within the hardware architecture library sends out Hello messages at simulation time 0.0 to determine the node-to-node connectivity. Each bus in the topology creates an internal routing table, based on the hello messages received, meaning the bus knows each end node it is connected to, and the user is freed from having to construct each bus routing table.

Hello Messages received by each to/from node are added to the routing table with the source, destination, port information. User entries to the routing table supplement the Hello message entries to simplify routing table construction.

Custom Routing with DeviceInterface (a.k.a I_O) Block

One can create custom routing within a model, simply by using the DeviceInterface block as a named port connected to the bus. The output of the DeviceInterface block passes transactions outside of the internal routing table of one portion of the hardware architecture and can reenter another portion of the hardware architecture with a second DeviceInterface block. The user needs to enter the DeviceInterface block names manually into the routing table, as if they are endpoint to/from nodes, while in reality they become gateway nodes. In the example below, the Cache_Miss_Name can now refer to DeviceInterface block, which could route to the SDRAM_1 shown, or other memories, assuming user routing between DeviceInterface blocks. Here is an example of custom routing with bridge functionality:

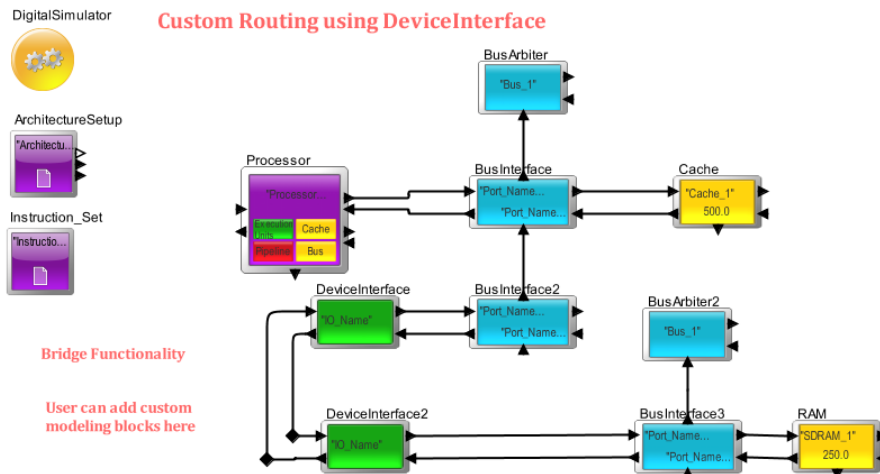


Figure 54 Custom Routing with I_O Block

Number_of_Samples

This field determines how many statistics samples will be collected for the simulation run. If set to 10, then the simulation time will be segmented into 10 equal times for obtaining statistics for the Processor, caches, execution units, bus, cache, or SDRAM blocks.

Util_stats_out

Here are typical statistical outputs for utilizations only (util_stats_out port on Architecture_Setup block):

```
{BLOCK = ".Processor_500_ADDs_Instruction.Arch_Setup",
Bus_1_Utilization_Pct_Max = 7.304347826087,
Bus_1_Utilization_Pct_Mean = 5.2869565217391,
Bus_1_Utilization_Pct_Min = 3.6521739130435,
Bus_1_Utilization_Pct_StDev = 1.3391304347826,
Cache_1_Utilization_Pct_Max = 1.5652173913043,
```

```

Cache_1_Utilization_Pct_Mean      = 0.5565217391304,
Cache_1_Utilization_Pct_Min      = 0.1739130434783,
Cache_1_Utilization_Pct_StDev    = 0.5132645065521,
DELTA                            = 0.0,
DS_NAME                          = "Architecture_Stats",
ID                                = 10,
INDEX                             = 0,
Processor_1_D_1_Utilization_Pct_Max = 4.9565217391304,
Processor_1_D_1_Utilization_Pct_Mean = 3.622712086394,
Processor_1_D_1_Utilization_Pct_Min = 2.5217391304348,
Processor_1_D_1_Utilization_Pct_StDev = 0.6703205236235,
Processor_1_INT_1_Utilization_Pct_Max = 30.0,
Processor_1_INT_1_Utilization_Pct_Mean = 27.0821422758912,
Processor_1_INT_1_Utilization_Pct_Min = 18.2009838369642,
Processor_1_INT_1_Utilization_Pct_StDev = 3.4499003893845,
Processor_1_INT_2_Utilization_Pct_Max = 27.4782608695652,
Processor_1_INT_2_Utilization_Pct_Mean = 24.3595441693535,
Processor_1_INT_2_Utilization_Pct_Min = 16.9360505973296,
Processor_1_INT_2_Utilization_Pct_StDev = 2.9988177798574,
Processor_1_I_1_Utilization_Pct_Max = 57.304347826087,
Processor_1_I_1_Utilization_Pct_Mean = 51.5928249820748,
Processor_1_I_1_Utilization_Pct_Min = 35.2775825720309,
Processor_1_I_1_Utilization_Pct_StDev = 6.3599231594066,
Processor_1_L_2_Utilization_Pct_Max = 4.0,
Processor_1_L_2_Utilization_Pct_Mean = 3.5771415496232,
Processor_1_L_2_Utilization_Pct_Min = 2.6001405481377,
Processor_1_L_2_Utilization_Pct_StDev = 0.3884179251631,
Processor_1_PROC_Utilization_Pct_Max = 57.2173913043478,
Processor_1_PROC_Utilization_Pct_Mean = 51.4825160304522,
Processor_1_PROC_Utilization_Pct_Min = 35.1370344342937,
Processor_1_PROC_Utilization_Pct_StDev = 6.3833018229608,
Processor_1_Register_Rd_Utilization_Pct_Max = 57.3913043478261,
Processor_1_Register_Rd_Utilization_Pct_Mean = 51.7965463970403,
Processor_1_Register_Rd_Utilization_Pct_Min = 34.9964862965566,
Processor_1_Register_Rd_Utilization_Pct_StDev = 6.4092284138104,
Processor_1_Register_Wr_Utilization_Pct_Max = 55.4395126196693,
Processor_1_Register_Wr_Utilization_Pct_Mean = 49.686043916307,
Processor_1_Register_Wr_Utilization_Pct_Min = 34.0126493323963,
Processor_1_Register_Wr_Utilization_Pct_StDev = 6.1062749523381,
SDRAM_1_Utilization_Pct_Max      = 11.5942028985507,
SDRAM_1_Utilization_Pct_Mean     = 7.6811594202899,
SDRAM_1_Utilization_Pct_Min     = 5.7971014492754,
SDRAM_1_Utilization_Pct_StDev   = 2.2264190573532,
TIME                             = 2.3E-5}

```

One will notice min, mean, stdev, max values for all architecture resources.

Internal_stats_out

Here are typical statistical output for throughput, latencies, and buffering within the architecture resources (internal_stats_out port on Architecture_Setup block):

```

{BLOCK                            = ".Processor_500_ADDs_Instruction.Arch_Setup",
Bus_1_Delay_Max                   = 3.8E-8,
Bus_1_Delay_Mean                   = 1.3212765957447E-8,
Bus_1_Delay_Min                   = 3.9999999999986E-9,
Bus_1_Delay_StDev                 = 1.3652894900276E-8,
Bus_1_Throughput_MBs_Max          = 130.4347826086957,
Bus_1_Throughput_MBs_Mean         = 89.3913043478261,
Bus_1_Throughput_MBs_Min          = 62.6086956521739,
Bus_1_Throughput_MBs_StDev        = 27.8825514070694,
Cache_1_Delay_Time_Max            = 3.2E-8,
Cache_1_Delay_Time_Mean           = 3.7647058823529E-9,
Cache_1_Delay_Time_Min            = 2.0E-9,
Cache_1_Delay_Time_StDev          = 7.0588235294118E-9,
Cache_1_Hit_Ratio_Max             = 100.0,
Cache_1_Hit_Ratio_Mean            = 95.5,
Cache_1_Hit_Ratio_Min             = 75.0,

```

```

Cache_1_Hit_Ratio_StDev      = 9.0691785736085,
Cache_1_Prefetch_Count_Max  = 1.0,
Cache_1_Prefetch_Count_Mean = 0.2,
Cache_1_Prefetch_Count_Min  = 0.0,
Cache_1_Prefetch_Count_StDev = 0.4,
Cache_1_Throughput_MBs_Max  = 31.304347826087,
Cache_1_Throughput_MBs_Mean = 11.1304347826087,
Cache_1_Throughput_MBs_Min  = 3.4782608695652,
Cache_1_Throughput_MBs_StDev = 10.2652901310426,
DELTA                        = 0.0,
DS_NAME                      = "Architecture_Stats",
ID                            = 10,
INDEX                        = 0,
Processor_1_Context_Switch_Time_Pct_Max = 17.5652173913043,
Processor_1_Context_Switch_Time_Pct_Mean = 10.528036160504,
Processor_1_Context_Switch_Time_Pct_Min = 8.7826086956522,
Processor_1_Context_Switch_Time_Pct_StDev = 2.7060851320617,
Processor_1_D_1_Hit_Ratio_Max = 100.0,
Processor_1_D_1_Hit_Ratio_Mean = 95.5349143610013,
Processor_1_D_1_Hit_Ratio_Min = 83.3333333333333,
Processor_1_D_1_Hit_Ratio_StDev = 6.1908148498191,
Processor_1_D_1_Throughput_MIPs_Max = 24.7826086956522,
Processor_1_D_1_Throughput_MIPs_Mean = 18.11356043197,
Processor_1_D_1_Throughput_MIPs_Min = 12.6086956521739,
Processor_1_D_1_Throughput_MIPs_StDev = 3.3516026181173,
Processor_1_INT_1_Throughput_MIPs_Max = 150.0,
Processor_1_INT_1_Throughput_MIPs_Mean = 135.4107113794562,
Processor_1_INT_1_Throughput_MIPs_Min = 91.0049191848208,
Processor_1_INT_1_Throughput_MIPs_StDev = 17.2495019469227,
Processor_1_INT_2_Throughput_MIPs_Max = 137.3913043478261,
Processor_1_INT_2_Throughput_MIPs_Mean = 121.7977208467678,
Processor_1_INT_2_Throughput_MIPs_Min = 84.6802529866479,
Processor_1_INT_2_Throughput_MIPs_StDev = 14.9940888992868,
Processor_1_I_1_Buffer_Length_Max = 1.0,
Processor_1_I_1_Buffer_Length_Mean = 0.7,
Processor_1_I_1_Buffer_Length_Min = 0.0,
Processor_1_I_1_Buffer_Length_StDev = 0.4582575694956,
Processor_1_I_1_Hit_Ratio_Max = 96.8,
Processor_1_I_1_Hit_Ratio_Mean = 95.9988414763396,
Processor_1_I_1_Hit_Ratio_Min = 93.6416184971098,
Processor_1_I_1_Hit_Ratio_StDev = 0.850823915122,
Processor_1_I_1_Throughput_MIPs_Max = 286.5217391304348,
Processor_1_I_1_Throughput_MIPs_Mean = 257.9641249103743,
Processor_1_I_1_Throughput_MIPs_Min = 176.3879128601546,
Processor_1_I_1_Throughput_MIPs_StDev = 31.799615797033,
Processor_1_L_2_Hit_Ratio_Max = 100.0,
Processor_1_L_2_Hit_Ratio_Mean = 98.0977801268499,
Processor_1_L_2_Hit_Ratio_Min = 92.5,
Processor_1_L_2_Hit_Ratio_StDev = 2.9900119064018,
Processor_1_L_2_Throughput_MIPs_Max = 20.0,
Processor_1_L_2_Throughput_MIPs_Mean = 17.885707748116,
Processor_1_L_2_Throughput_MIPs_Min = 13.0007027406887,
Processor_1_L_2_Throughput_MIPs_StDev = 1.9420896258153,
Processor_1_Register_Rd_Buffer_Length_Max = 1.0,
Processor_1_Register_Rd_Buffer_Length_Mean = 0.7,
Processor_1_Register_Rd_Buffer_Length_Min = 0.0,
Processor_1_Register_Rd_Buffer_Length_StDev = 0.4582575694956,
Processor_1_Register_Rd_Throughput_MIPs_Max = 286.9565217391305,
Processor_1_Register_Rd_Throughput_MIPs_Mean = 258.9827319852018,
Processor_1_Register_Rd_Throughput_MIPs_Min = 174.9824314827829,
Processor_1_Register_Rd_Throughput_MIPs_StDev = 32.0461420690518,
Processor_1_Register_Wr_Throughput_MIPs_Max = 277.1975630983464,
Processor_1_Register_Wr_Throughput_MIPs_Mean = 248.4302195815351,
Processor_1_Register_Wr_Throughput_MIPs_Min = 170.0632466619818,
Processor_1_Register_Wr_Throughput_MIPs_StDev = 30.5313747616902,
Processor_1_Stall_Time_Pct_Max = 3.9130434782609,
Processor_1_Stall_Time_Pct_Mean = 2.4951121451687,
Processor_1_Stall_Time_Pct_Min = 1.1243851018974,
Processor_1_Stall_Time_Pct_StDev = 0.8058290563884,

```



```

Processor_1_Task_Delay_Max           = 1.282E-6,
Processor_1_Task_Delay_Mean         = 7.1638958829534E-7,
Processor_1_Task_Delay_Min          = 1.64E-7,
Processor_1_Task_Delay_StDev        = 2.9889837826806E-7,
SDRAM_1_Delay_Time_Max              = 6.6666666666667E-8,
SDRAM_1_Delay_Time_Mean             = 6.3095238095238E-8,
SDRAM_1_Delay_Time_Min              = 1.6666666666667E-8,
SDRAM_1_Delay_Time_StDev            = 1.2876968840943E-8,
SDRAM_1_Throughput_MBs_Max          = 111.304347826087,
SDRAM_1_Throughput_MBs_Mean         = 72.695652173913,
SDRAM_1_Throughput_MBs_Min          = 55.6521739130434,
SDRAM_1_Throughput_MBs_StDev        = 22.0094497663393,
TIME                                 = 2.3E-5}

```

The Processor stall time relates to the execution pipeline waiting for an execution unit to complete, assuming the pipeline is waiting for the results. The Processor idle time relates to time the Processor cannot start an execution because the execution unit is busy. The cache hit ratios are in percent (100.0 maximum). The throughput should correlate to the utilization percentages in the first statistics output, done for convenience, and a different way to look at performance.

Statistics_to_Plot

The Architecture Setup parameter, `Statistics_to_Plot` are simply lists of the statistics one wishes to plot from the above two statistics summaries. One just needs to add a comma-separated list to the parameter line, that match the above statistics names on the left. At the output of the Architecture_Setup block, one can add a relation, plus add a parameter to the relation (configure relation) named "width" to reflect how many traces one wishes to see on the plot.

Internal_Plot_Trace_Offset

The Architecture_Setup parameter, `Internal_Plot_Trace_Offset` indicates how many spaces between state plots. The reason this is valuable, is that the state plots reflect the internal execution unit buffering, based on pipeline execution.

Listen_to_Architecture

The Listen_to_Architecture pulldown menu, allows one to listen to specific blocks in the architecture, and watch operation at the detailed level. For example the Processor, if selected in this menu pulldown has the following sequence, on the start of a new task, including context switch cycles, prefetch, and task execution of instructions (500 ADDs):

```

Field_Name_Mapping Array:
{{Ext_Field      = "ID",
Int_Field        = "A_Address"}, {Ext_Field      = "A_Bytes",
Int_Field        = "A_Bytes"}, {Ext_Field      = "A_Data",
Int_Field        = "A_Data"}, {Ext_Field      = "A_IDX",
Int_Field        = "A_IDX"}, {Ext_Field      = "A_Instruction",
Int_Field        = "A_Instruction"}, {Ext_Field  = "A_Priority",
Int_Field        = "A_Priority"}, {Ext_Field    = "A_Source",
Int_Field        = "A_Source"}, {Ext_Field      = "A_Destination",
Int_Field        = "A_Destination"}, {Ext_Field  = "A_Task_ID",
Int_Field        = "A_Task_ID"}, {Ext_Field    = "A_Time",
Int_Field        = "A_Time"}}

"Processor_1 Bus In:
{A_Destination   = "\"Architecture_1\"",
A_Hop            = "\"output1\"",
A_Instruction    = "\"Hello\"",
A_Source         = "\"Port_1\""}"
"Processor_1 Bus In:
{A_Destination   = "\"Architecture_1\"",
A_Hop            = "\"output1\"",
A_Instruction    = "\"Hello\"",
A_Source         = "\"Port_3\""}"
"Processor_1 (ADD) virtual input at Cycle: 1"
"Processor_1 context switch at Cycle: 2"
"Processor_1 context switch at Cycle: 3"
"Processor_1 context switch at Cycle: 4"

```



```

"Processor_1 context switch at Cycle: 65"
"Processor_1 context switch at Cycle: 65"
"Processor_1 context switch at Cycle: 66"
"Processor_1 context switch at Cycle: 66"
"Processor_1 context switch at Cycle: 67"
"Processor_1 context switch at Cycle: 67"
"Processor_1 context switch at Cycle: 68"
"Processor_1 context switch at Cycle: 68"
"Processor_1 context switch at Cycle: 69"
"Processor_1 context switch at Cycle: 69"
"Processor_1 context switch at Cycle: 70"
"Processor_1 context switch at Cycle: 70"
"Processor_1 context switch at Cycle: 71"
"Processor_1 context switch at Cycle: 71"
"Processor_1 context switch at Cycle: 72"
"Processor_1 context switch at Cycle: 72"
"Processor_1 context switch at Cycle: 73"
"Processor_1 context switch at Cycle: 73"
"Processor_1 context switch at Cycle: 74"
"Processor_1 context switch at Cycle: 74"
"Processor_1 context switch at Cycle: 75"
"Processor_1 context switch at Cycle: 75"
"Processor_1 context switch at Cycle: 76"
"Processor_1 context switch at Cycle: 76"
"Processor_1 context switch at Cycle: 77"
"Processor_1 context switch at Cycle: 77"
"Processor_1 context switch at Cycle: 78"
"Processor_1 context switch at Cycle: 78"
"Processor_1 context switch at Cycle: 79"
"Processor_1 context switch at Cycle: 79"
"Processor_1 context switch at Cycle: 80"
"Processor_1 context switch at Cycle: 80"
"Processor_1 context switch at Cycle: 81"
"Processor_1 start pipeline at Cycle: 81"
"Processor_1 (ADD) to I_1 (1, 3) at Cycle: 81"
"Processor_1 (ADD) to D_1 (1, 4) at Cycle: 81"
"Processor_1 (2) empty at Cycle: 81"
"Processor_1 (3) empty at Cycle: 81"
"Processor_1 (4) empty at Cycle: 81"
"Processor_1 prefetch I_1 internal (L_2) cache at cycle: 82"
"Processor_1 I_1 complete at cycle: 82"
"Processor_1 Register_Rd complete at Cycle: 82"
"Processor_1 start pipeline at Cycle: 82"
"Processor_1 (ADD) to I_1 (1, 3) at Cycle: 82"
"Processor_1 (ADD) to D_1 (1, 4) at Cycle: 82"
"Processor_1 start pipeline at Cycle: 82"
"Processor_1 wait done I_1 (2, 3) at cycle: 82"
"Processor_1 (3) empty at Cycle: 82"
"Processor_1 (4) empty at Cycle: 82"
"Processor_1 I_1 complete at Cycle: 82"
"Processor_1 L_2 busy at Cycle: 82"
"Processor_1 Register_Rd complete at cycle: 82"
"Processor_1 start pipeline at Cycle: 82"
"Processor_1 (ADD) to I_1 (1, 3) at Cycle: 82"
"Processor_1 (ADD) to D_1 (1, 4) at Cycle: 82"
"Processor_1 start pipeline at Cycle: 82"
"Processor_1 wait done I_1 (2, 3) at Cycle: 82"
"Processor_1 start pipeline at Cycle: 82"
"Processor_1 wait done D_1 (3, 4) at cycle: 82"
"Processor_1 (ADD) to INT (3, -1) at Cycle: 82"
"Processor_1 (4) empty at Cycle: 82"
"Processor_1 I_1 complete at Cycle: 83"
"Processor_1 INT_1 busy at Cycle: 83"
"Processor_1 L_2 busy at Cycle: 83"
"Processor_1 Register_Rd complete at cycle: 83"
"Processor_1 start pipeline at Cycle: 83"
"Processor_1 (ADD) to I_1 (1, 3) at Cycle: 83"
"Processor_1 (ADD) to D_1 (1, 4) at Cycle: 83"
"Processor_1 start pipeline at Cycle: 83"
"Processor_1 wait done I_1 (2, 3) at Cycle: 83"
"Processor_1 start pipeline at Cycle: 83"
"Processor_1 wait done D_1 (3, 4) at Cycle: 83"
"Processor_1 (ADD) to INT (3, -1) at Cycle: 83"
"Processor_1 start pipeline at Cycle: 83"
"Processor_1 (ADD) to D_1 (4, -1) at Cycle: 83"
"Processor_1 I_1 complete at Cycle: 83"
"Processor_1 INT_1 busy at cycle: 83"

```



"Processor_1 L_2 busy at Cycle: 83"
"Processor_1 Register_Rd complete at Cycle: 83"
"Processor_1 start pipeline at Cycle: 83"
"Processor_1 (ADD) to I_1 (1, 3) at Cycle: 83"
"Processor_1 (ADD) to D_1 (1, 4) at Cycle: 83"

Listen_to_Architecture: Pipeline View

PPC_7410_1	FPU_s_add	DECODE	EXECUTE	STORE	Active	@Instr 0	@Cycle 202
PPC_7410_1	FPU_s_add	FPU_s_add	EXECUTE	STORE	Active	@Instr 1	@Cycle 203
PPC_7410_1	FPU_s_add	FPU_s_add	FPU_s_add	STORE	Active	@Instr 2	@Cycle 204
PPC_7410_1	*b	FPU_s_add	FPU_s_add	FPU_s_add	Stall	@Instr 3	@Cycle 205
PPC_7410_1	*b	FPU_s_add	FPU_s_add	FPU_s_add	Stall	@Instr 3	@Cycle 206
PPC_7410_1	*b	FPU_s_add	FPU_s_add	FPU_s_add	Active	@Instr 3	@Cycle 207
PPC_7410_1	b	*b	FPU_s_add	FPU_s_add	Stall	@Instr 4	@Cycle 208
PPC_7410_1	b	*b	FPU_s_add	FPU_s_add	Stall	@Instr 4	@Cycle 209
PPC_7410_1	b	*b	FPU_s_add	FPU_s_add	Active	@Instr 4	@Cycle 210

State Plot

The State Plot for the above Processor, at top level instruction level:

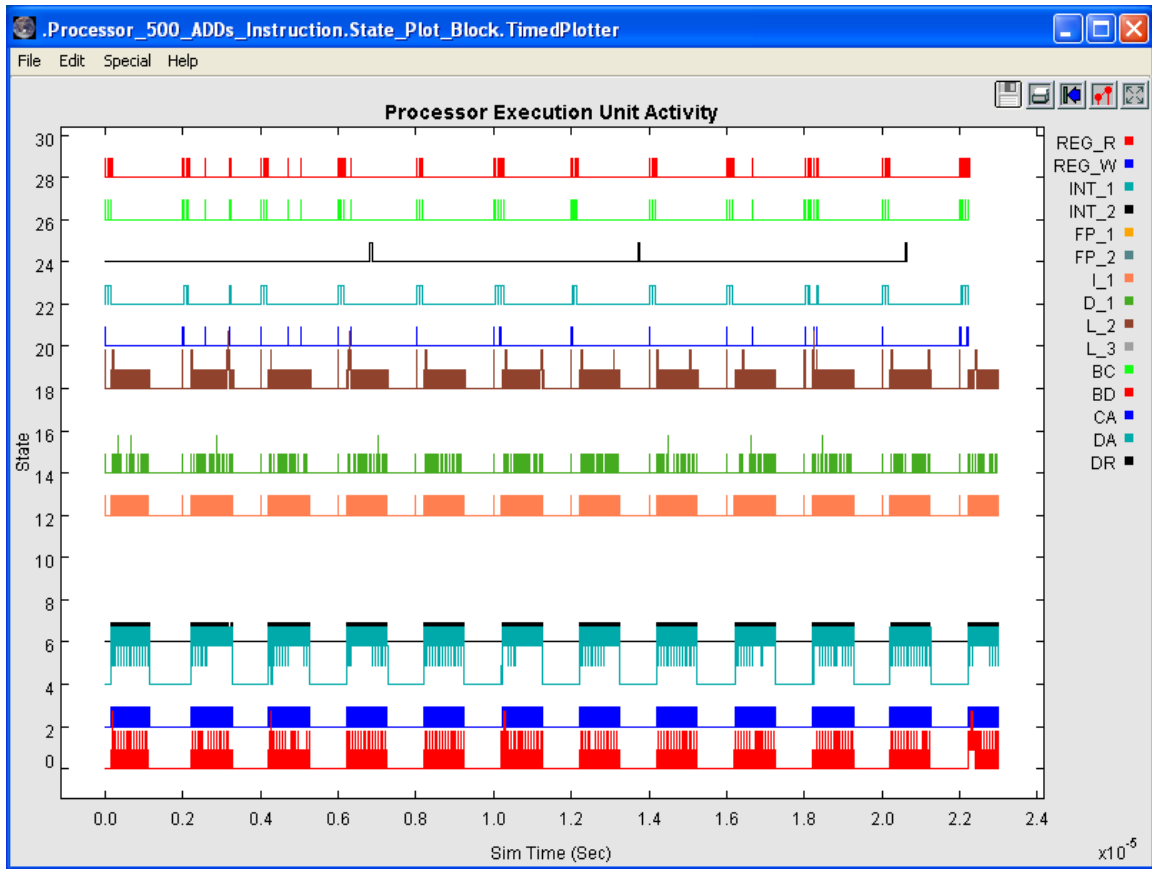


Figure 55 Architecture_State Plot Output

One can see in Figure 9, the register activity at the bottom in red (Reg_R: Register Read), blue (Reg_W: Register Write) traces. The next two traces, INT_1 and INT_2 are the two integer units, with more activity from INT_1 for these tasks. The next two traces, L_1, and D_1 are the lower level cache activity, the next trace up is the L_2 cache activity, including on-going pre-fetch during instruction execution. The next two traces, CA (Cache_1 Access), and DA (DRAM Access) show the equivalent L_3, SDRAM memory activity. The black trace DR (DRAM Refresh) shows dynamic RAM refreshing. The top two traces BS (Bus Controller), BS (Bus Data) show the external bus traffic. Each task here can be expanded to see individual instructions executing, and their relation to all of the architecture resources.

Processor Model Features

The statistical processor block can be hierarchical to support multiple processor cores. The processor block will consist of the instruction stack, instruction pipeline, and external processor/bus/cache/DRAM blocks. A single processor block will be able to call a second processor block for out of order instruction processing as instruction streams. For example, the PowerPC has two integer/floating point instruction streams, and the AltiVec Single Instruction Multiple Data (SIMD) unit that can be modeled as a separate processor. The statistical processor

block can dispatch instructions from a single pipeline to multiple external execution units, and can execute them in parallel.

Instructions can be sent from the behavioral level directly to the processor, or via and RTOS function, depending on the type of processor being modeled, whether common microprocessor, FPGA, ASIC, DSP, or custom microcontroller. Typically, the basic or macro level instruction will be requested through the RTOS or behavioral directly, and executed on the processor model. If common ADD, SUB, MULT, DIV microprocessor instructions, and then a behavioral block can send requests to the processor model for execution as a single instruction or task level set of instructions. If FPGA, ASIC, or DSP, then a behavioral block can send FFT level instruction with operands to the processor model for execution as a single instruction or task level set of instructions, using a Data Structure that has the additional information for the FFT algorithm, for example.

The statistical processor block will support thread context switching as a result of RTOS pending tasks, or hardware interrupts sent directly to the processor model. Context switching cycles is a parameter of the processor model. Branch prediction can also be performed by a processor model by annotating instruction mnemonics with a '*' in front of BRCH type instructions, assuming the Instruction_Set block contains the '*' in front of the instruction mnemonic. If the '*' is appended to the instruction name in the task, then perform a pipeline flush as a result of instruction missed branch prediction. If no '*' character is appended, then perform branch with branch prediction correct. In other words, branch prediction can be controlled from the instruction sequence coming in, or expanded in the Processor by external execution, and setting the field called A_Branch to true, this will also cause a pipeline flush to occur if the branch prediction is incorrect. The '*' instruction simply sets this flag, so branch prediction can be controlled at the instruction level, or at the pipeline level with the Data Structure field A_Branch.

The statistical processor block also supports DMA style operations, whereby once the instruction starts execution; it is placed in an interrupt queue, awaiting a hardware interrupt when the instruction completes to resume operation. These instructions are denoted with a '#' character in front of the instruction name, assuming the Instruction_set block contains the '#' in front of the instruction mnemonic. The executing task is suspended, and a new one starts, while the first one completes. The DMA_Controller is the block that executes the DMA Instructions driven by a Database block with the specific instruction.

DMA Database Format:

A_Task	A_Instruction	A_IDX	A_Task_Source	Burst	A_Task_Addre	A_Addre	A_Addre	A_Command	A_Bytes	A_Priorit	A_Destination
MyTask	MOV	0	SDRAM_1	8	3	0	0	Read	100	0	DMA_1
MyTask	MOV	1	SDRAM_1	8	3	0	0	Read	100	0	DMA_1
MyTask2	MOV	1	SDRAM_1	8	2	0	0	Write	100	0	DMA_1
MyTask3	MOV	1	SDRAM_1	8	3	0	0	Read_Write	100	0	DMA_1

The statistical processor block can also call a VisualSim scheduler, Scheduler_SW or Scheduler_HW, simply by using the "task" Action in the pipeline, and using the Execution_Location as the Scheduler name. The Condition field of the pipeline must reference an instruction in the Instruction_Set, and the Processor will append the task time, sent to the scheduler, based on the number of instruction cycles, and the processor speed. The scheduler will then return the instruction to the pipeline upon completion.

Here is a processor model connecting directly to a scheduler:

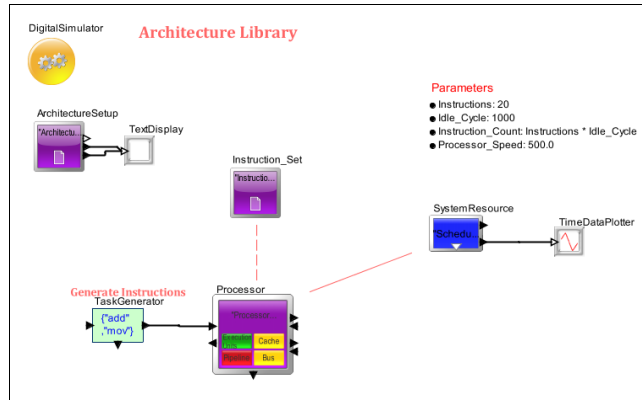


Figure 56 Processor to Scheduler_SW for instruction execution

The pipeline setup for this configuration, note SCHED_1, the name of Sched_SW block in above model. INT_1 is where the instruction is defined, calculates the time needed for scheduler execution.

```

/* First row contains Column Names. */
Stage_Name Port_or_Virtual Action Condition ;
1_PREFETCH L_1 read none ;
2_DECODE none exec none ;
3_EXECUTE L_1 wait none ;
3_EXECUTE SCHED_1 task INT_1 ;
4_STORE L_1 write none ;

```

The statistical processor block can also call a Task_Generator block directly from the pipeline and return after completion. The Task_Generator could perform an operation on the instruction, or modify the branch prediction field (A_Branch), for example.

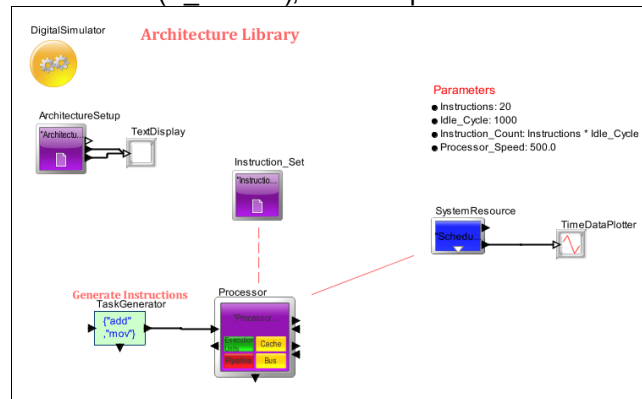


Figure 57 Processor to TaskGenerator for pipeline, instruction execution

The Soft_Gen block is used to generate new tasks to execute on the processor. The mix of instructions in the task is read from the file that is referenced by the parameter Read_My_Instruction_Mix_Table.

The instructions are placed in the A_Instruction field of the Processor_DS template Data Structure and sent on the output port.

The generated instruction assumes that each instruction takes exactly one cycle to execute for the purpose of generating the tasks.

The Read_My_Instruction_Mix_Table file has two parts. The first Part defines instruction mnemonics for each type. The number of types is the parameter *Number_Instruction_Types*. There is a line delimiter ';' at the end of every line. The instructions will be randomly selected based on the Percentages of each type (Pct) in the second half of table

The second Part of the file defines individual tasks, and the mix within each task. Each line starts with the task name and is followed by the duration of the task. The duration can either be in number of instructions or the duration of the task in number of cycles (Relative_Time). This generator assumes that each task has an execution time of 1 cycle. This will be followed by the Type Name and Percentage for each of the types in Part One. Each task line must end with the line delimiter ';'. The number of types must match the parameter and the number of items in Part One. The total percentages of all types cannot exceed 100%.

Sample Template of SoftGen or TaskGenerator is shown below

```

VisualSim Architect - file://E:/VisualSim/VisualSim16_64_May...re/Processor/Instruction_Mix_Table.txt
File Help
|
/* My Instruction Mix Table -- First Part defines instruction mnemonics for each type, line delimiter ';'
   these instructions will be randomly selected based on the Percentage (Pct) in second half of table

Description      Type (used below)      Mnemonic List  */
Integer          INT                MV_MOV MV_MVN MV_MRS MV_MSR ;
ARITHMETIC       FP                ART_ADD ART_ADC ART_SBC ART_MLAS ;
LOGICAL          LOG                LGL_TEQ LGL_ORR LGL_AND ;
Input_Output     IO                LD_LDR ;
Branch           BRCH              *BR_B ;

/* -- Second Part defines individual tasks, and the mix within each task, starts with the task name,
   and either relative time (convert to number of instructions) or number of Instructions that
   will be used by the percent entries line delimiter ';' The number of types just needs to match
   the list above and there is a column for each type, plus the Pct cannot exceed 100%....

Note:           The Number_Instructions assumes one instruction per cycle of the Processor,
                which may be optimistic for multi-cycle instructions.

A_Task_Name     Relative_Time(double)_or
                Number_Instructions(int) Type Pct  Type Pct  Type Pct  Type Pct  Type Pct  */
My_Task_1       500                INT 10   FP 48   LOG 10   IO 7    BRCH 25 ;
My_Task_2       700                INT 15   FP 25   LOG 60   IO 0    BRCH 0  ;
My_Task_3       100                INT 40   FP 38   LOG 10   IO 7    BRCH 5  ;

```

Cache and Memory Overview

The Memory blocks model Cache and SDRAM activity in terms of performance, using threads and a relative addressing methodology. This means the exact address is not needed in the model saving considerable time in adding it to the model, or in the time the simulation needs to process specific addresses. Relative addressing provides sufficient internal address information to model cache activity accurately, above 80%. The model generates, uses internal relative addresses without the user needing to perform any special processing in a model.

The Processor can be configured with multiple caches that can be used in the Processor pipeline. Here is an example of an I_1 (Instruction), D_1 (Data), L_2 (Hierarchical) cache structure one can define in a model:

```

I_1 {Cache_Speed_Mhz=500.0, Size_KBytes=64.0, Words_per_Cache_Line=16,
     Cache_Miss_Name=L_2}
D_1 {Cache_Speed_Mhz=500.0, Size_KBytes=64.0, Words_per_Cache_Line=16,
     Cache_Miss_Name=L_2}
L_2 {Cache_Speed_Mhz=500.0, Size_KBytes=64.0, Words_per_Cache_Line=16,
     Cache_Miss_Name=Cache_1}

```

These definitions define the cache speed, cache size, cache words per line, and the cache miss name. If a miss occurs, then this is the cache that is called. One notes the I_1 and D_1 caches call the L_2 cache on a miss, and the L_2 cache calls the cache "Cache_1" which is external to

the Processor chip, going through an external bus. The I_1, D_1 go to the L_2 via internal bus with no contention.

The Processor block then uses these caches with the Pipeline description:

Stage Name	Execute Location	Action	Condition ;
1_PREFETCH	I_1	instr	none ;
1_PREFETCH	D_1	read	none ;
2_DECODE	I_1	wait	none ;
3_EXECUTE	D_1	wait	none ;
3_EXECUTE	INT	exec	none ;
4_STORE	D_1	write	none ;

This is the basic four stage pipeline with prefetch, decode, execute, and store cycles. One will notice only the first two caches are used in the pipeline description, since it accesses these memories, and the L_2 cache does not appear, since it is accessed by I_1, D_1 only for a miss case.

The Cache has also been updated to support shared caches between Processor blocks, differing access words per access, and DMA activity. The following parameters can be used in the cache:

```
I_1 {Processor_Name=Processor_Name, Cache_Speed_Mhz=500.0,
Size_KBytes=64.0, Words_per_Cache_Access=8, Words_per_Cache_Line=16,
Cache_Miss_Name=L_2}
D_1 { Processor_Name=Processor_Name, Cache_Speed_Mhz=500.0,
Size_KBytes=64.0, Words_per_Cache_Access=8, Words_per_Cache_Line=16,
Cache_Miss_Name=L_2}
L_2 { Processor_Name=Processor_Name, Cache_Speed_Mhz=500.0,
Size_KBytes=64.0, Words_per_Cache_Access=8, Words_per_Cache_Line=16,
Cache_Miss_Name=Cache_1}
```

Cache Thread

A Cache thread is maintained for each thread, or task, in the Processor using relative addressing. This also means each cache has its own list of active threads. When a thread starts in the processor, the I_1, D_1 caches begin to prefetch a line from the Cache during the context switchover time parameter:

Context_Switch_Cycles: 100

The user can vary this according to the Processor setup. Once, the Context_Switch_Cycles period is over, the instructions will be processed, and each instruction will begin processing. In addition, the total number of cache threads is maintained by the Processor, in determining how much memory is available to each thread. This information is used when a cache crosses a cache line boundary to determine if there is a cache miss due to less memory per thread.

Cache Misses

Cache misses are primarily determined by the relative addressing of instructions. This is dependent on other Processor activity, and if the cache is external via a bus, then the time can affect the prefetch mechanism. Once, a cache miss occurs then it will literally prefetch the next level memory for this word.



Cache Instrs, Reads, Writes

The Processor cache perform cache operations for instr (I_1), reads (D_1), writes (D_1); depending on the pipeline "Action" column. A cache "instr" action performs a read without waiting in the pipeline for the result, a "read" performs a read with the pipeline waiting for the result, and a "write" just writes to the cache.

If a cache "read" is not complete in the right pipeline stage, due to off-chip access, or other Processor activity, then the pipeline will stall, statistics provided. Pipeline stalls are sometimes mis-interpreted as cache misses. The Processor model keeps track of this case.

In addition, if the Processor/cache pipeline, there are many variations that have been designed, runs out of sufficient buffering, due to long instructions prior; then the Processor will insert an idle cycle to allow the Processor to catch up, statistics provided for this case as well.

This suggests that the instruction sequence can introduce cache related delays not associated with the cache itself, more the Processor, or external processing delay. This can be as important as a very detailed cache model.

DRAM Memory

The external DRAM memory block can perform accesses based on user defined instruction delays, obtained from data sheets of the respective memory. The parameter field looks like:

"Read 10.0, Prefetch 10.0,Write 7.0, ReadWrite 8.0, Erase 6.0"

This is a very flexible way to model individual instruction processing at the SDRAM level. The only restriction is any "read" command start with "Read", and "Prefetch" maintained for internal use.

In addition, there is a "Memory_Type" pull down for different memory technologies, including SDR, DDR, DDR2, DDR3, QDR, or RAMBUS type of clocking access methodologies. Currently, a memory block is in development that will also maintain "banks" of memory for parallel access schemes, or for page style of access that is critical to application performance. The DDR will process two requests for each Memory_Speed_Mhz period, for example.

DRAM Processing

DRAM memory processing takes the parameter for Memory_Speed_Mhz as the memory controller speed that will access the first word in a memory request, and the above delays for individual instructions will then be added to this memory controller access, to model burst access of N words properly. The number of memory cycles will then be determined for the read or write style access. "Write" commands will not return any value to the bus, to properly model write commands.

External Bus

Our Architecture Bus, also processes Cache and DRAM requests according to command type, output the first word at the proper time for large transfers to/from memory to maintain the proper bus timing, as related to arbitration, or protocol differences .

Cache Summary

Our Cache memory methodology is very efficient, models relative addresses, prefetched cache lines/words, total cache thread activity, and pipeline stalls or idle cycles. Cache misses are directed to the next level memory, which in turn models its own prefetch independent of



the requesting cache. The Processor statistics also provides an estimate of "how" much cache memory is being used based on words accessed; this is another measure of cache size, dependent on the Cache size parameter entered, secondary effect. This statistic appears as:

```
Processor_1_D_1_KB_per_Thread_Max = 0.052,  
Processor_1_D_1_KB_per_Thread_Mean = 0.0184,  
Processor_1_D_1_KB_per_Thread_Min = 0.0,  
Processor_1_D_1_KB_per_Thread_StDev = 0.0192831532691
```

This can give an estimate of actual cache usage, based on the number of threads used, and cache accesses.

Bus, Switch and Controller Toolkit

Bus Arbiter

Block Description

The BusArbiter helps the master (traffic generator) and slave (traffic receiver) devices to communicate smoothly by routing the generated bus traffic. The BusArbiter is similar to the existing Bus Port, Bus Controller in addition of Arbitration modes that allows preemption and enable user to implement their own bus processing algorithms.

Block Usage

An arbiter that controls the bus traffic can be depicted using the BusArbiter. This is connected to BusInterface to represent the linear bus model. Block supports 1) FCFS, 2) FCFS with preemption and 3) CUSTOM modes of arbitration schemes.

Functionality

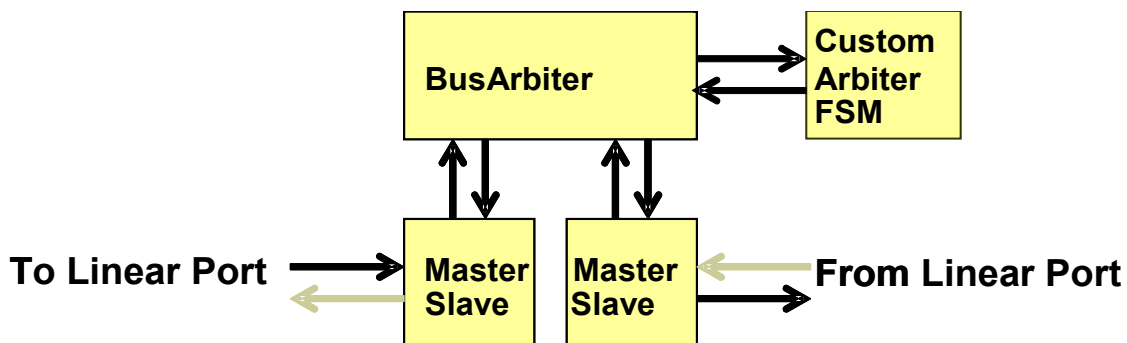


Figure 58 Bus Arbiter Flow Diagram

The Linear Bus topology allows only one bus master to actively use the bus at one time. The Bus Arbiter block maintains an inbuilt routing table that can determine the source and destination devices for processing a transaction. The following arbitration modes can select which master to gain the bus access.

Internal Mode: FCFS

A default mode, accepts requests one at a time serves them in incoming order. Also the Linear Controller block monitors the requests for priorities A_Priority of incoming data structure and choose the master with highest priority request as the next bus transaction master. Preemption is allowed when higher priority request is waiting while a lower priority request is in progress. Each time before start processing new requests the arbiter checks for preempted requests if any. The priority of preempted request is again if low, the arbiter will increment its priority in order to get a better chance of being selected in the next time. This allows the lower priority transactions to not wait too long as its priority is increased each time it is not selected.

In short, the preempted request will be selected if its priority is high, else higher priority request among the incoming data structures will be selected and increment the priority of preempted

request. The requests with priority A_Priority of incoming data structure are equal then very first requested Master is the default Master.

External Mode: CUSTOM

This mode enables users to easily implement their own arbitration scheme to suit their particular needs. An user can allowed to access and modify the core Bus Transaction data structure externally using the input and output arbiter ports of Linear Controller block. For example, one can modify the order of transaction; can customize bus selection of next transaction; issuing an address/control cycle for any fragment is allowed. This can be performed by Processing, Decision and Virtual Machine blocks that supports RegEx functions.

User can implement their own arbitration algorithms using the following RegEx (...) functions.

Name of the RegEx functions	Functionality
readBusPorts("Arch_Bus_Name")	Read the available Linear Port Names as an ArrayToken of Strings.
lengthBusQueue("Arch_Bus_Name", "Port_Name")	Obtain the length of the named Port Input Queue.
lengthBusPreempt("Arch_Bus_Name", "Port_Name")	Obtain the length of the named Port Preempt Queue.
clearBusQueue("Arch_Bus_Name", "Port_Name")	Clear the named Port Input Queue
clearBusPreempt("Arch_Bus_Name", "Port_Name")	Clear the named Port Preempt Queue
readBusQueue ("Arch_Bus_Name", "Port_Name", Position) (String, String, int)	Obtain a copy of the Transaction (RecordToken) by Port Name of Input Queue by position argument, assumes user obtained length prior.
readBusPreempt ("Arch_Bus_Name", "Port_Name", Position)	Obtain a copy of the Transaction (RecordToken) by Port Name of Preempt Queue by position argument, assumes user obtained length prior
removeBusQueue ("Arch_Bus_Name", "Port_Name", Position) (String, String, int)	Remove the Transaction (RecordToken) by Port Name of Input Queue by position argument, assumes user obtained length prior.
removeBusPreempt ("Arch_Bus_Name", "Port_Name", Position)	Remove the Transaction (RecordToken) by Port Name of Preempt Queue by position argument, assumes user obtained length prior.
writeBusQueue ("Arch_Bus_Name", "Port_Name", Position, Transaction) (String, String, int, RecordToken)	Write the Transaction (RecordToken) by Port Name of Input Queue by position argument, assumes user obtained length prior.
writeBusPreempt ("Arch_Bus_Name", "Port_Name", Position, Transaction)	Write the Transaction (RecordToken) by Port Name of Preempt Queue by position argument, assumes user obtained length prior.
preemptedBusQueue ("Arch_Bus_Name", "Port_Name")	Checks the named Port Input Queue been preempted with a higher priority transaction? true means yes, false means no. Also return false if named Port Input Queue is a last fragment.



The string arguments,

“Arch_Bus_Name” – is the concatenation of parameters Architecture_Name and Bus_Name of Linear Controller.

“Port_Name” – is used in Point to Point bus.

To find the first fragment of a transaction one can follow the condition in processing blocks as

```
First_Word = (input.A_Addr_Ctrl_Flag && input.A_First_Word && (input.A_Bytes ==  
input.A_Bytes_Remaining + input.A_Bytes_Sent)) ? true : false
```

The A_Addr_Ctrl_Flag of the Bus Transaction data structure is always true by default. One can ignore the Address/Control cycle by setting the field A_Addr_Ctrl_Flag to false. The field A_First_Word is true for the first word transfer on every burst, else false.

```
Preemption = First_Word ? false : preemptedBusQueue(Arch_Name, Arch_Name)
```

The above RegEx expression determines the occurrences of preemption; if the active transaction is the very first word transfer of a burst then there is no preemption. Else allows the higher priority transaction if waited to preempt the lower priority transaction.

Statistics

IO_per_sec: Input and output transactions per second.

Input_Buffer_Occupancy_in_Words: No. of words occupied in FIFO_Buffers Input queue.

Preempt_Buffer_Occupancy_in_Words: No. of words occupied in FIFO_Buffers Preempt queue.

The Architecture Setup block in the statistic name *Bus_Name_Statistic_Name* each collects the Statistic sample.

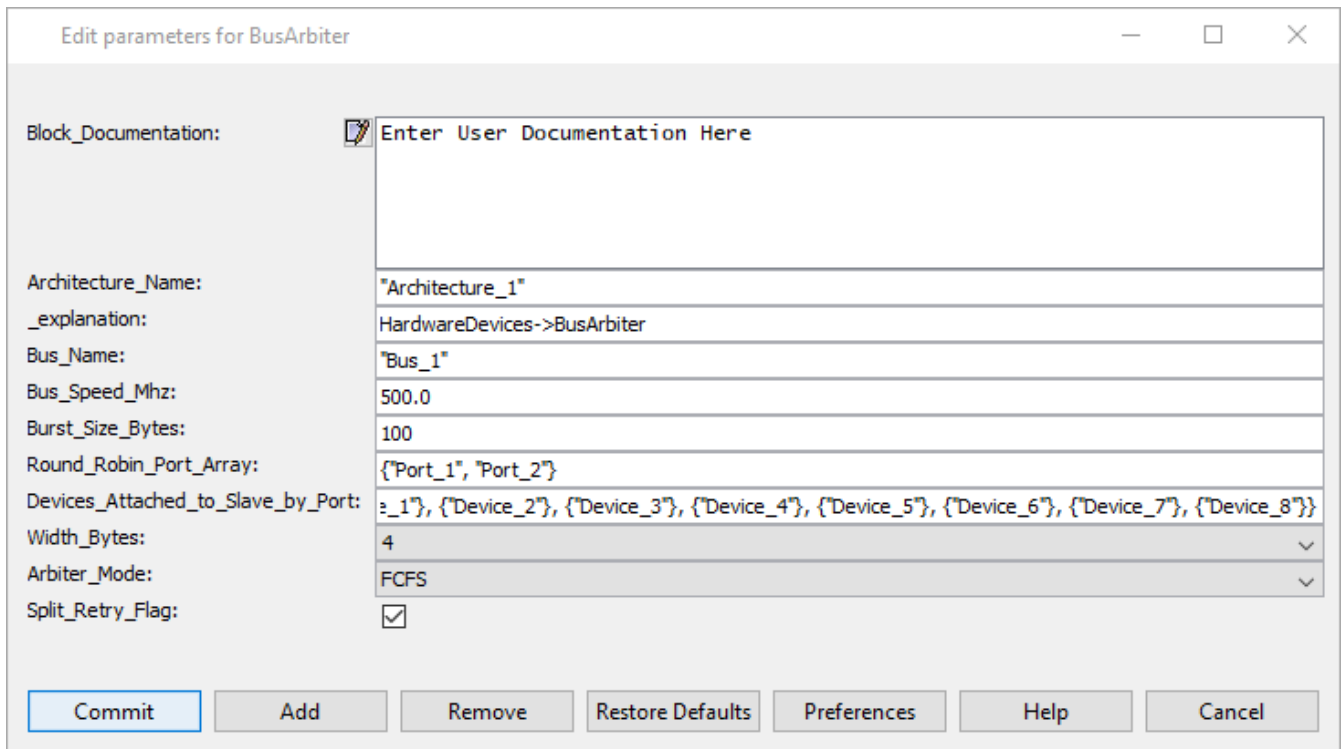
State Plots

BD: Bus Data

BC: Bus Control

These are Bus signals shows the external bus traffic. The Linear Bus updated the state transition with the Architecture Setup block. The Architecture Setup block sends the plotting information to the hierarchical State_Plot block through the virtual connection (IN block).

Configuration of Parameters



Parameter	Value
Block_Documentation:	<input checked="" type="checkbox"/> Enter User Documentation Here
Architecture_Name:	"Architecture_1"
_explanation:	HardwareDevices->BusArbiter
Bus_Name:	"Bus_1"
Bus_Speed_Mhz:	500.0
Burst_Size_Bytes:	100
Round_Robin_Port_Array:	{"Port_1", "Port_2"}
Devices_Attached_to_Slave_by_Port:	Device_1, {"Device_2"}, {"Device_3"}, {"Device_4"}, {"Device_5"}, {"Device_6"}, {"Device_7"}, {"Device_8"}
Width_Bytes:	4
Arbiter_Mode:	FCFS
Split_Retry_Flag:	<input checked="" type="checkbox"/>

Buttons: Commit, Add, Remove, Restore Defaults, Preferences, Help, Cancel

Figure 59 BusArbiter Configuration Window

Architecture_Name

Name of the common architecture to which blocks in the model belong to, Type is String. There can be different architectures existing; hence the unique Architecture name is defined with each block.

Bus_Name

Name of the bus model, Type is String. The linear ports that constitute the physical bus need to be configured with the same bus name.

Bus_Speed_Mhz

Speed of the linear bus model, Type is double. This determines the rate at which the linear bus model can operate.

FIFO_Buffers

It is the Length of the FIFO buffers. There are two FIFO buffers 1) Input queue and 2) Preempt queue, Type is integer. Default value is 8. All requests are initially added to the FIFO_Buffers input queue and each request from the queue is processed further. Also can set priorities among each request to sort higher priority request in front of the FIFO_Buffers input queue in descending order. FIFO_Buffers preempt queue holds all the preempted requests.

Burst_Size_Bytes

The maximum size of the data that can be sent across the bus at any instant of time, Type is integer. Default is 100.

Width_Bytes

Width of the data traffic in words, Type is pulldown.

The values are 2, 4, and 8.

Arbiter_Mode

Represents modes of bus arbitration scheme. The values are 1) FCFS 2) CUSTOM. Type is pull-down. Default mode is FCFS.

Split_Retry_Flag

By selecting the split and retry operation is enabled. It provides a mechanism for slaves to release the bus when they are unable to supply the data for a transfer immediately. This mechanism allows the transfer to finish on the bus therefore allow a higher priority master to get access to the bus.

Bus Interface

Block Description

Represents a Bus Interface that can receive data traffic and send data traffic out on the Bus Arbiter. Co-ordinates data transfer by handling control to the Linear Controller.

Block Usage

Can be used to depict any number of linear ports that are linked together to form a Linear Bus model connected to several devices (master and slave).

Functionality

Bus Interface adds the requests received from the master on to a queue. Then forwards the request to the Bus Arbiter. The Bus Arbiter identifies the right master and slave and sends out the request to the BusInterface. The BusInterface then transfers the data to the slave.

The BusInterface tentatively has six ports: input from child Bus Interface blocks (1 port), output to parent Bus Interface or BusArbiter (1 port), input/output for two Bus Interface (4 ports).

Configuration of Parameters

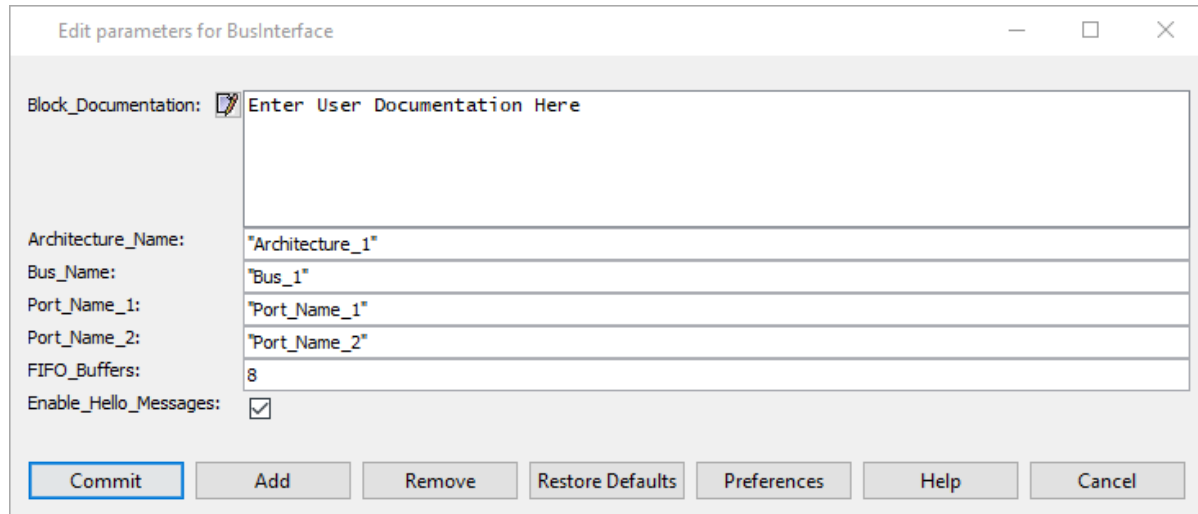


Figure 60 BusInterface Configuration Window

Architecture_Name

Name of the common architecture to which blocks in the model belong to, Type is String. There can be different architectures existing; hence the unique Architecture name is defined with each block.

Bus_Name

Name of the linear bus model, Type is String. The linear ports that constitute the physical bus need to be configured with the same linear bus name.

Port_Name_1

Name of the BusInterface that is connected to the linear bus, Type is String. The routing table in the Architecture_Setup block refers to this port name as the hop name.

Port_Name_2

Name of the BusInterface that is connected to the linear bus, Type is String. The routing table in the Architecture_Setup block refers to this port name as the hop name.

FIFO_Buffer

It is the Length of the FIFO buffer. Length of Input queue, Type is integer. Default value is 8. All requests are initially added to the FIFO_Buffer input queue and each request from the queue is send to BusArbiter.

DMA_Controller

Block Description

The DMA_Controller block is a hardware DMA block that can be used in a model. The block can receive requests directly from the processor block or from the Req port.

Functionality

The DMA Controller block contains multiple channels that are defined using the DMA_Channels parameter. A database is required to characterize the task operation of the DMA block. Every channel has an individual queue.

When a request comes in, the DMA block matches the A_Task_Name and A_Instruction fields of the data structure with the database content. If there is no a match, then an error is reported. If there is a match, then the attributes of this request are taken from the matched line in the database. The A_Task_Address field of the database will assign the request to a channel. The burst size, command (Read/Write), data size (A_Bytes) and priority (A_Priority) are taken from the database. The queue is not reordered for the priority. If there are multiple lines for the A_Task_Name and A_Instruction, then the lines are executed in the sequence listed in field A_Idx. The DMA is typically connected to Bus on the right-side. A processor identifies a DMA operation by prefix # for the instruction in the Instruction_Table. The data structure arriving at the processor does not need to have the # prefix, just the instruction name. The database name is required in the optional Processor_Setup parameter called DMADatabase.

The first request in each channel is sent out on the bus. The channel is locked until the data structure completes all the lines associated with this transaction. When the transaction is completed, it sends the data structure to the ack port if it came from via the req port or back to the processor pipeline. The Device_to_DMA and DMA_to_Device parameters can be used to model any hardware delays to improve accuracy. The field A_Task_Source specifies the destination. The A_Destination field in the table specifies which DMA to use for this transfer as there can be multiple DMA blocks in a model. Each DMA block requires a separate Database setup block.

The DMA Controller is configurable using a Database setup. The Database fields are:

A_Task_Name	Processor Task
A_Instruction	Processor Instruction
A_IDX	Instruction index
A_Task_Source	Processor internal memory or external memory
Burst_Word_Size	Burst size
A_Task_Address	Channel number
A_Command	Read, Write or Read_Write transaction
A_Bytes	Total bytes of transaction
A_Priority	Priority index of the transaction, used internally.
A_Destination	DMA block name

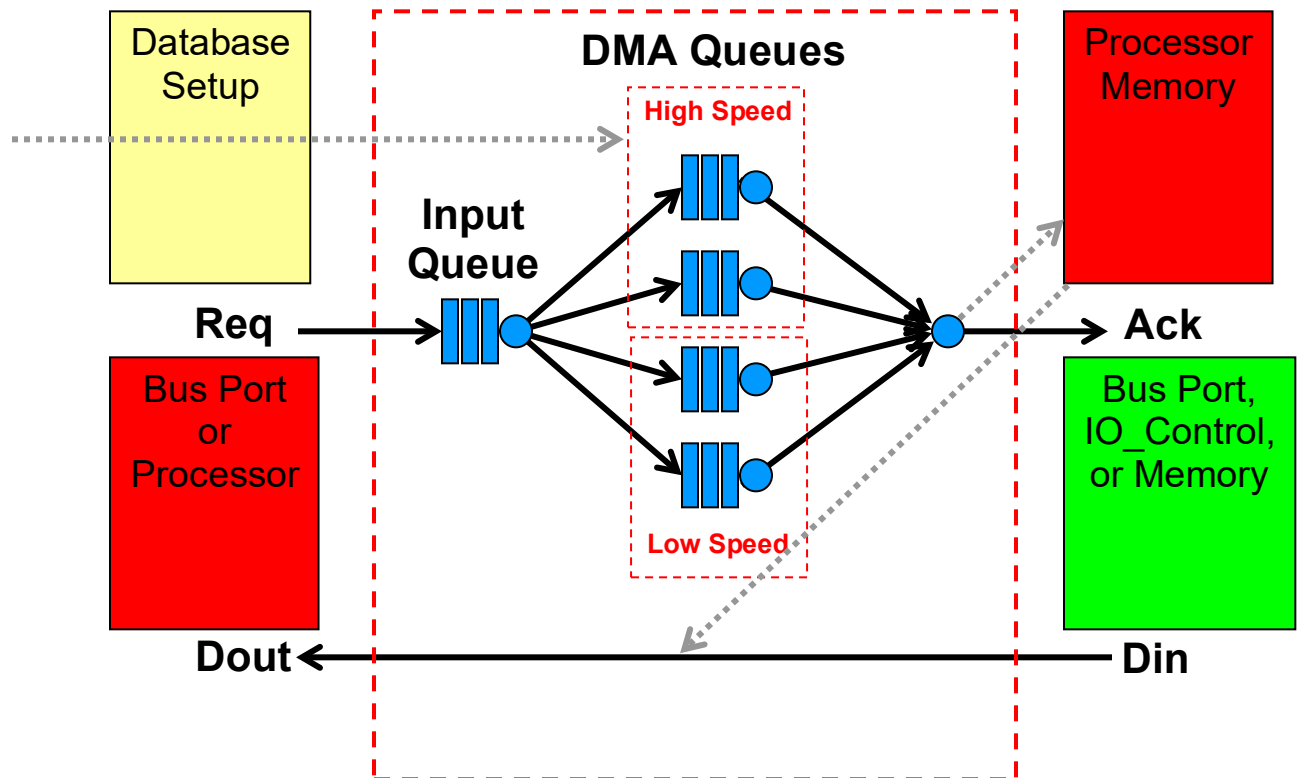


Figure 61 DMA Flow Diagram

Example csv file:

A_Task_Name	A_Instruction	A_IDX	A_Task_Source	Burst_Word_Size	A_Task_Address
MyTask	MOV	0	L_2	8	1
MyTask2	MOV	1	SDRAM_1	8	2
MyTask3	MOV	1	SDRAM_1	8	3

A_Command	A_Bytes	A_Priority	A_Destination
Read	100	0	DMA_1
Write	100	0	DMA_1
Read_Write	100	0	DMA_1

DMA execution of Sequence of Tasks

The Xilinx Multiport memory controller supports transfer of sequences of Reads and Writes. The VisualSim DMA Controller supports the same by executing sequence of tasks defined in the Database. Once a DMA transaction completes it looks to see if there is a DMA Transaction with the next index number. The DMA executes sequence of tasks defined in the database and then proceeds to process the next transaction.

A_Task_Name	A_Instruction	A_IDX	A_Task_Source	Burst_Word_Size	A_Task_Address
MyTask	MOV	0	SDRAM_1	8	3
MyTask	MOV	1	SDRAM_1	8	3

A_Command	A_Bytes	A_Priority	A_Destination
Read	100	0	DMA_1
Read	100	0	DMA_1

After executing the task “MyTask” with A_IDX as 0, DMA looks in the database if there is a task in sequence with A_IDX as 1. After executing the tasks in sequence, the block sends out the DS through the Dout port.

The figure below illustrates the DMA sequence of tasks in Xilinx Memory Controller.

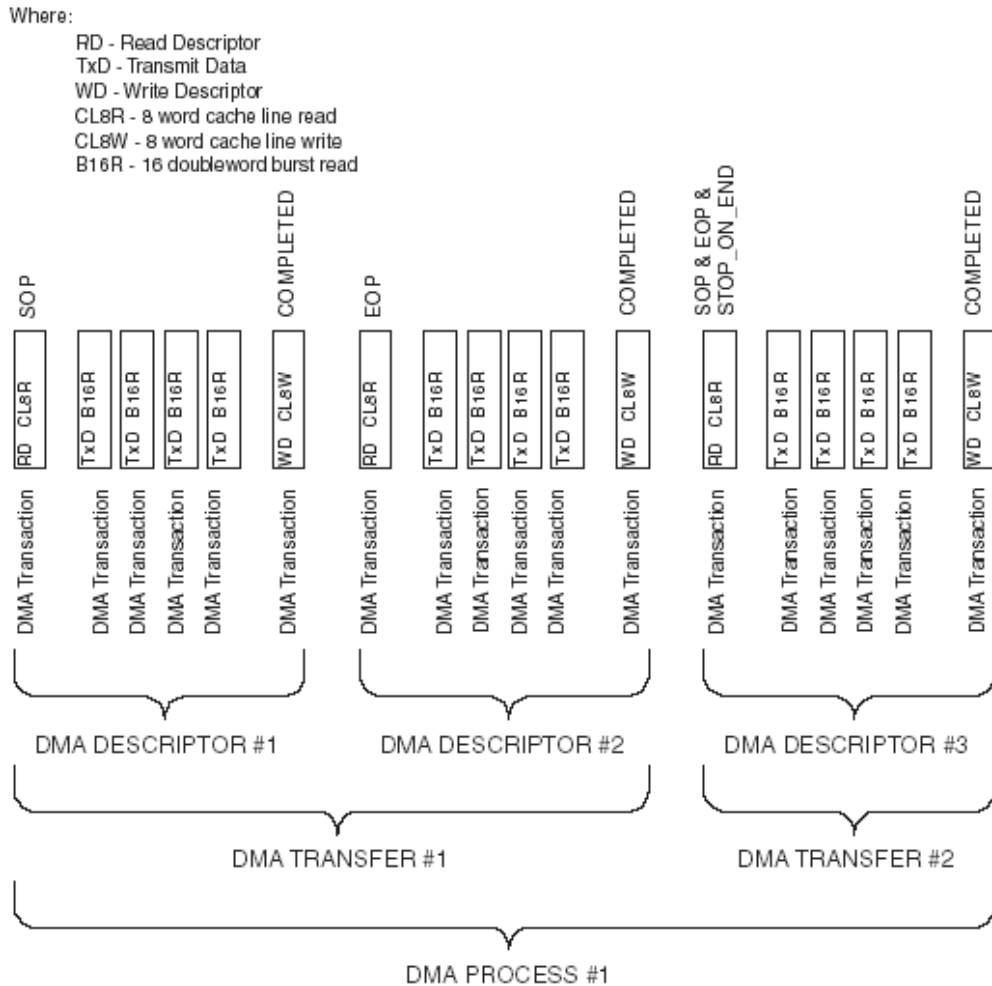


Figure 62 CDMAC Illustration of Tx Engine Flow

Routing Functionality

If the DMA block is connected to a Linear Bus or another bus that adds devices to the Routing Table based on the Hello Messages, no additional entry is required for Bus connectivity. A line is

required for the device connected to the ack port of the DMA. This is added to the Architecture_Setup routing table as follows:

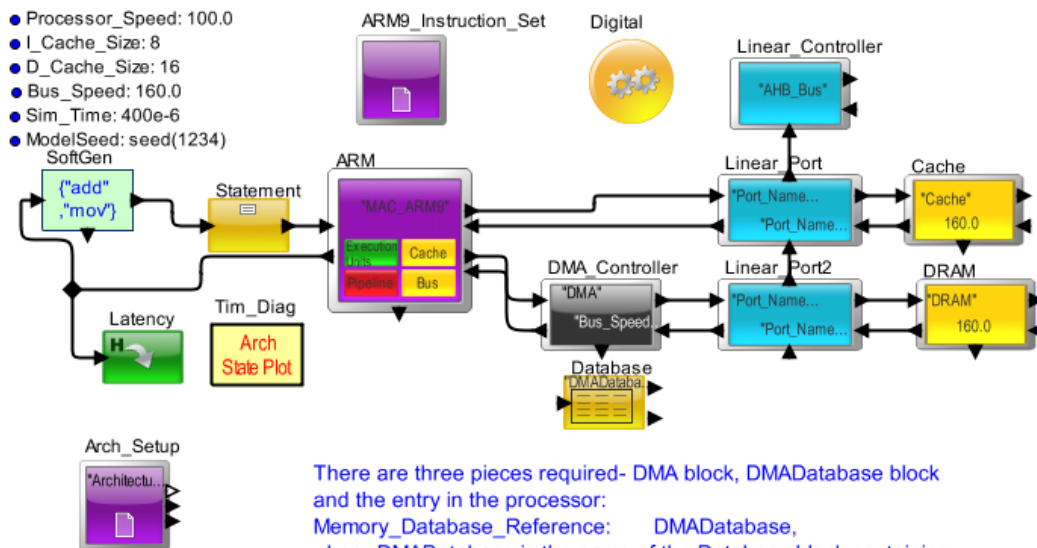
```

/* First row contains Column Names.          */
Source_Node  Destination_Node  Hop      Source_Port ;
DMA          Trigger           DMA      Dout       ;

```

Example of DMA in a Model

Processor- DMA to DRAM



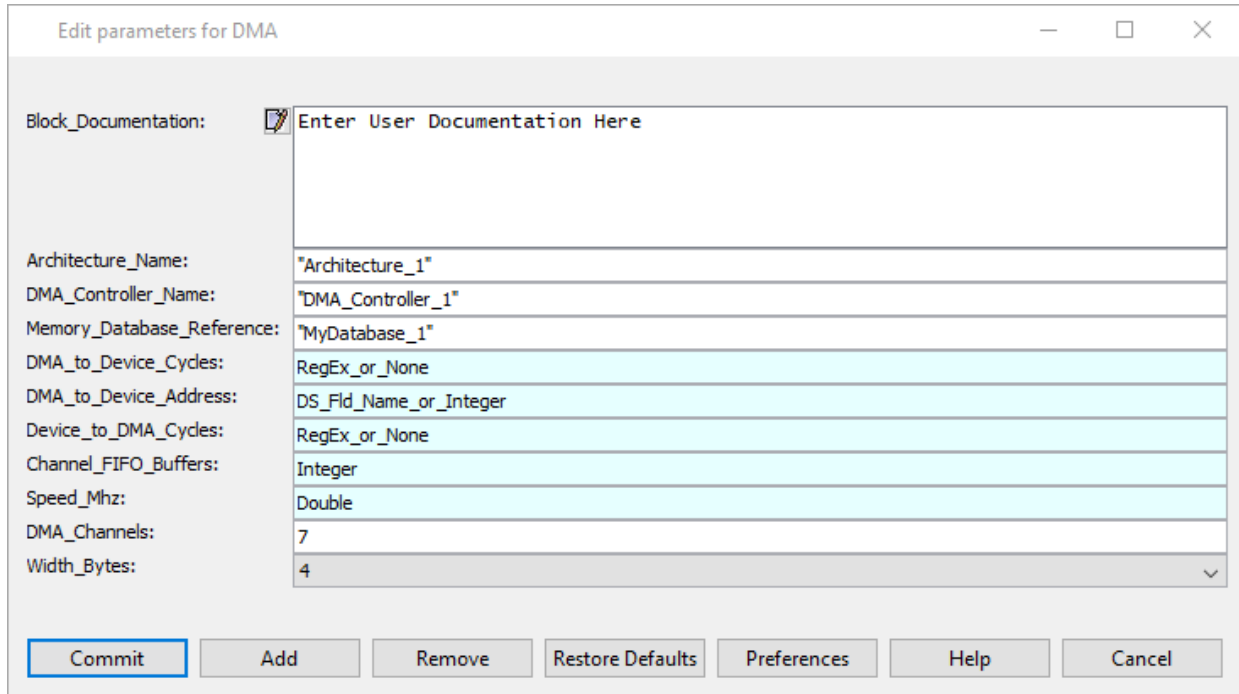
There are three pieces required- DMA block, DMADatabase block and the entry in the processor:
 Memory_Database_Reference: DMADatabase,
 where DMADatabase is the name of the Database block containing the DMA Task activity
 it is best to connect the DMA to one of the ports of the processor either directly or via a Bus.

Figure 63 Using DMA in a Model

Statistics

IO_per_sec: Input and output transactions per second.

Parameters



Block_Documentation:	<input checked="" type="checkbox"/> Enter User Documentation Here
Architecture_Name:	"Architecture_1"
DMA_Controller_Name:	"DMA_Controller_1"
Memory_Database_Reference:	"MyDatabase_1"
DMA_to_Device_Cycles:	RegEx_or_None
DMA_to_Device_Address:	DS_Fld_Name_or_Integer
Device_to_DMA_Cycles:	RegEx_or_None
Channel_FIFO_Buffers:	Integer
Speed_Mhz:	Double
DMA_Channels:	7
Width_Bytes:	4

Buttons: Commit, Add, Remove, Restore Defaults, Preferences, Help, Cancel

Figure 64 DMA_Controller Configuration Window

Architecture_Name

Name of the common architecture to which blocks in the model belong to, Type is String. There can be different architectures existing; hence the unique Architecture name is defined with each block.

DMA_Controller_Name

Name of the DMA_Controller block, Type is String. The block name has to be unique within the same architecture. Another architecture can exist with the same block name. The name identifies the unique block instances.

Memory_Database_Reference

Name of the Database block that has the DMA setup details. Type is String. The specified Database block has to exist in the model with either a reference to the Database configuration file (*.csv) or defined in the Data Structure Text field of the Database block.

DMA_to_Device_Cycles

DMA_to_Device_Cycles is the Cycles taken to send data from DMA_Controller to memory. Type is integer.

The parameter value can be either a RegEx_or_Integer.

If a RegEx is specified the result of the regular expression is used as parameter value.

An integer value- number of cycles can also be specified like 100.

DMA_to_Device_Address

DMA_to_Device_Address is the Address of Memory or Device to which the DMA_Controller routes a transaction. Type is integer.

The parameter value can be either a DS_Fld_Name_or_Integer. If a Data Structure field name is specified the incoming Data Structure's field value will be used as parameter value;example:A_Address_Min. An integer value for the address can also be specified like 101.

Device_to_DMA_Cycles

Device_to_DMA_Cycles is the Cycles taken to send data from memory to DMA_Controller. Type is integer.

The parameter value can be either a RegEx_or_Integer.

If a RegEx is specified the result of the regular expression is used as parameter value.

An integer value- number of cycles can also be specified like 100.

Channel_FIFO_Buffers

It is the Length of the FIFO buffer of each channel; Length of both the Input and Output queues that hold the transactions flowing in and out of the block. Type is integer.

An integer value for the size of FIFO buffer is specified like 20.

Speed_Mhz

Speed of the DMA_Controller block in Mega hertz, Type is double. This is the rate at which the DMA Controller block processes transactions. Speed is used to calculate the DMA Controller cycle time; DMA Controller cycle time = $1.0E-06 / \text{Speed in Mhz}$.

Burst_Size_Bytes

The burst size of DMA transfer in bytes. A High speed channel would transfer entire data completely in a transaction as a burst transfer.

DMA_Channels

The number of channels in the DMA

Width_Bytes

The size of DMA transfers in bytes. A slow speed channel would transfer word by word. The number of words to be transferred depends on the total bytes/width bytes.

Request Acknowledge Node/ Asynchronous Bus

Introduction

The **Request Acknowledge Node** (Req_Ack_Node) block is a high performance, hardware level, asynchronous bus for interconnecting processor, memory subsystems, and high bandwidth peripherals. This block was used in creating the CoreConnect Bus, for example. The Req_Ack_Node can be configured as a Master, Controller, or Slave by the block pulldown parameter named Node_Type, as shown below.

IBM CoreConnect Bus Model

Parameters.

- Architecture_Name: "Architecture_1"
- Bus_Name: "Bus_1"
- Architecture_Bus_Name: Architecture_Name + "_" + Bus_Name
- CoreConnect_Speed_Mhz: 1000.0
- Burst_Size_Bytes: 64
- FIFO_Buffers: 32
- Request_Clock_Multiplier: 0.0

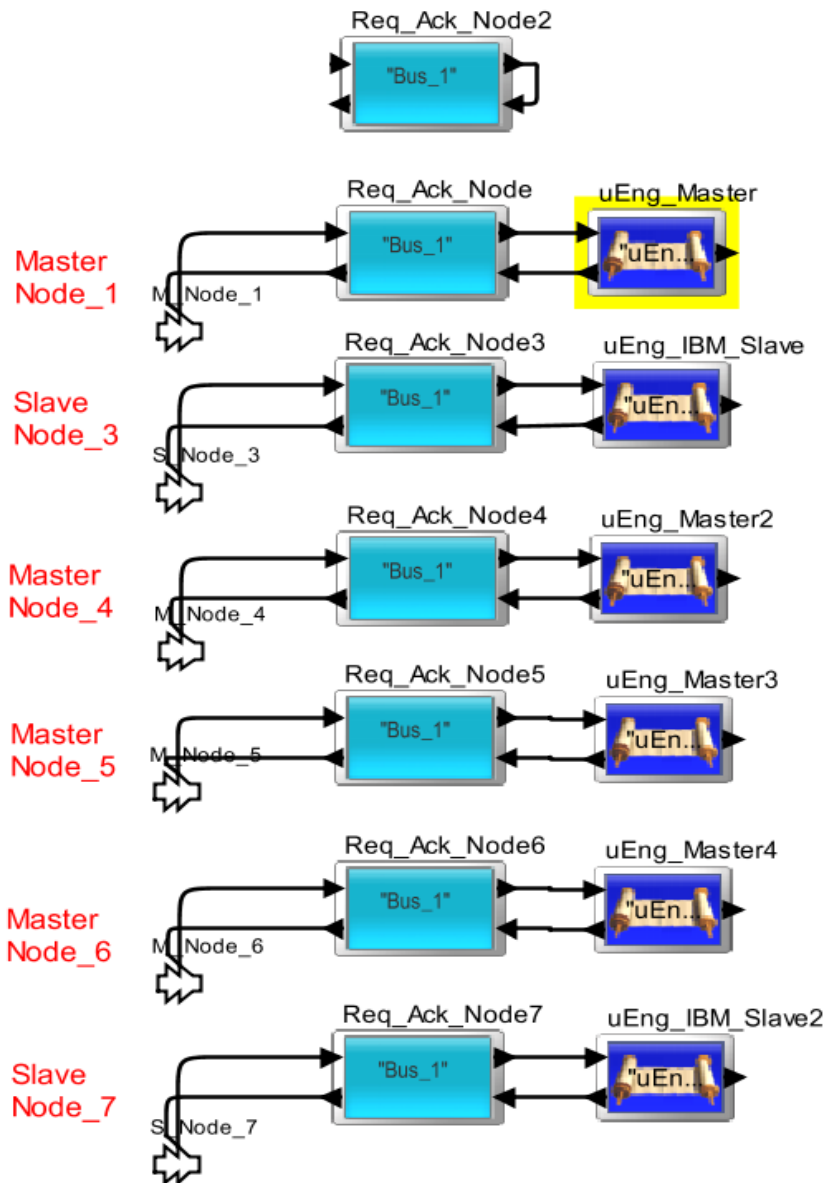


Figure 65 Master, Slave and Controller Block Connection

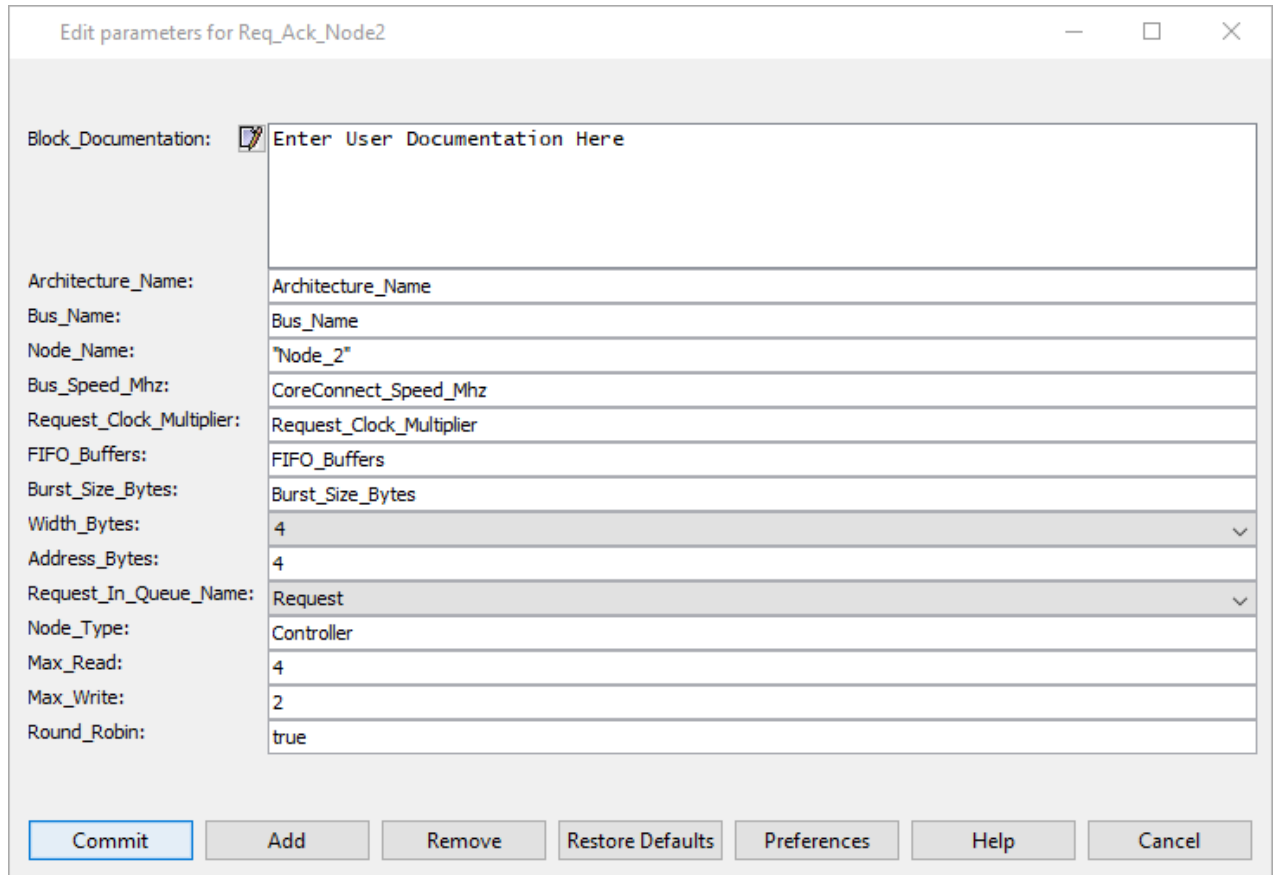


Figure 66 Req_Ack Node View

Block Configure Parameters

The Req_Ack_Node configure parameter settings:

- Architecture_Name: "Architecture_1" Unique Name of Architecture
- Bus_Name: "Bus_1" Unique Name of Bus
- Node_Name: "Node_1" Unique Node Name
- Bus_Speed_Mhz: 33 Bus Speed in Mhz
- Request_Clock_Multiplier: 0.99 Speed of Request Channel
Fraction of Clock Time, can be zero
- FIFO_Buffers Number of Transactions that
can be stored in Port Buffer
- Burst_Size_Bytes Largest Byte transfer
over the bus.
- Width_Bytes Width of Bus Channels in Bytes
- Address_Bytes Address Bytes can be larger than
Width_Bytes for 64 bit words on
32 bit bus width, for example.
- Request_In_Queue_Name Name of Queue where incoming
Requests are placed, default Request
Channel.
- Node_Type Master, Controller, or Slave
- Max_Read User added for maximum queue

- Max_Write
- Round_Robin

Size on Controller Node, user must add this parameter to make it valid.
 User added for maximum queue Size on Controller Node, user must add this parameter to make it valid.
 true or false to select the next master Selected. If true, the controller will Direct a new request to the oldest Request pending.

Data Structure: Processor_DS

The Processor_DS was selected as the best data structure to send through the Req_Ack_Node block. This data structure is used by the Processor block and other Architectural Library busses. The following are specific data structure fields to be set prior to entering a master port.

- A_Bytes* – Total number of bytes to be transferred
- A_Bytes_Remaining* – A_Bytes (total) minus A_Bytes_Sent (bus width)
- A_Bytes_Sent* – bus width
- A_Command* – Req_Ack_Node bus commands, see below:

<u>Req_Ack_Node Bus Commands</u>	<u>A_Command field</u>
IO Read	“Read_IO”
IO Write	“Write_IO”
Memory Read	“Read_Memory”
Memory Write	“Write_Memory”

Any Read or Write command whether it is from an IO device or Memory is handled by the bus in a similar fashion. Hence the bus looks for A_Command that starts with the “Read_*” string to process a Read transaction and starts with the “Write_*” string to process a Write transaction. Read commands will return to the source node, Write commands will execute on the slave device and “not” return. One exception is if the slave device is a Processor block.

- A_First_Word* – default is true, not used by the Req_Ack_Node block.
- A_Priority* – the priority of the transaction, higher priority gains bus access.
- A_Source* – routing source node, or transaction initiator.
- A_Hop* – represents the next node in the routing path external to the Master Slave ports, and internal to block it is used to pass a message from the Ack port output to the Din input port. Internal use as a block level command.
- A_Status* – used internally by the Req_Ack_Node block to designate the channel of the data structure: either Request, Address, Read, or Write .
- A_Destination* – routing destination node, or transaction target.

The Req_Ack_Node has fields that it adds to the Processor_DS for internal routing, and identification.

- A_Bus_Source* – internal Req_Ack_Node bus source name.
- A_Bus_Destination* – internal Req_Ack_Node bus destination name



- A_Bus_ID* – internal Req_Ack_Node bus transaction ID, unique.
- A_Bus_Index* – internal Req_Ack_Node integer field indicating same as *A_Status*.

A_Bus_ID is used to keep transactions uniquely identified when removing requests from the Master, Controller, and Slave blocks. A sample set of data, a user can set in fields of data structure "Processor_DS":

Data Structure Field	Data Type	Sample Data
<i>A_Bytes</i>	int	64
<i>A_Command</i>	string	"Read_Memory"
<i>A_First_Word</i>	boolean	true
<i>A_Priority</i>	int	1
<i>A_Source</i>	string	"IO_1"
<i>A_Destination</i>	string	"SDRAM_1"

Acknowledge Port to Data-In Port Block Commands

Here is a list of block commands supported by the Master. The internal block command is set in the "A_Hop" field of the Processor_DS:

<u>Master Commands</u>	<u>Action</u>
"send_to_a_destination"	Send to A_Bus_Destination, Master or Slave
"send_to_a_source"	Send to A_Bus_Source, Master or Slave
"send_queue_element"	Send DS to Dout Port of Req_Ack_Node
"drop_queue_element"	Drop DS, no further action
"enqueue"	Put DS into Channel, based on A_Status field
"rearbitrate"	Reprocess DS back to source

Here is a list of block commands supported by the Controller. The internal block command is set in the "A_Hop" field of the Processor_DS:

<u>Controller Commands</u>	<u>Action</u>
"send_to_a_destination"	Send to A_Bus_Destination, Master or Slave
"send_to_a_source"	Send to A_Bus_Source, Master or Slave

Here is a list of block commands supported by the Slave internal block command is set in the "A_Hop" field of the Processor_DS:

<u>Slave Commands</u>	<u>Action</u>
"send_to_a_destination"	Send to A_Bus_Destination, Master or Slave
"send_to_a_source"	Send to A_Bus_Source, Master or Slave
"send_queue_element"	Send DS to Dout Port of Req_Ack_Node
"drop_queue_element"	Drop DS, no further action



"enqueue"	Put DS into Channel, based on A_Status field
"rearbitrate"	Reprocess DS back to source

Routing Table

The Req_Ack_Node supports the Architectural Library auto-routing capability for busses. If the transactions must traverse multiple busses, or one wishes to over-ride the auto-routing, then additions can be made to the routing table:

```

/* First row contains Column Names.                                     */
Source_Node Destination_Node Hop Source_Port ;
IO_1         SDRAM_1         Node_1      to_bus   ;
IO_2         SDRAM_1         Node_3      to_bus   ;
SDRAM_1      IO_1            Node_2      output   ;
SDRAM_1      IO_2            Node_2      output   ;

```

The third column represents the next hop in the routing, which might be the name of the bus port, I_O, Cache, DRAM, IO_Controller, Memory_Controller, or DMA_Controller. The Source_Port is the name of the port the transaction is leaving, and applies if there is more than one output port from a bus or memory block. The Req_Ack_Node Bus has a single input and output per port, so the value of the Source_Port column is not critical to entering routing information. If any routing entries are missing, exceptions are thrown indicating the missing source and destination pair during the model execution.

Bus Statistics

The following statistics are collected in the same Req_Ack_Node bus model.

- **Delay** – Transaction delay
- **IOs_per_sec** – Input Output transactions per sec
- **Node_Buffer_Occupancy_in_Words** – Input buffer occupancy in words
- **Throughput_MBps** – Throughput in Mbps
- **Utilization_Pct** – Utilization percentage

The sample statistics collected in the model are given below in min, mean, stdev, and max values for the Req_Ack_Node bus.

```

{BLOCK = ".Req_Ack_Node_Read_Read_N_Bytes.Architecture_Setup",
Bus_1_Address_Buffer_Occupancy_in_Words_Max      = 1.0,
Bus_1_Address_Buffer_Occupancy_in_Words_Mean     = 1.0,
Bus_1_Address_Buffer_Occupancy_in_Words_Min     = 1.0,
Bus_1_Address_Buffer_Occupancy_in_Words_StDev    = 0.0,
Bus_1_Delay_Max                                  = 8.0E-9,
Bus_1_Delay_Mean                                 = 7.0E-9,
Bus_1_Delay_Min                                  = 4.0E-9,
Bus_1_Delay_StDev                                = 1.4142135623731E-9,
Bus_1_IOs_per_sec_Max                            = 6.0E6,
Bus_1_IOs_per_sec_Mean                           = 4.0E6,
Bus_1_IOs_per_sec_Min                            = 3.0E6,
Bus_1_IOs_per_sec_StDev                          = 1.309307341416E6,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Max = 0.33333333333333,

```

```

Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Mean = 0.33333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Min = 0.33333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Max = 0.33333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Mean = 0.33333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Min = 0.33333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Max = 0.33333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Mean = 0.33333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Min = 0.33333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Max = 0.33333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Mean = 0.33333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Min = 0.33333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Read_Buffer_Occupancy_in_Words_Max = 2.0,
Bus_1_Read_Buffer_Occupancy_in_Words_Mean = 2.0,

```

Bus_1_Read_Buffer_Occupancy_in_Words_Min = 2.0,
Bus_1_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Request_Buffer_Occupancy_in_Words_Max = 4.0,
Bus_1_Request_Buffer_Occupancy_in_Words_Mean = 4.0,
Bus_1_Request_Buffer_Occupancy_in_Words_Min = 4.0,
Bus_1_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Throughput_MBs_Max = 112.0,
Bus_1_Throughput_MBs_Mean = 112.0,
Bus_1_Throughput_MBs_Min = 112.0,
Bus_1_Throughput_MBs_StDev = 0.0,
Bus_1_Utilization_Pct_Max = 5.79,
Bus_1_Utilization_Pct_Mean = 5.79,
Bus_1_Utilization_Pct_Min = 5.79,
Bus_1_Utilization_Pct_StDev = 0.0,

Power Modeling Toolkit

The **VisualSim** Power Manager is the first System-Level Design solution to trade-off performance and power in a single architecture model. This Library is used to evaluate the effectiveness of power scheduling algorithms and to estimate the power consumption of a design. The power manager updates the instantaneous and cumulative power using dynamic state change information of the individual devices. The power manager is fully integrated with the Architecture Library, scheduler blocks and the RegEx language. Function calls are available to dynamically alter the power level of a state or record a state change. The devices analyzed can be standard devices, custom hardware acceleration and software components. The Power Manager library consists of a manager block and utility functions for state change transitions, change in power levels and statistics generation. An initial charge for the Power Manager is specified as a parameter of the block.

Multiple power blocks can be maintained in a single model. Any number of devices can be associated with a single Power Manager block. Each standard device can have up to 4 power states (standby (leakage), idle, active and wait) and a transition cycle time. For custom blocks, there can be up to 12 states defined. These the current state and power in a state can be modified during the simulation using the RegEx functions explained later in this Chapter. The **VisualSim** Power Manager is based on dynamic system operation and will enable users to design application-based power schedulers and make trade-offs between performance and power consumption including battery drain.

The power for each device is maintained individually. When the operation state of a device changes (idle, standby, wait and busy), the power level goes to the new state. There is a delay to go to the new state called transition cycles. The number of cycles in that column is multiplied by the Power_Manager speed (Parameter) to compute the transition delay. The transition cycles can be turned off to disable this performance impact.

Let us take an example. When the Processor is in standby state, the power_manager maintains the current power consumed by this processor to be value in the standby column for this processor. When processor moves to active/busy state, the power_manager delays by the transition cycles and then changes the current value for the processor to the value in the active column. This is an automatic internal process and the user does not have to do anything.

If the user has a hardware accelerator, the model uses the functions listed below update the state change or a new power level.

Introduction

The Power_Manager maintains a list of the devices supported. The Power Manager can read the power level for each state from the text window or a file, at initialize and store the configuration information internally. The file type can be csv or txt. There are two types of supported devices- (1) Hardware library and scheduler blocks and (2) functions. Each referenced block in the Power Manager contains the following;

Block Name	Standby State	Active State	Wait State	Idle State	Transition Cycles
Architecture_Name + " " +	Double	Double	Double	Double	Integer (Based on

Block_Name					Manager speed)
Scheduler + “_” + Block_Name	Double	Double	Double	Double	Integer (Based on Manager speed)

Table 5 : Manager_Setup format

All blocks supporting the Power Manager are considered to be in standby state at the start of a simulation. Some blocks may have only one actual power state, such as Active, whereas others may have two, three or four states. There are four possible power states: standby, active, wait or idle. Custom blocks can have up to 12 columns. For example, the processor maintains 4 states while the cache has only two states. There is also an entry for transition cycles, based on the Power Manager Clock speed, for changing from one state to another, typically related to a natural time constant of the chip, or may be a controlled transition to minimize power spikes. Zero is a legal number of Transition Cycles. Each time there is a state change in a block that supports the Power Manager, the new state is sent to the Power Manager block. The Transition Cycles entry will schedule an event to change the power for the requesting block to the new value, assuming a non-zero entry.

How it works

A Cache block in Standby state gets a request and sets the internal state to BUSY while processing request. At the end of the request, the state is set back to IDLE, if no requests are pending. Cache will send powerUpdate (Arch_1, Cache_1, ACTIVE) when Cache goes busy, and powerUpdate (Arch_1, Cache_1, STANDBY) when the transaction completes.

Assumptions

- Power Update RegEx function can communicate with this block. If the requested device is not found, then ignore the updates.
- Power Change RegEx function allows one to modify the power dynamically during simulation execution to reflect voltage changes on top of state information.
- Power Current or Power Cumulative RegEx functions can obtain all power information from Power Manager.
- Transition Cycles can range from 0 to N.
- Instantaneous power is updated at the end of the Transition Cycles.
- Active State is minimal entry for any referenced block, others optional.
- Power information includes instantaneous power per block, cumulative power consumption per block, total in Battery_Units, average power in the units of the block and total discharge for the Power_Manager.

Block Level Parameters

The block level parameters reflect the different Power Manager features discussed in the introduction, power leakage notwithstanding:

Manager_Name: This is the name of the Power Manager block. This is used to send and receive data from RegEx functions and also to call for advanced plotting.

Manager_Setup: This is a Text Window containing columns for the power states and rows for each device. The current implementation offers four predefined power states. This window can

also be provided the path to a text file or a CSV file. The file can contain the same information. The CSV file does not require a “;” at the end of each line of the file.

Manager_Speed_Mhz: The rate at which the Power Manager is charged.

Battery_Charge_Hr: This is the number of charge units per hour.

Battery_Units: This is a pull-down indicating the units for the charge parameter and the units for the power state value in Manager_setup. The values are Micro_Watts, Milli_Watts and Watts.

Enable_Transition_Cycles: This checkbox turns on and off the Transition cycles.

Block Ports

The Power Manager has two ports: the instantaneous power level after Transition Cycles have completed for individual blocks, and battery charge port, showing the remaining cumulative power in selected units. The output from both ports can be connected to a Timed Plotter block. The instantaneous power level will be in the “Battery_Units of the block while the Cumulative Power will be in Battery_Units per hour.

Block Methods

Six RegEx functions are available for use to update the Power_Manager and to retrieve power information. These RegEx functions can be called from Expression and Script blocks.

Function 1:

powerUpdate(String Manager_Name, String Arch_Block_Name, String Power_State)

Eg: Result_A = powerUpdate(Pwr_Mgr1,ArchSetup_uP, “Active”)

Description: This function updates the current power state of the block at the Power_Manager. The Power_State can accept any name in the Manager Setup table. The Manager_Name must match name in the Power_Manager block while the Arch_Block_Name must match the name (Arch_Lib + Block_Name) in the Manager_Setup field. The three parameters can be string values or indirect references such as memory or field names. If the new state is the same as the existing state, no update is made. If the new state is different, then the instantaneous power and the cumulative power columns are updated. This function does not return any value.

Function 2:

newPowerLevel(String Manager_Name, String Arch_Block_Name, String Power_State, double New_Value)

Eg: Result_A = newPowerLevel(Pwr_Mgr1,ArchSetup_uP, “Active”,0.13)

Description: The newPowerLevel function enables the user to dynamically modify the power level of a particular state during the course of a simulation. For example the Active state power level can be changed, if the chip or board changed the voltage. The function updates the power level in the Manager_Setup column. The Power_State can accept any name in the Manager Setup table. The Manager_Name must match name in the Power_Manager block while the Arch_Block_Name must match the name (Arch_Lib + Block_Name) in the Manager_Setup field. The three parameters can be string values or indirect references such as memory or field names. This block does not return any value.

Function 3:

double powerCurrent(String Manager_Name, String Arch_Block_Name)

Eg: Result_A = powerCurrent(Pwr_Mgr, Scheduler_Sched1)

Description: This RegEx functions returns the instantaneous power as a double value for the selected Block. If the Block_Name= “total”, then the total value for the entire model, i.e. all the devices will be generated. The Manager_Name must match name in the Power_Manager block while the Arch_Block_Name must match the name (Arch_Lib + Block_Name) in the Manager_Setup field. The two parameters can be string values or a memory or field names that contains this name.

Function 4:

double powerCumulative(String Manager_Name, String Arch_Block_Name)

Eg: Result_A = powerCumulative(Pwr_Mgr, ArchSetup_DMA1)

Description: This RegEx functions returns the cumulative power consumed as a double value for the selected Architecture Block. If the Block_Name= "total", then the total value for the entire model, i.e. all the devices will be generated. The Manager_Name must match name in the Power_Manager block while the Arch_Block_Name must match the name (Arch_Lib + Block_Name) in the Manager_Setup field. The two parameters can be string values or a memory or field names that contains this name.

Function 5:

double addBatteryCharge(String Manager_Name, Double new_battery_charge_)

Eg: Result_A = addBatteryCharge(Pwr_Mgr, 1.0)

Description: This RegEx functions adds additional battery charge during the simulation. This value will be added to the existing battery charge. The Manager_Name must match name in the Power_Manager block while the new battery charge must be positive double value. A double negative value will decrease the battery charge by that amount. The two parameters can be string values or a memory or field names that contains this name. This returns the new charge value as a type double.

Function 6:

double getBatteryCharge(String Manager_Name)

Eg: Result_A = getBatteryCharge("Pwr_Mgr")

Description: This RegEx functions gets the current battery charge during the simulation. The Manager_Name must match name in the Power_Manager block. The parameter can be string values or a memory or field names that contains this name. This returns the current charge value as a type double.

Function 7:

Data_Structure powerManager(String Manager_Name)

Eg: Result_A = (powerManager("Manager_1").Architecture_1_AHB_Bus).Active

Description: This function extracts the current power table. You can then use the results of this function to extract the power level of a state for an individual device. The return is a data structure. The example shows how to return the particular double value. The powerManager function returns the following:

```
Architecture_1_AHB_Bus           = {Active = 0.1, Time = 2.517059E-4, Wait = 0.0,
Architecture_Block = "Architecture_1_AHB_Bus", Standby = 0.025, Cumulative = 2.42591E-5,
Idle = 0.0, Current = 0.025, Cycles = 1},
Architecture_1_DMA               = {Active = 0.15, Time = 0.0, Wait = 0.0, Architecture_Block =
"Architecture_1_DMA", Standby = 0.05, Cumulative = 0.0, Idle = 0.0, Current = 0.05, Cycles = 1},
Architecture_1_DRAM              = {Active = 0.15, Time = 2.517379E-4, Wait = 0.0,
Architecture_Block = "Architecture_1_DRAM", Standby = 0.05, Cumulative = 3.5082705E-5, Idle
= 0.0, Current = 0.05, Cycles = 1},
Architecture_1_MAC_ARM9          = {Active = 0.2, Time = 2.518719E-4, Wait = 0.1,
Architecture_Block = "Architecture_1_MAC_ARM9", Standby = 0.075, Cumulative =
4.6035870000002E-5, Idle = 0.1, Current = 0.1, Cycles = 1},
total                            = {Time = 0.0, Active = 0.0, Wait = 0.0, Architecture_Block =
"total", Standby = 0.0, Cumulative = 1.05377675E-4, Idle = 0.0, Current = 0.225, Cycles = 0}
```

Function 8:

Double powerUpdateN(String power_manager_name, String block_name, String power_state, integer Queue_Number)

Eg: Result_A = powerUpdateN("Manager_1", Name, "Wait", input.A_Queue)

Description: This function updates the current power state of a particular Queue of the Smart_Timed_Resource block. The return is the new power value in Watts for the Queue.

Power Utilities

The Packaged Power Model is a self-contained modeling environment with a complete model-based power solution with traffic profile, power generator, hardware power, software power, state-based power finite state machines, and pre-defined modeling reports. Traffic Profile can be constant, variable, or trace set by parameters or spreadsheet *.csv files. Power Generator can be constant, variable, trace, or time-based (solar, etc.) set by parameters or spreadsheet *.csv files. Hardware and software power profiles can be analyzed independently. One unique feature is the ability to obtain the power consumption and end-to-end latency for a set of software tasks. The power FSM supports Active, Standby, Suspend, Off states with separate timers for Suspend and Off states. If the provided power FSM does not match the actual system operation, users can modify the finite state machine.

Next, the traffic profile, resources+FSM, hardware/software execution, power generator, and report details are described.

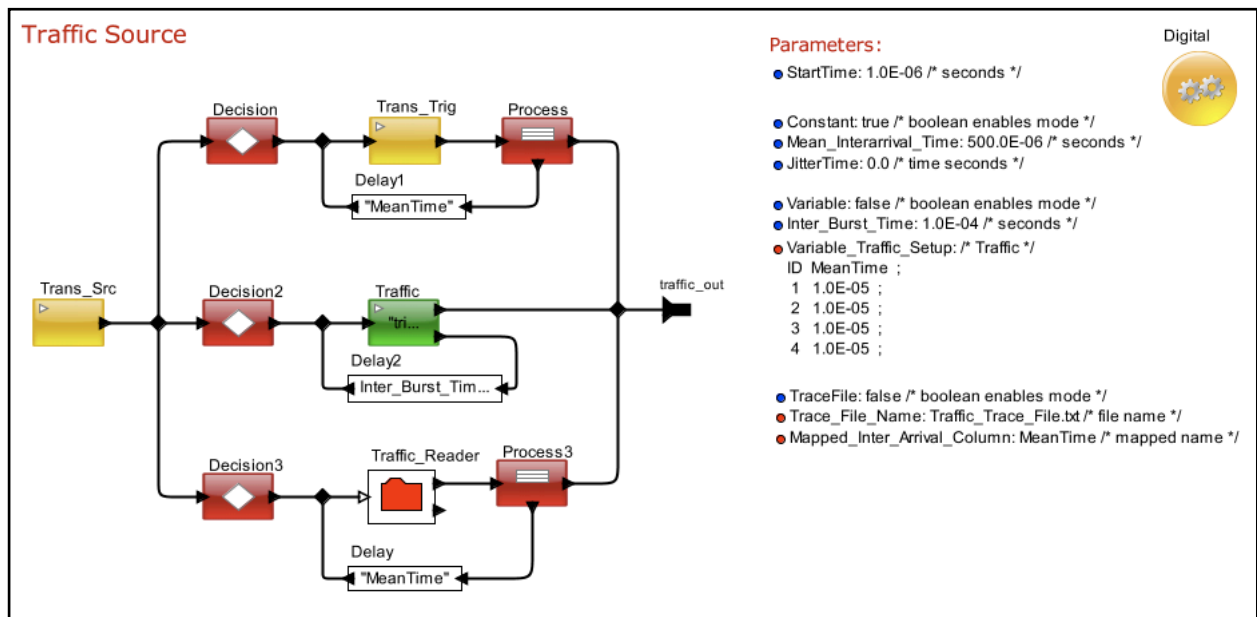
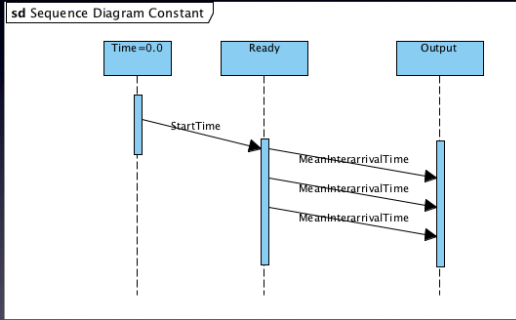


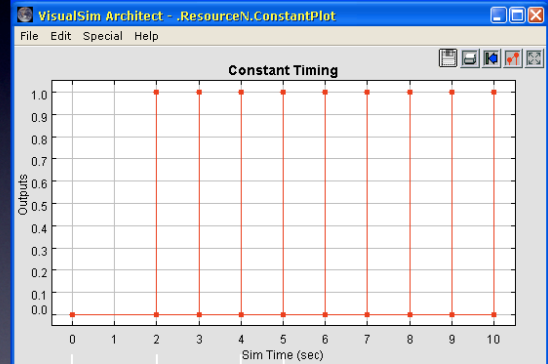
Figure 1. Traffic Profile Block Diagram

Power Traffic: Constant Mode

UML Sequence Diagram View



Timing Diagram View

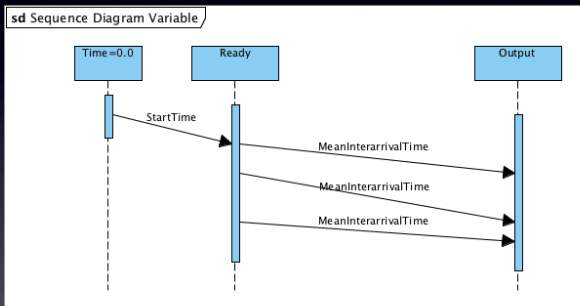


StartTime = 2.0
Mean_Interarrival_Time = 1.0

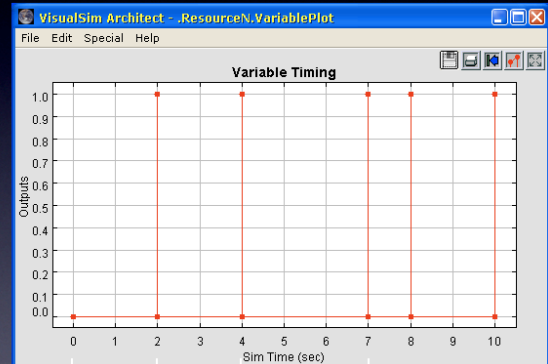
Figure 2. Traffic Profile Constant Mode

Power Traffic: Variable Mode

UML Sequence Diagram View



Timing Diagram View



StartTime = 2.0
Mean_Interarrival_Time = 3.0

Figure 3. Traffic Profile Variable Mode

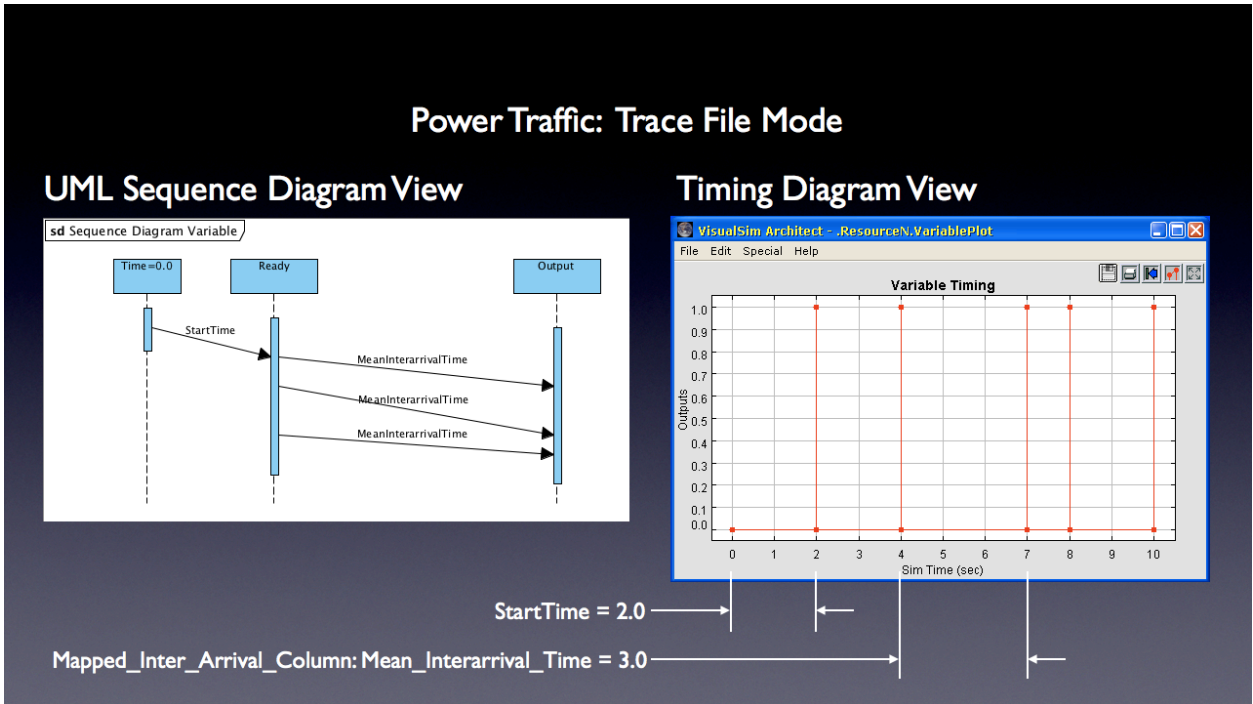


Figure 4. Traffic Profile Trace Mode

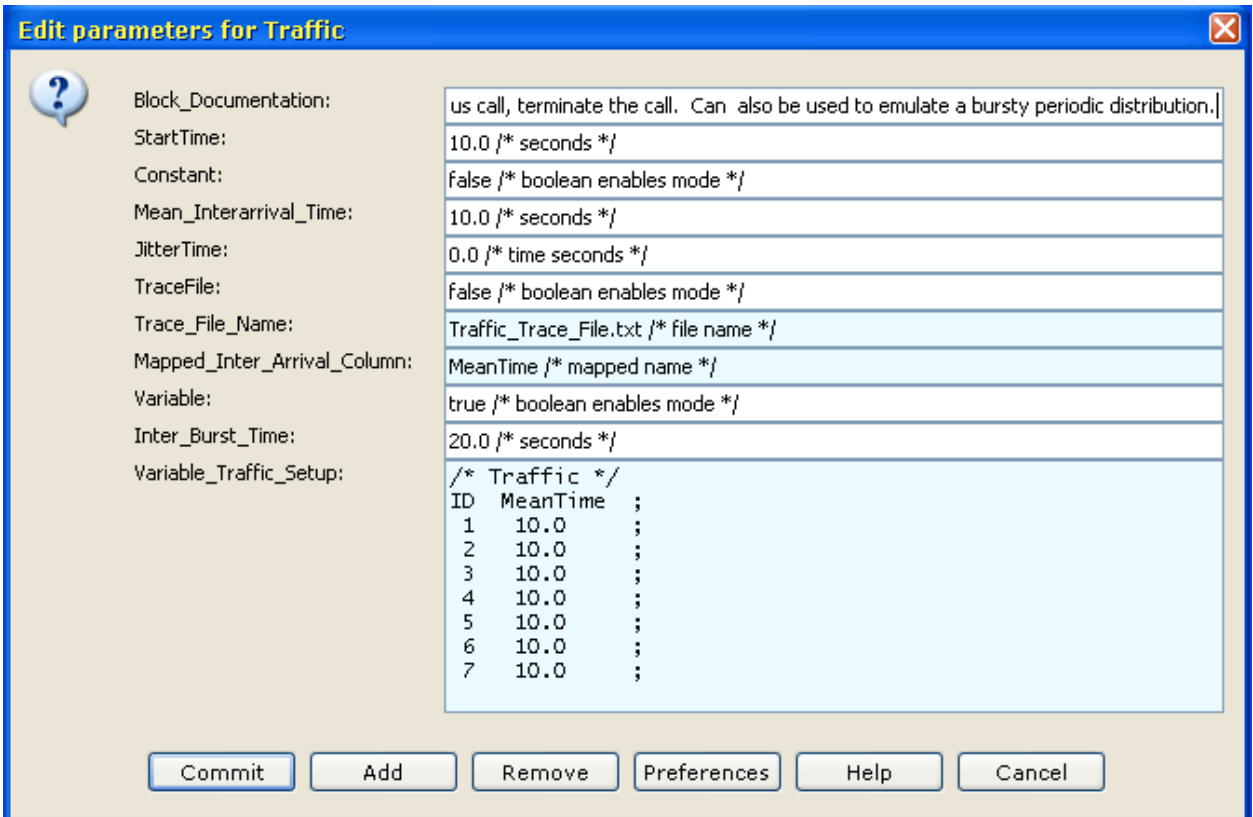


Figure 5. Traffic Profile Parameters

Traffic Profile Parameters

Name	Type	Description
Constant	boolean	mode of operation
StartTime	double	first data structure
Mean_Interarrival_Time	double	time between data structures
JitterTime	double	add jitter to Mean_Interarrival_Time (default 0.0)
Variable	boolean	mode of operation
StartTime	double	first data structure
VariableTrafficSetup	text	ID and MeanTime columns
Inter_Burst_Time	double	time between Variable_Traffic_Setup
Trace	boolean	mode of operation
StartTime	double	first data structure
Trace_File_Name	string	file name
Mapped_Inter_Arrival_Column	string	this is the Mean_Interarrival_Time in the trace file, i.e. time between data structures

Note: More than one Mode can be enabled for complex traffic.

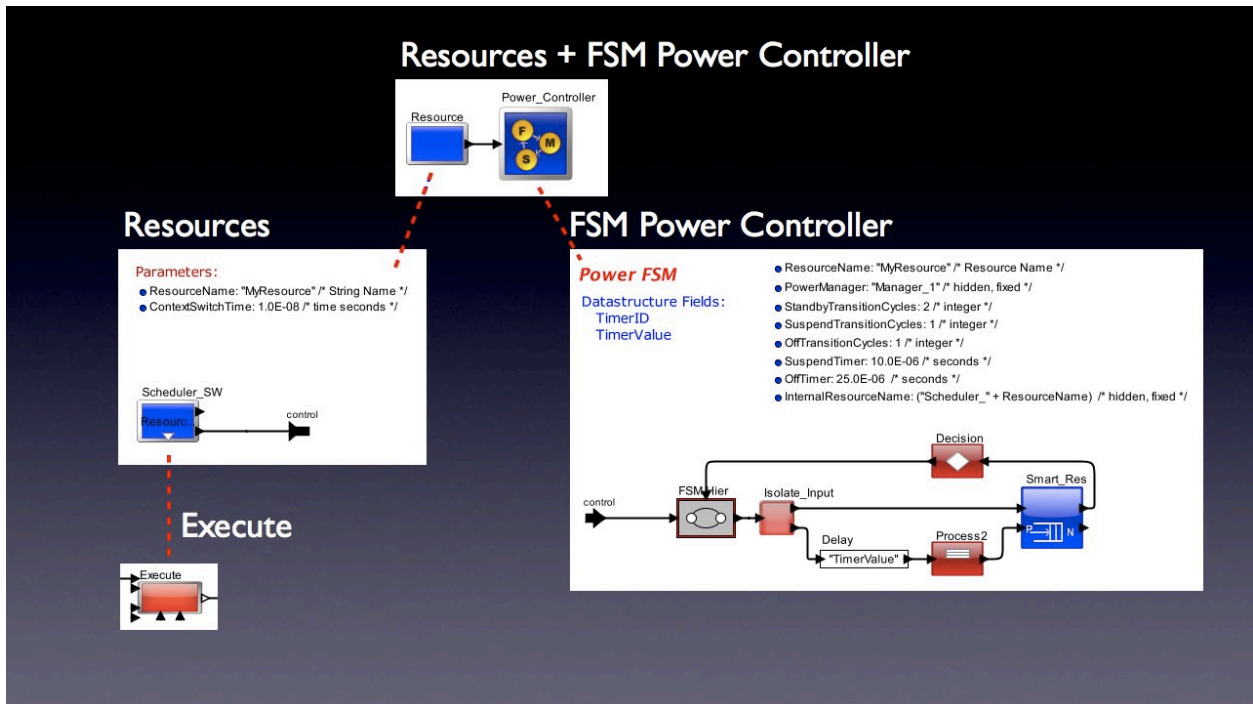


Figure 6. Resources Modal FSMs Block Diagram

Power Finite State Machine: Active, Standby, Suspend, Off States

- Startup: **init** to **off** transition. Sets resource to OffTransitionCycles for first transaction. Power State = **off**
- First Transaction: **off** to **active** transition, based on OffTransitionCycles. Power State = **active**
- First Transaction Complete: **active** to **standby** transition, based on resource completing. Power State = **standby**
- Standby Timer expires: **standby** to **suspend** transition, else return to active, based on StandbyCycles. Power State = **suspend**
- Suspend Time expires: **suspend** to **off** transition, else return to active, based on SuspendCycles. Power State = **off**

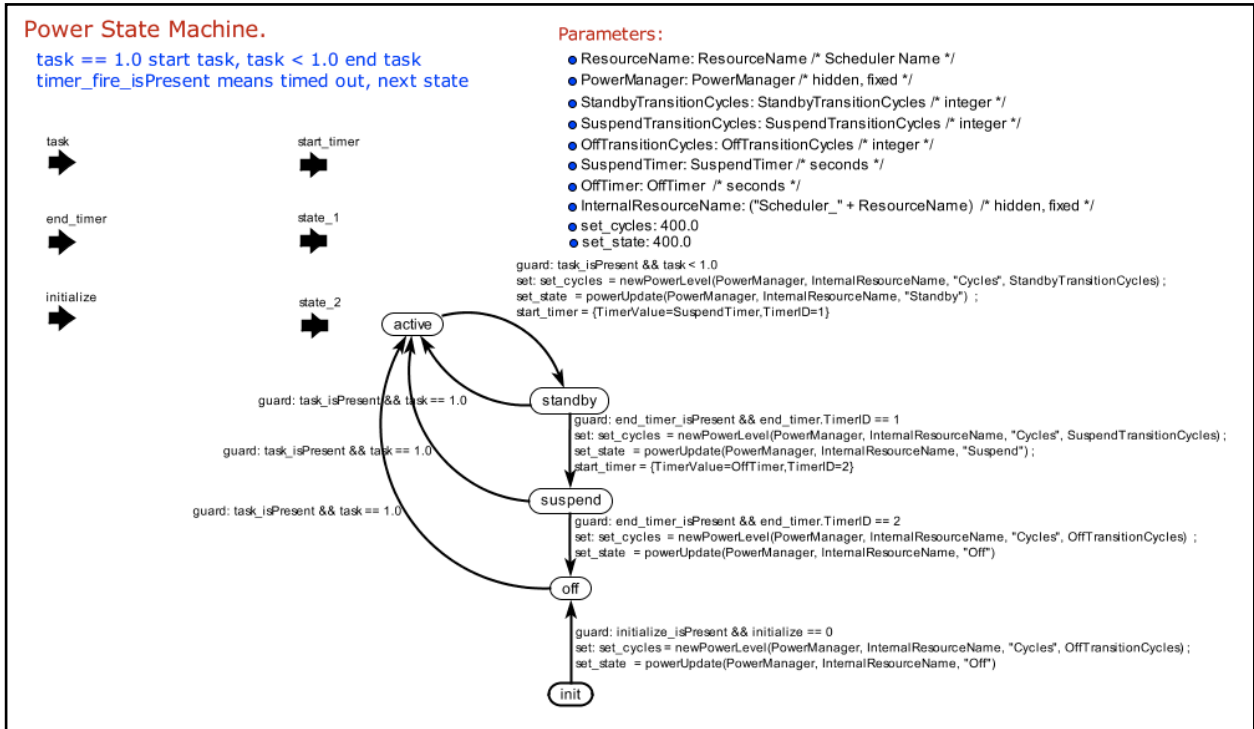


Figure 7. Power FSM States and Transitions

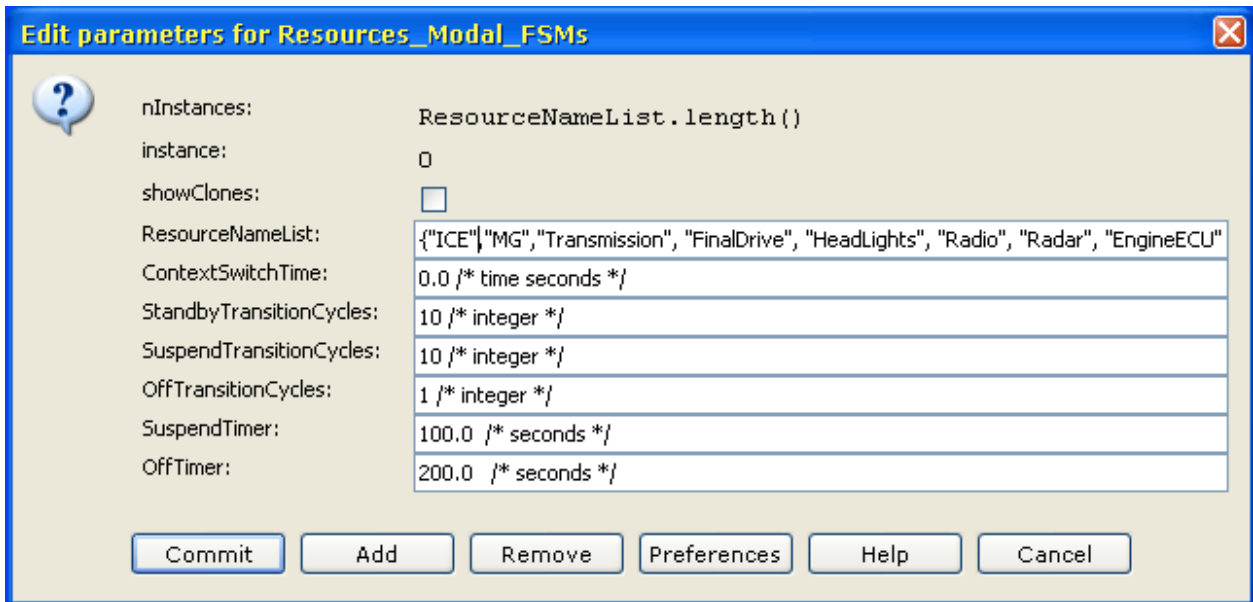


Figure 8. Resources Modal FSMs Parameters

Resources Modal FSMs Parameters

Name	Type	Description
ResourceNameList	array	list of resources with individual power states: Active, Standby, Suspend, Off
ContextSwitchTime	double	time to switch between resources
SuspendTransitionCycles	integer	cycles between power states Suspend and Active
OffTransitionCycles	integer	cycles between power states Off and Active
SuspendTimer	double	If expires, then resource goes to Suspend state
OffTimer	double	If expires, then resource goes to Off state

Note: ResourceNameList must match Execute references.

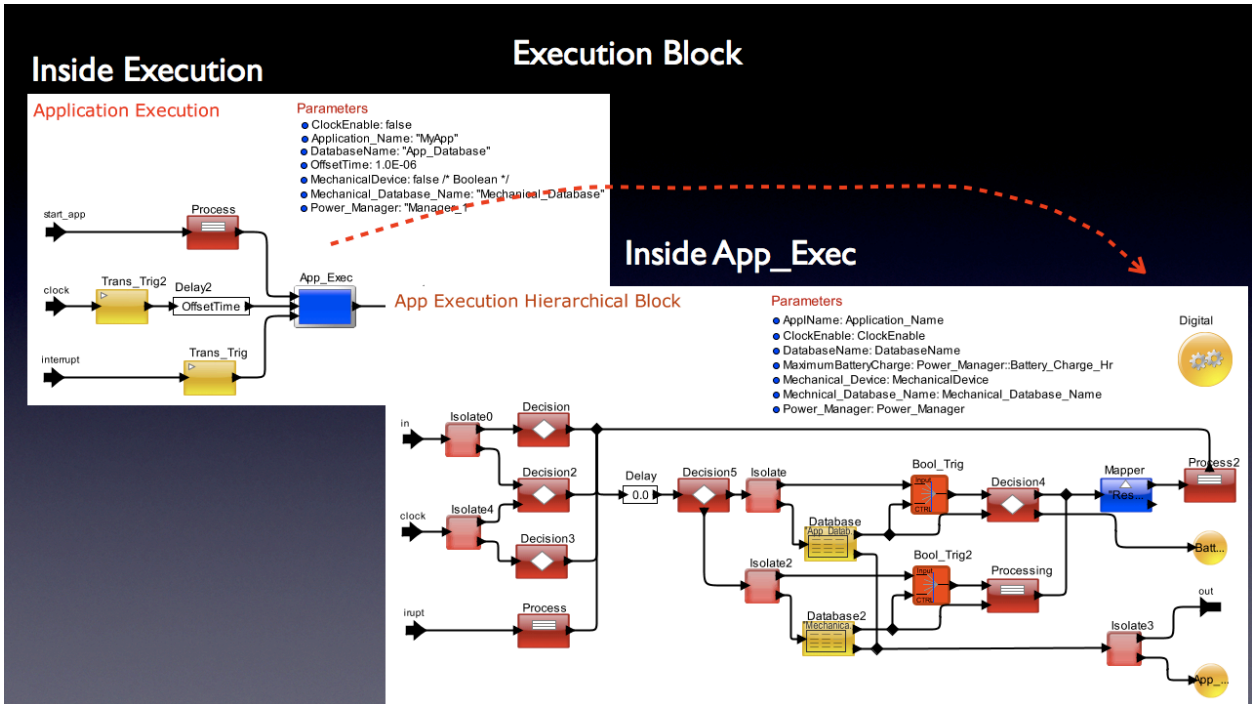


Figure 9. Execution Block Diagram

Figure 10. Execution Parameters

Execution Parameters

Name	Type	Description
Application_Name	string	resource name to execute. Must match existing resource name
ClockEnable	boolean	boolean to enable clock port
MechanicalDevice	boolean	boolean to indicate this is a mechanical device
OffsetTime	double	time to offset or delay clock to trigger execution
PowerManager, DatabaseName, Mechanical_Database_Name	string	fixed, pre-set value

Note: Execute must match ResourceNameList references.

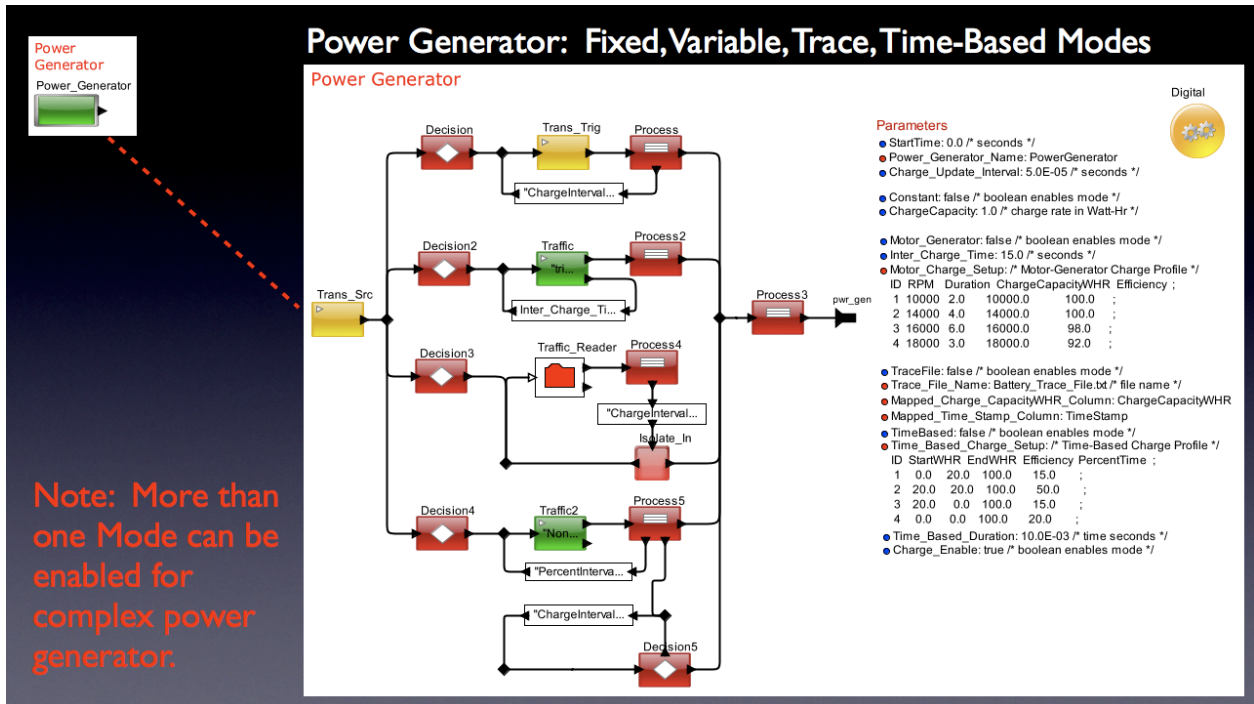
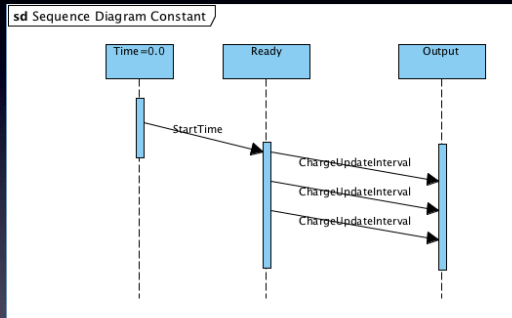


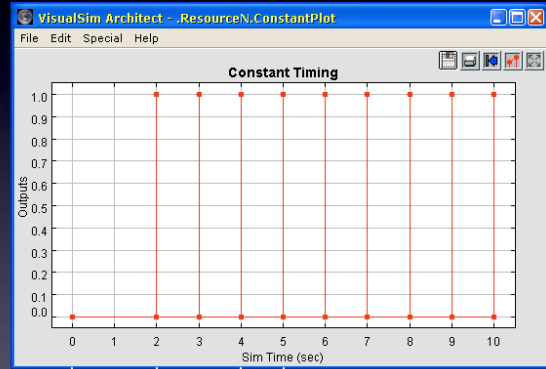
Figure 11. Power Generator Block Diagram

Power Generator: Constant Mode

UML Sequence Diagram View



Timing Diagram View



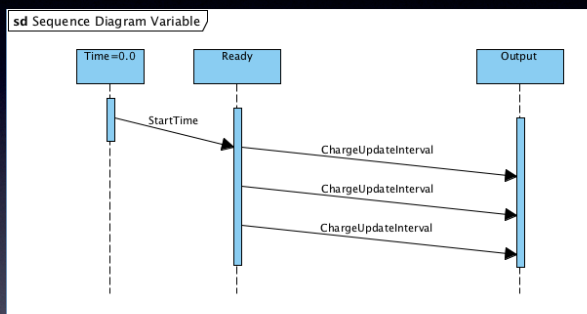
StartTime = 2.0

ChargeUpdateInterval = 1.0

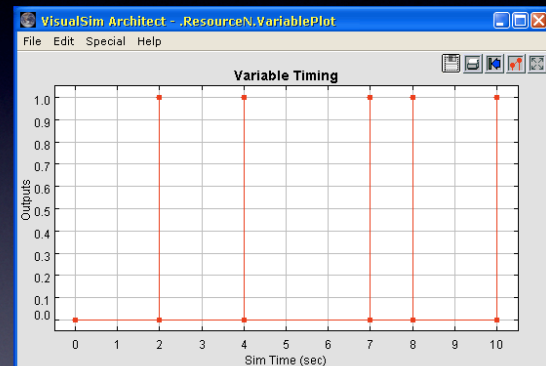
Figure 12. Power Generator Constant Mode

Power Generator: Variable Mode

UML Sequence Diagram View



Timing Diagram View



StartTime = 2.0

ChargeUpdateInterval = 3.0

Figure 13. Power Generator Variable Mode

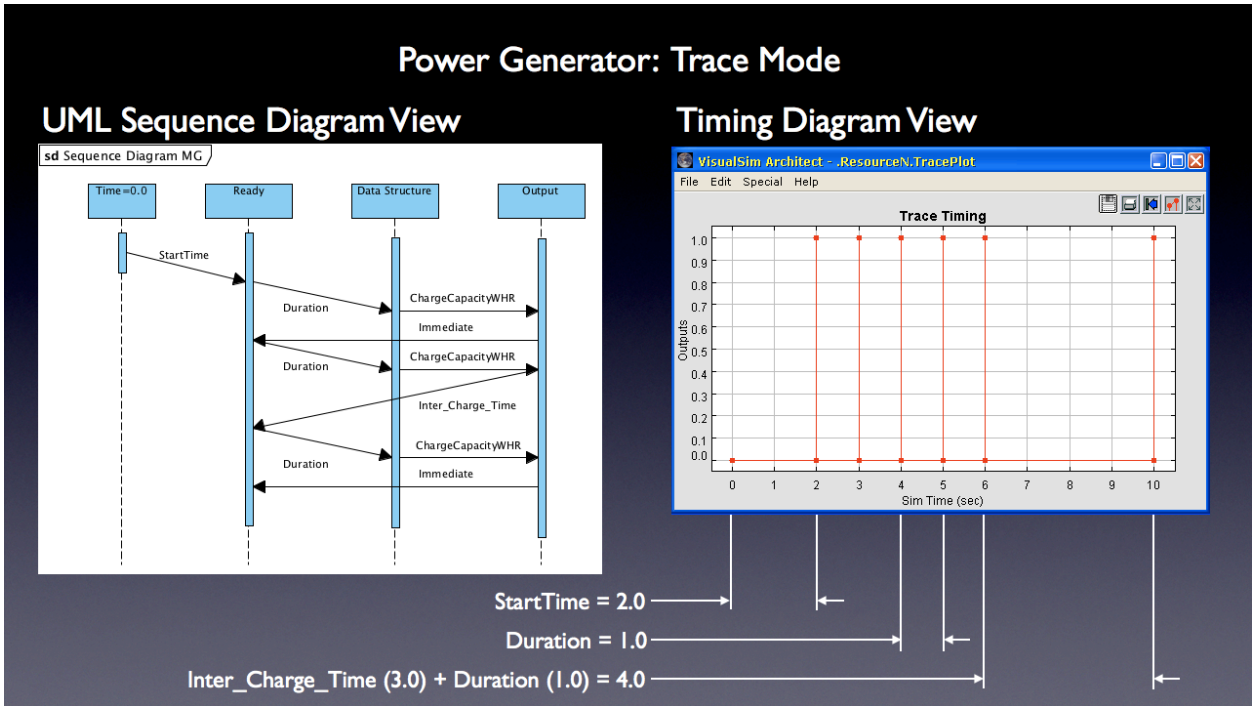


Figure 14. Power Generator Trace Mode

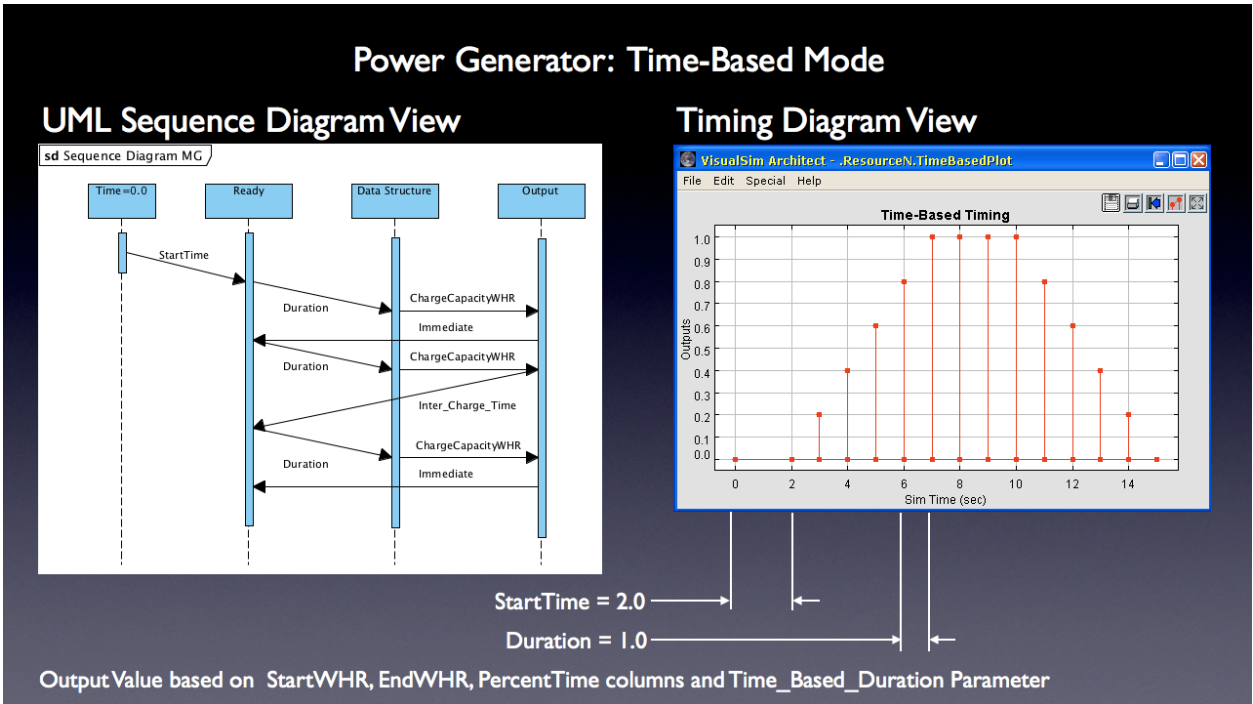


Figure 15. Power Generator Time-Based Mode

Edit parameters for Power_Gen

Block_Documentation: :e. Like regenerative in Hybrids Solar- Arrays in Satellites. This has a specific patterns

Power_Generator_Name: PowerGenerator

StartTime: 0.0 /* seconds */

Charge_Update_Interval: 5.0E-05 /* seconds */

Constant: false /* boolean enables mode */

ChargeCapacity: 1.0 /* charge rate in Watt-Hr */

TraceFile: false /* boolean enables mode */

Trace_File_Name: alSim/V5800/demo/Power/P_HEV_Power_Model/Battery_Trace_File.txt /* file name */

Mapped_Charge_CapacityWHR_Column: ChargeCapacityWHR

Mapped_Time_Stamp_Column: TimeStamp

Motor_Generator: true /* boolean enables mode */

Motor_Charge_Setup:

/* Motor-Generator Charge Profile */					
ID	RPM	Duration	ChargeCapacityWHR	Efficiency	
1	0	10.0	0.0	100.0	;
2	2500	10.0	55.0	100.0	;
3	2500	10.0	55.0	100.0	;
4	4000	10.0	105.0	100.0	;
5	4500	10.0	110.0	100.0	;
6	4000	10.0	105.0	100.0	;
7	2500	10.0	55.0	100.0	;
8	2500	10.0	55.0	100.0	;

Inter_Charge_Time: 20.0 /* seconds */

TimeBased: false /* boolean enables mode */

Time_Based_Charge_Setup:

/* Time-Based Charge Profile */				
ID	StartWHR	EndWHR	Efficiency	PercentTime
1	0.0	20.0	100.0	15.0
2	20.0	20.0	100.0	50.0
3	20.0	0.0	100.0	15.0
4	0.0	0.0	100.0	20.0

Time_Based_Duration: 10.0E-03 /* time seconds */

Charge_Enable: true /* boolean enables mode */

Commit Add Remove Preferences Help Cancel

Figure 16. Power Generator Parameters

Power Generator Parameters

Name	Type	Description
Power_Generator_Name	string	unique power generator name
Constant	boolean	mode of operation
StartTime	double	first data structure
Charge_Update_Interval	double	time between data structures
ChargeCapacity	double	charge rate in Watt-Hr
Motor_Generator	boolean	mode of operation
StartTime	double	first data structure
Motor_Charge_Setup	text	ID, RPM, Duration, ChargeCapacityWHR, and Efficiency columns used by the Motor Generator mode
Inter_Charge_Time	double	time between Motor_Charge_Setup
Trace	boolean	mode of operation
StartTime	double	first data structure
Trace_File_Name	string	file name
Mapped_Charge_CapacityWHR_Column	string	charge rate in WHR
Mapped_Time_Stamp_Column	string	time between data structures
Time-Based	boolean	mode of operation
StartTime	double	first data structure
Time_Based_Charge_Setup	text	columns for ID, StartWHR, EndWHR, Efficiency, and PercentTime
Time_Based_Duration	string	relates to Percent time column to calc time

Name	Type	Description
Charge_Enable	boolean	enables charging to maximum value

Note: More than one Mode can be enabled for complex motor generator operation.

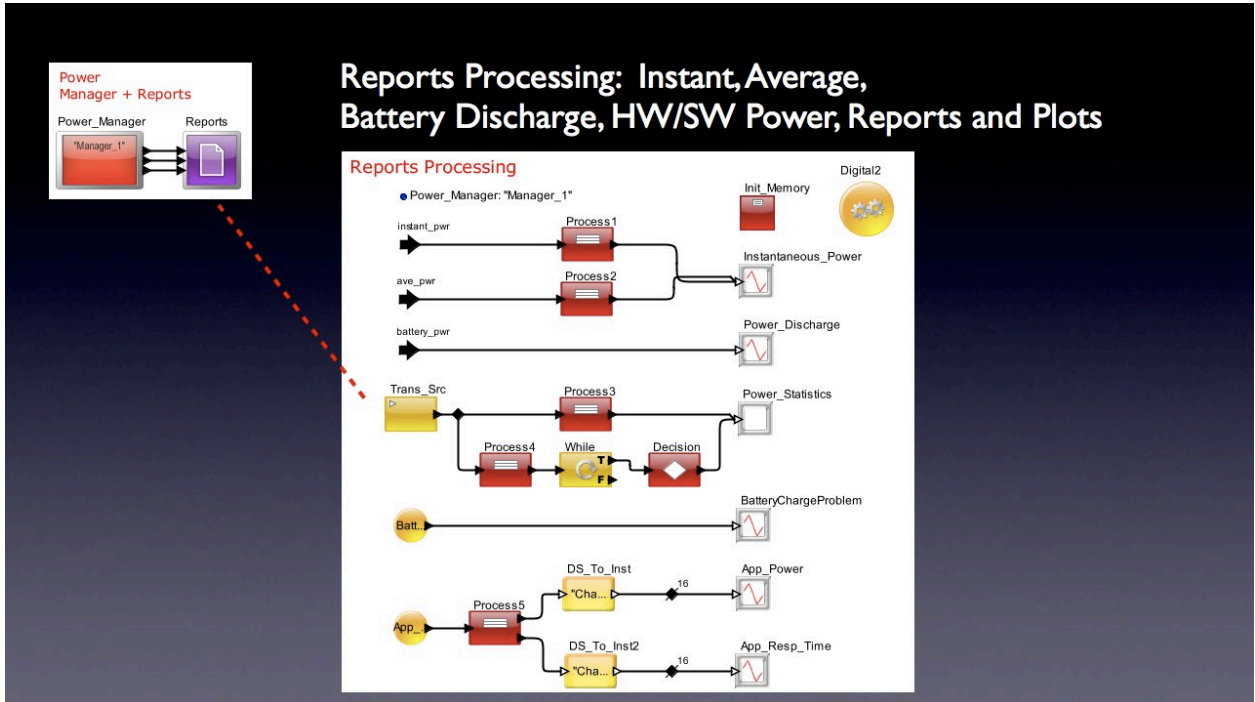


Figure 17. Reports Processing Block Diagram

Reports Processing

- Instant Power (watts), Average Power (watts), Battery Power (watts-seconds) plotted from Power_Manager
- Power Statistics: Text output with Average Power (watts), Minimum Power (watts), Maximum Power (watts), and Total Power (watts-seconds). Each Application shows Cumulative (watts-seconds) and average (watts).
- Battery Charge Problem: Plot indicates battery charge exceeds Maximum Battery Charge parameter with 1.0 values, else 0.0 values. The plot also indicates the simulation time on the X axis.
- Each Application Execution reports the power consumed and the latency.

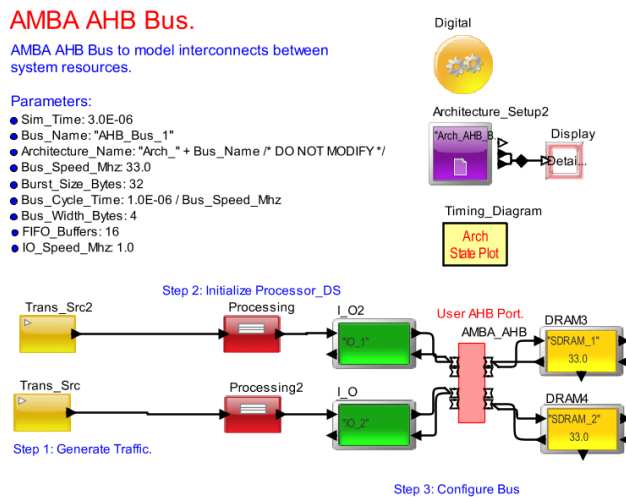
Bus and Interface Standards

1 AMBA Buses

1.1 AMBA AHB

The interconnects of the System-On-chip can be modeled using the AMBA AHB Bus block. The System resources are modeled using the Architecture library blocks; Processor, Cache, DRAM, I_O device.

1.1.1 Sample Model



1.1.2 Setup

To use the AMBA AHB Bus block certain Model Parameters, Model memories, Data Structure generated and flowing into the bus (ex: Processor_DS) etc need to be setup.

1.1.3 Model Parameters

- Sim_Time: 3.0E-06
- Bus_Name: "Bus_1"
- Architecture_Name: "Arch_" + Bus_Name
- Bus_Speed_Mhz: 100
- Burst_Size_Bytes: 64
- Bus_Width_Bytes: 8
- FIFO_Buffers: 8

1.1.4 Initialize the Processing Data Structure

The Standard library blocks like "Traffic" block can be used to generate the necessary Processor_DS. The following Processor_DS fields needs to be initialized to send transactions through the AMBA AHB Bus. This can be achieved with "Processing" or "Expression_List" blocks

Processor_DS

A_Command: Read or Write

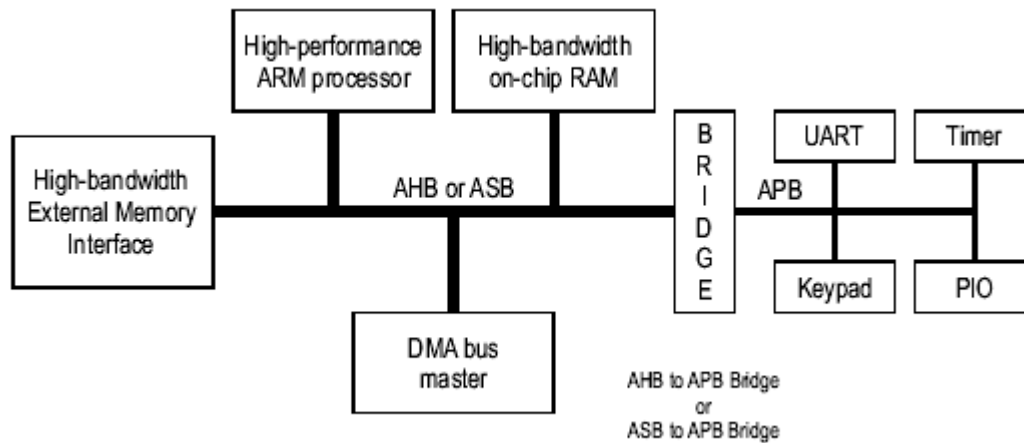
A_Bytes: Size of Bytes transferred in the transaction

A_Destination: Destination cache or memory

If an DeviceInterface block is used, the DeviceInterface block fields can also be used to replace the Processor_DS field values.

DeviceInterface field	MapsTo	Processor_DS field
IO_Destination	A_Destination	
IO_Command	A_Command	
IO_Bytes	A_Bytes	

1.1.5 A Typical AMBA Bus System Architecture



An AMBA-based microcontroller typically consists of a high-performance system *backbone* bus (AMBA AHB or AMBA ASB), able to sustain the external memory bandwidth, on which the CPU, on-chip memory and other *Direct Memory Access* (DMA) devices reside. This bus provides a high-bandwidth interface between the elements that are involved in the majority of transfers.

The AMBA AHB bus can be modeled using the AMBA AHB Bus Hierarchical block present in the Bus Library. The CPU, on chip memory etc can be modeled using the Processor, Superscalar_Processor, DRAM blocks in the Architecture library. The DMA can be modeled using the DMA_Controller block in the Bus library.

Also located on the high performance bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are located.

The AMBA APB Bus can be modeled using the AMBA APB Bus Hierarchical block present in the Bus Library. The bridge between the AHB and the APB can be modeled using the IO_Controller block in the Bus library.

The IO_Controller when operating in Custom mode can act as a bridge.

The IO_Controller block when operating in internal mode can also be used to model connecting the AMBA AHB bus to banks of memory. The DRAM in the memory bank can be accessed either using the address mode or using the DRAM name. PI refer to IO_Controller block documentation for details.

The connectivity to peripheral devices can be modeled using an I_O block connected to the bus.

1.1.6 AMBA Bus Features supported:

AMBA AHB

- ✓ High performance
- ✓ Pipelined operation
- ✓ Multiple bus masters
- ✓ Burst transfers
- ✓ Split transactions

AMBA APB

- ✓ Low power
- ✓ Latched address and control
- ✓ Simple interface
- ✓ Suitable for many peripherals

1.1.7 Typical AMBA System components and the Mapping to VisualSim

AMBA System component	Map to VisualSim
AHB Master	Typically a Processor or DMA. Traffic generators can also be used to send transactions to the bus routed through an I_O block that provides flexibility of setting certain fields of the Processor_DS.
AHB Slave	High bandwidth On-chip RAM or an External memory represented using the DRAM block.
AHB Arbiter	The AMBA AHB Hierarchical Bus consists of a BusArbiter (arbiter) and Linear Bus Ports (to connect the Master an slave devices). Supports FCFS, Custom arbitration.
AHB Decoder	Decoding the address of each transfer and provide a select signal for the slave that is involved in the transfer is abstracted as locating the slave using the routing table, A_Destination field represents the name of the slave and its availability is found by obtaining the block status.

1.1.8 Routing Table

The blocks directly connected to the bus would be located using the internal table maintained by the bus. To trace the route of a transaction flowing from the Processor through the bus to Cache or DRAM, entries in the routing table needs to be updated.

1.1.9 Routing Table for the Sample Model

```

/* First row contains Column Names.          */
Source_Node Destination_Node Hop      Source_Port ;
DeviceInterface_1      SDRAM_1      Port_1      output  ;
SDRAM_1      DeviceInterface_1      Port_2      output  ;
Port_1      SDRAM_1      Port_2      port2 ;
Port_2      DeviceInterface_1      Port_1      port1 ;

```

Tips: The easy way to identify the entries required is to run the model first without any routing entries, and the model will prompt for entry required between a Source block and a Destination node. The appropriate entries can be then added incrementally till it succeeds.

1.1.10 Bus Statistics

Statistics collected for the AMBA Bus include

- Throughput in Mbps
- Utilization percentage
- Input Output transactions per sec (IOs_per_second)
- Input buffer occupancy in words

1.1.11 Statistics of the Sample Model

Collected for a Read Transaction of 256 bytes data.

```

BLOCK              = ".AMBA_AHB_Bus_Model.Architecture_Setup",
Bus_1_Delay_Max    = 3.3E-7,
Bus_1_Delay_Mean   = 1.75E-7,
Bus_1_Delay_Min    = 2.0E-8,
Bus_1_Delay_StDev  = 1.55E-7,
Bus_1_IOs_per_sec_Max = 1.3333333333333333E6,
Bus_1_IOs_per_sec_Mean = 1.3333333333333333E6,
Bus_1_IOs_per_sec_Min = 1.3333333333333333E6,
Bus_1_IOs_per_sec_StDev = 0.0,
Bus_1_Input_Buffer_Occupancy_in_Words_Max = 64.0,
Bus_1_Input_Buffer_Occupancy_in_Words_Mean = 33.0,
Bus_1_Input_Buffer_Occupancy_in_Words_Min = 4.0,
Bus_1_Input_Buffer_Occupancy_in_Words_StDev = 21.2367605815953,
Bus_1_Throughput_MB_S_Max = 176.0,
Bus_1_Throughput_MB_S_Mean = 176.0,
Bus_1_Throughput_MB_S_Min = 176.0,
Bus_1_Throughput_MB_S_StDev = 0.0,
DELTA              = 0.0,
DS_NAME            = "Architecture_Stats",
ID                 = 1,
INDEX              = 0,
SDRAM_1_Delay_Time_Max = 5.76E-7,
SDRAM_1_Delay_Time_Mean = 2.88E-7,
SDRAM_1_Delay_Time_Min = 0.0,
SDRAM_1_Delay_Time_StDev = 2.88E-7,

```

```

SDRAM_1_Throughput_MBs_Max      = 45.3333333333333,
SDRAM_1_Throughput_MBs_Mean     = 45.3333333333333,
SDRAM_1_Throughput_MBs_Min      = 45.3333333333333,
SDRAM_1_Throughput_MBs_StDev    = 0.0,
TIME                             = 1.5E-6}

{BLOCK                            = ".AMBA_AHB_Bus_Model.Architecture_Setup",
Bus_1_Utilization_Pct_Max        = 24.6666666666667,
Bus_1_Utilization_Pct_Mean       = 24.6666666666667,
Bus_1_Utilization_Pct_Min        = 24.6666666666667,
Bus_1_Utilization_Pct_StDev      = 0.0,
DELTA                            = 0.0,
DS_NAME                          = "Architecture_Stats",
ID                                = 1,
INDEX                            = 0,
SDRAM_1_Utilization_Pct_Max      = 38.4,
SDRAM_1_Utilization_Pct_Mean     = 38.4,
SDRAM_1_Utilization_Pct_Min      = 38.4,
SDRAM_1_Utilization_Pct_StDev    = 0.0,
TIME                             = 1.5E-6}

```

1.1.12 Validation comparing with AMBA Spec

Figure 3-29 of AMBA Specification shows an AHB master arbitration and Timing diagram.

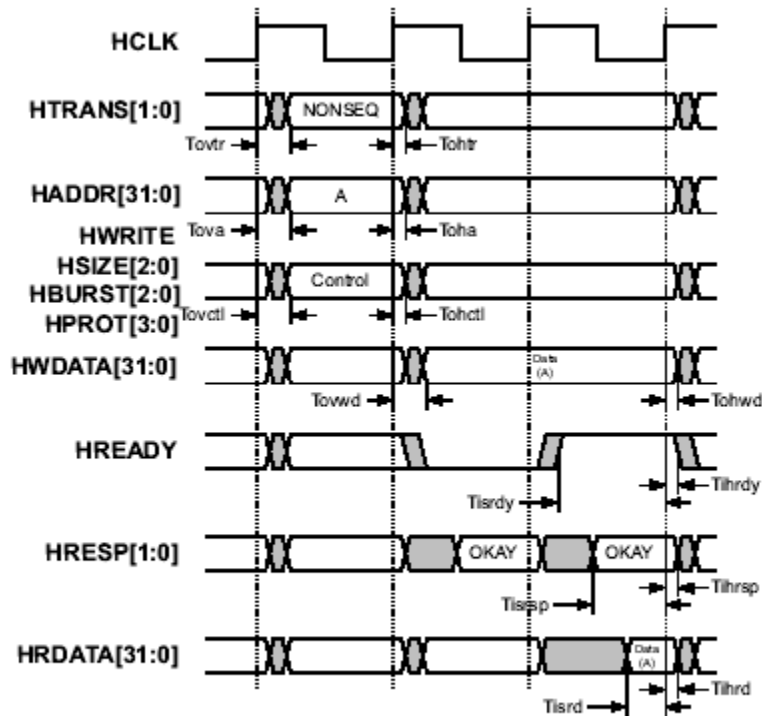


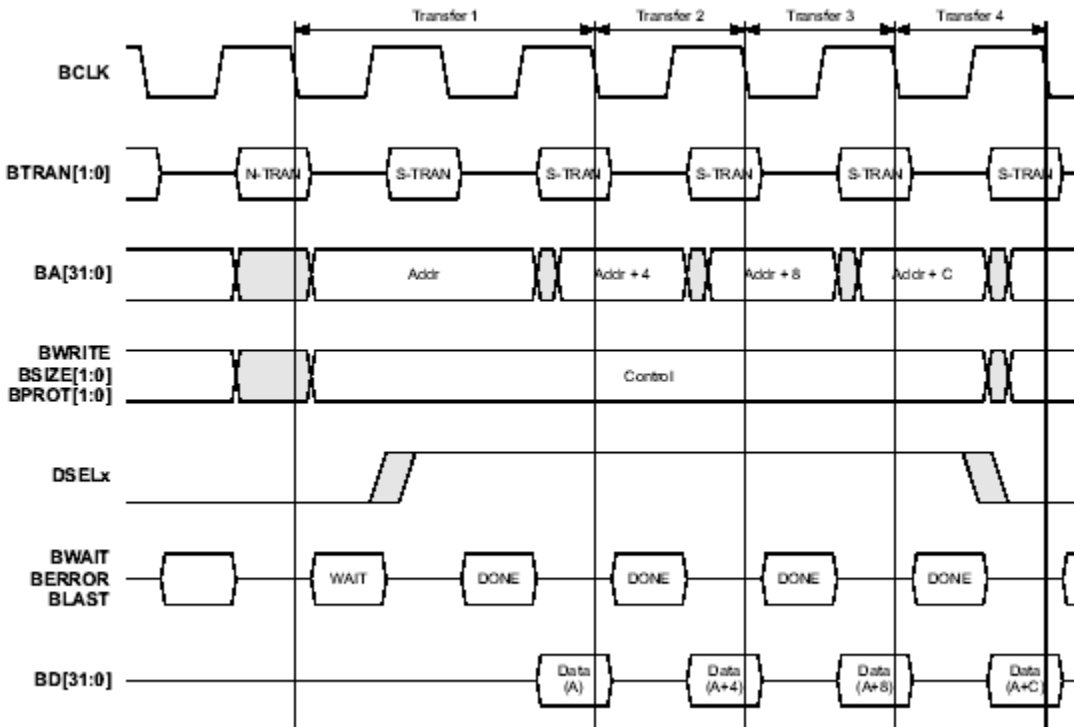
Figure 3-29 AHB master transfer timing parameters

The VisualSim AMBA AHB Bus implements the AMBA AHB Protocol at a higher level of abstraction. The A_Command field of the Processor DS maps to the transfer types Read or a Write. The A_Addr_Ctrl_Flag field of the Processor_DS maps to the Address/Control cycle. With the default FCFS mode of arbitration the VisualSim AMBA Bus works with the A_Addr_Ctrl_Flag set to true; i.e. introduces one address/control cycle. The slave response OKAY, SPLIT/RETRY etc is mapped to the Slave status.

In the VisualSim Bus Timing diagram, the Address/Control cycles can be seen as single plot of Bus Control (legend-BC). The Bus data transferred can be seen as plot of Bus Data (legend-BD).

The actual data Read or a data write happening at the slave (SDRAM) can be seen as a DRAM Access (legend-DA).

Figure 4-2 from the AMBA Specification shows the use of NONSEQUENTIAL and SEQUENTIAL transfers to perform a burst transaction.



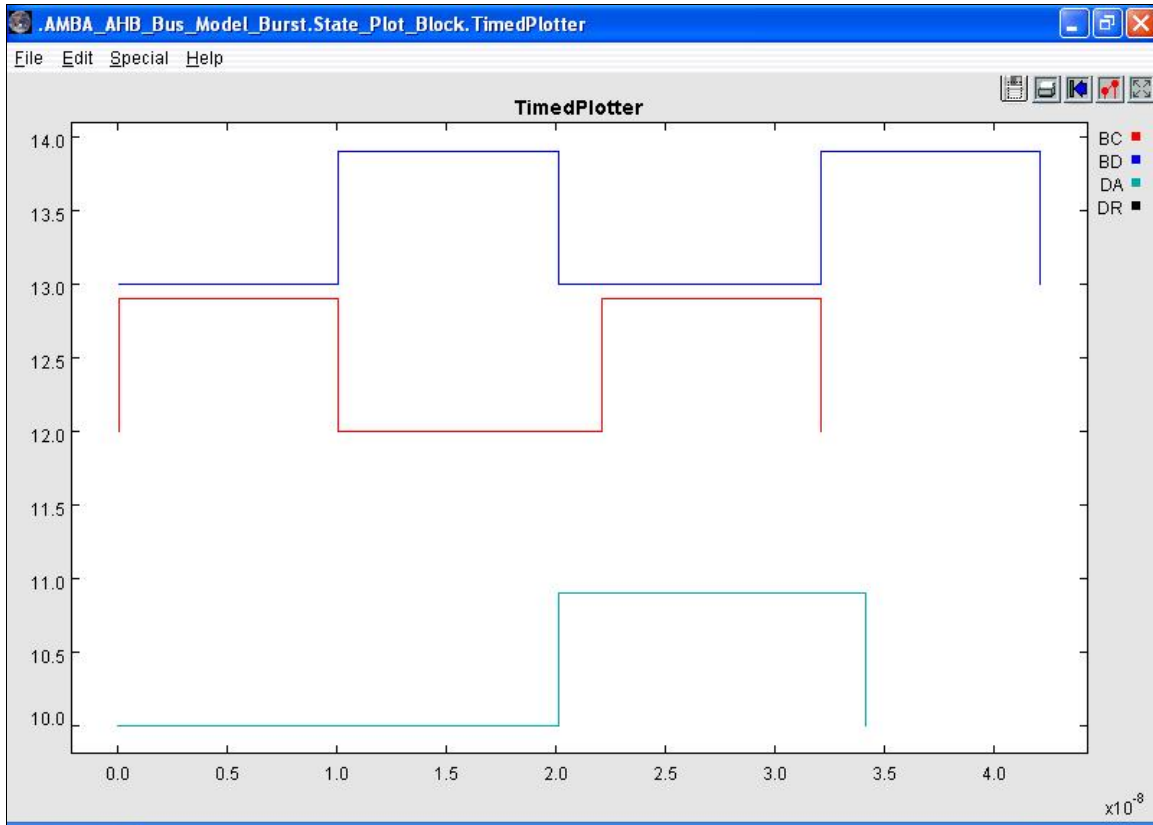
The VisualSim AMBA AHB Bus can be configured to have any of the supported data bus widths (8, 16, 32, 64, 128, 256, 512 or 1024-bits wide). However, it is recommended that a minimum bus width of 32 bits is used and it is expected that a maximum of 256 bits will be adequate for almost all applications.



The Width_Bytes parameter is used to specify the data bus width and the Burst_Size_Bytes parameter is used to specify the bytes that can be transferred in a burst operation.

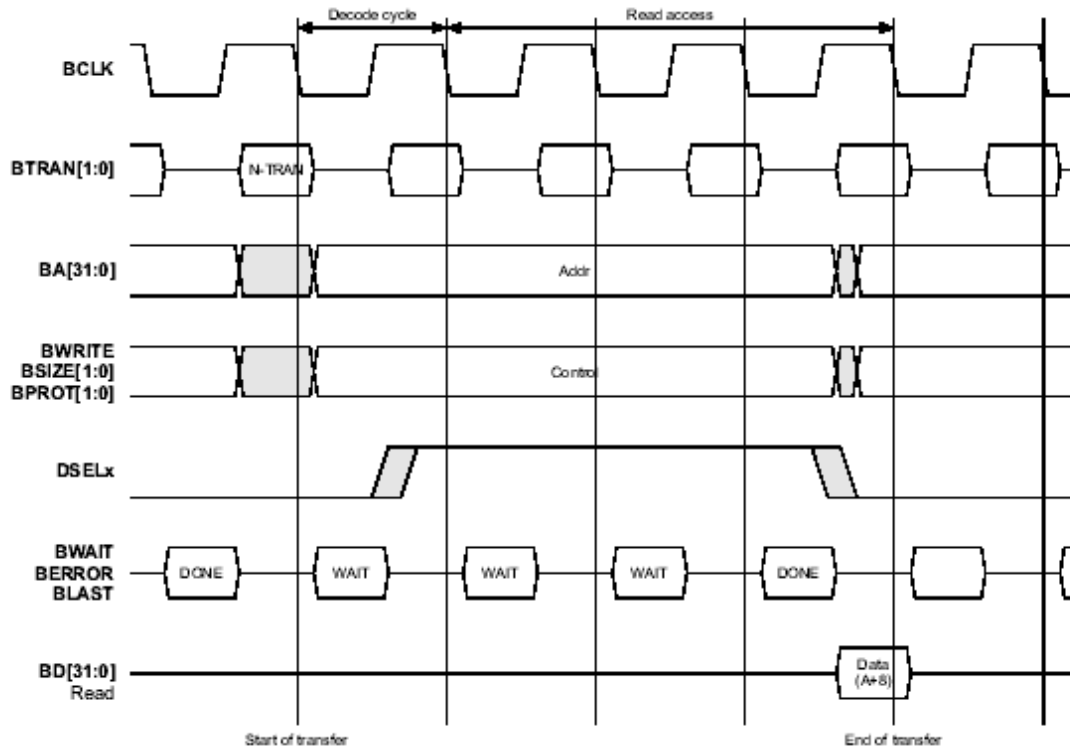
The Figure above shows address/control cycles followed by the data cycles for a Burst Read Operation. The VisualSim Timing diagram for a similar 4 byte Read transfer also depicts the same.

A similar burst operation of a 4 byte Read Transfer done with the VisualSim AMBA AHB Bus is shown in the following figure.



A NONSEQUENTIAL transfer occurs for either a single transfer or at the start of a burst of transfers. An 8 byte non-sequential Read transfer shows address/control cycles longer.

Figure 4-3 from AMBA Specification shows a typical NONSEQUENTIAL read transfer including wait states.



With the VisualSim AMBA AHB Bus configured to run in Custom mode, the address/control cycles could be manipulated by the user to see a similar behavior as seen in the specification.

The Processor_DS field A_Addr_Ctrl_Flag (boolean) represents an Address or Control Flag. To introduce longer address/control cycles, the A_Addr_Ctrl_Flag flag can be set to true in custom mode and the same would introduce an additional address/control cycle.

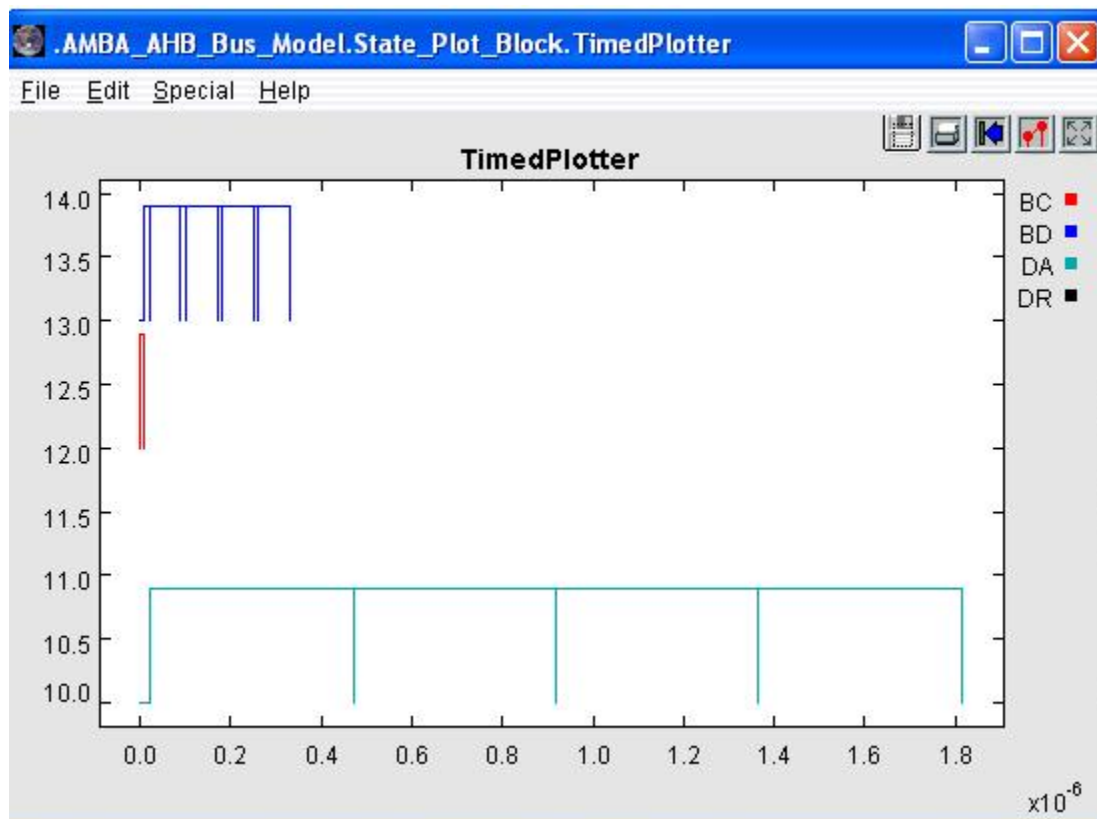
Operational View of the Model

The traffic generator pumps in transactions through the bus. The Processor_DS setup determines the type of Transaction – Read or a Write, the Size of bytes to be transferred, the source and destination blocks. On receiving a DS from the Bus Master, the Bus Port places it on the FIFO Buffer and transfers control to the Bus Controller. The Bus Controller decides which master to grant access to the bus and allows transfer through the bus to the respective slave. The Bus Controller handles the response from the slave (OKAY, Split/Retry) and processes till the transfer is complete.

A Write transaction is processed as follows.

1. The address/control signal (Write request) is transferred through the bus in one cycle (Sample model: Cycle time is 10 ns).
2. The data fragments are transferred through the bus taking several cycles, depending on the bus width, burst size, bus speed, bytes transferred etc. (Sample model: 32 clock cycles)
3. The slave now takes several cycles to write the data, nothing returns on successful completion.

Timing Diagram – Write Transaction



Data Fragmentation – Write Transaction

{A_First_Word, A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Address/Control Signal:

First fragment: {true, 256, 192, 64, Write} //sends address, 8 bytes

Data Transfer:

The 256 bytes to be transferred through the bus are sent in 4 burst operations (Burst_Size_Bytes = 64).

Burst #1

First fragment: {true, 256, 192, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 192, 64, Write} //sends remaining words, 56 bytes

Burst #2

First fragment: {true, 256, 128, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 128, 64, Write} //sends remaining words, 56 bytes

Burst #3

First fragment: {true, 256, 64, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 64, 64, Write} //sends remaining words, 56 bytes

Burst #4

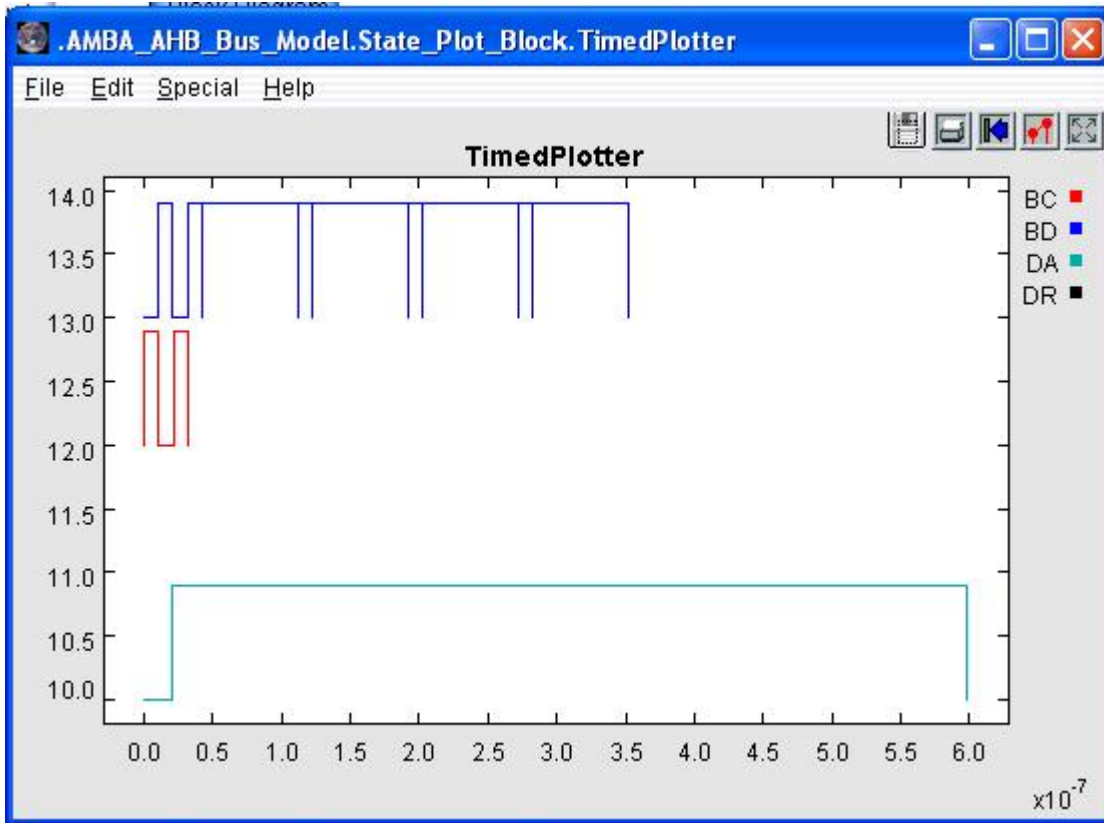
First fragment: {true, 256, 0, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 0, 64, Write} //sends remaining words, 56 bytes

A Read transaction is processed as follows.

1. The address/control signal (Read request) is transferred through the bus in one cycle (Sample model: Cycle time is 10 ns).
2. The slave now responds by performing the data Read and returns the data fragments setting the A_Command as 'Write' so the bus would return the data to the master.
3. The data fragments are transferred to the master through the bus taking several cycles, depending on the bus width, burst size, bus speed, bytes transferred etc. (Sample model: 32 clock cycles).

Timing Diagram – Read Transaction



Data Fragmentation

{A_First_Word, A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Address/Control Signal:

First fragment: {true, 256, 192, 64, Read} //sends address, 8 bytes

Data Transfer:

The slave DRAM reads and returns the data fragments setting the A_Command as 'Write'.

The 256 bytes to be transferred through the bus are sent in 4 burst operations (Burst_Size_Bytes = 64).

Burst #1

First fragment: {true, 256, 192, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 192, 64, Write} //sends remaining words, 56 bytes

Burst #2

First fragment: {true, 256, 128, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 128, 64, Write} //sends remaining words, 56 bytes

Burst #3

First fragment: {true, 256, 64, 64, Write} //sends first word, 8 bytes



Second fragment: {false, 256, 64, 64, Write} //sends remaining words, 56 bytes

Burst #4

First fragment: {true, 256, 0, 64, Write} //sends first word, 8 bytes

Second fragment: {false, 256, 0, 64, Write} //sends remaining words, 56 bytes

1.2 AMBA APB Bus

AMBA APB Bus is a subset of AHB Bus and is used with low bandwidth peripheral devices that does not require high bandwidth of the main system bus.

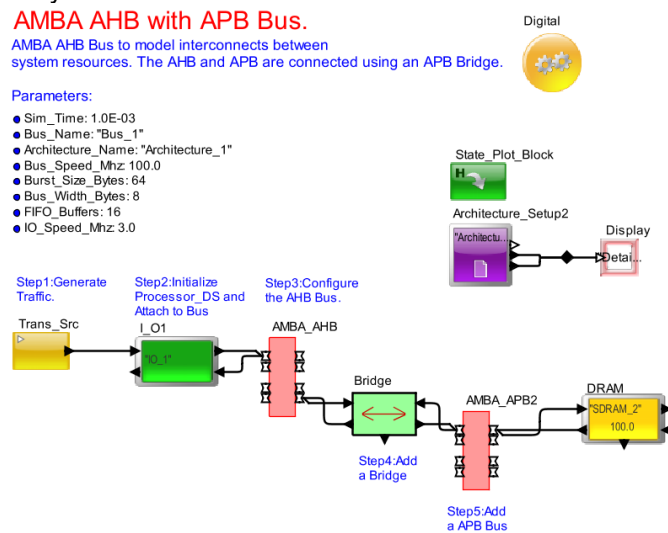
AMBA APB Hierarchical block models the APB Bus.

Features Supported:

- ✓ Low power
- ✓ Latched address and control
- ✓ Simple interface
- ✓ Suitable for many peripherals

The AMBA APB appears as a local secondary bus that is encapsulated as a single AHB or ASB slave device. APB provides a low-power extension to the system bus which builds on AHB or ASB signals directly.

APB Bridge connecting the AHB Bus and the APB Bus can be modeled using the Bridge block with Delay cycles of 1 cycle.



1.3 AMBA AXI

The AMBA AXI Bus protocol is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for a high-speed sub micron interconnect. The objectives of the AMBA AXI Bus are to be suitable for high-bandwidth and low-latency designs and enable high-frequency operation without using complex bridges. It meets the interface requirements of a wide range of components and it is suitable for memory controllers with high initial access latency and provide flexibility in the implementation of interconnect architectures.

The VisualSim AXI Bus library provides design engineers the ability to construct platform models with different variations of the AXI protocol quickly and efficiently. The library functionality and timing are fully open for the user to modify either using parameters or by editing the script written in the high-level Virtual Language. The library also has built-in flow control for communication with masters, slaves and other AXI buses. The AXI can be connected in any combination of topology.

1.3.1 Setup

To include the AXI Bus block in a model, the following steps must be followed:

- a. From Folder-> **Interfaces and Buses** → **AMBA** → **AMBA_AXI**, drag on to the BDE. This is an instance of the AMBA_AXI. When opening to view details, always select “Open Instance”.
- b. Set the block parameters to match your block diagram. Refer to the Parameter selection.
- c. Connect each Master to the multi-ports on left-side. Connect all Slaves to the right-side. Make sure you connect the input to the AXI first and then the output. All connections must be made to a input/output of a Hierarchical block or another a single block.
- d. The AXI Bus expects the Data Structure named Processor_DS, to be used. If you are using any other Data Structure template, make sure the fields are mapped appropriately. Look at Section 1.4 for a listing of all the fields used within the AXI block.
- e. The AXI block supports up to 16 Master and 8 Slave ports.
- f. Modify the various Thresholds to add the threshold for all the ports including the new Master ports.
- g. Select the number of channels and arbitration
- h. Select whether the Master or Slave will use the AXI arbitration

1.3.2 Sample Model

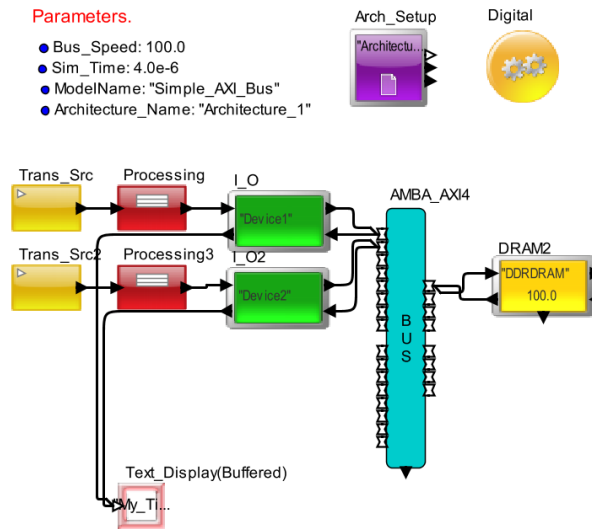


Figure 1-1 AMBA AXI Bus Model

1.3.3 Setup

To Run the Model, the Model Parameters and the transaction data structure flowing into the respective channel of the AXI Bus Block (ex: Processor_DS) etc need to be setup.

1.3.4 Model Parameters, Data Structure Fields and Ports

Parameter Name	Value (Data Type)	Explanation
Architecture_Name	"Architecture_1" (String)	Name of the Architecture Setup block
Bus_Name	"AXI" (String)	Unique name for this Bus. Different from all architecture blocks and global model memories.
AXI_Speed_MHz	200.0 (Double)	Bus Speed
AXI_Cycle_Time	1.0E-06 / AXI_Speed_MHz	AXI Bus Cycle Time calculation
Bus_Width	8 (Int)	Width
Write_Threshold	64 (Int)	Outstanding Write Data on the Bus. Depends on Threshold parameter below for units- bytes or transactions.
Read_Threshold	64 (Int)	Outstanding Read Data on the Bus. Depends on Threshold parameter below for units- bytes or transactions.
Master_Request_Threshold	{2,2,2,2,2,2,2,2}	Array list of number of outstanding requests per Master
Number_Master	16 (Int)	Connected Masters (Left-side)
Number_Slaves	8 (Int)	Connected Slaves (Right-side)
Threshold_Trans_T_Bytes_F	true for Transaction false for Bytes (Boolean)	This is the flow control mechanism Slave. If true, then the Bus holds transmission based on the number of

		transactions. If false, then the Bus holds the transmission based on the number of Bytes irrespective of the number of transactions.
Arbiter_FIX_1_RR_2_CUSTOM_3	Fixed Priority is 1; Round_Robin is 2 and user defined arbitration is 3. For 3, the Custom_File and Custom_Path fields must be set.1	1
Slave_Speeds_Mhz	{AXI_Speed_Mhz, AXI_Speed_Mhz, AXI_Speed_Mhz, AXI_Speed_Mhz}	This is used to synchronizes the timing between the slave devices and the bus. Each index is for the slave ports in order.
Extra_Cycles_for_RdReq_WrReq_RdData_WrData	{0,0,0,0}	For more details on this special parameter, read Section on Extra Cycles.
Devices_Attached_to_Slave_by_Port	{{"RAM1"}, {"RAM2"}} (Array of arrays of string)	This is an array of arrays. Each array contains the list of devices that are accessed via a Slave Port. Index 0 of the array is for Slave Port 1, Index 1 is for Slave Port 2 and so on. There must be one array with one value for each Slave. The names are strings.
Master_First_Word_Flag	true (Boolean)	true returns the first word while a false sends out the last word back to the Master for a Read operation
Single_Request_Channel	false (Boolean)	There are separate Read and Write Request queues by default, which is false. If set to true, all the Read/Write Requests are sent to the Read Request Queue from the Master Transaction block. The user will need to modify only the arbitration logic.
Master_Throttle_Enable	{true,true,true,true}	If set to true, then the Master will send a acknowledge Event to the device connected on that port using the return wire. The device cannot send any new Read Request or Write data until it receives a acknowledgement for the previous transfer. The Read and Write are handled independently. If set to false, the Read and Write Threshold will be the only factor affecting the flow control.
Slave_Throttle_Enable	{true,true,true,true}	If set to true, then the Slave will not send the next Read Request or Write data until it receives a acknowledgement for the previous transfer. The Read and Write are handled independently. If set to false, the Read and Write Threshold will be the only factor affecting the flow

		control.
Single_Request_Channel	true	Boolean value of true or false. if true, then there is a single request channel for reads and writes. if false, then the read and write have separate channels. The Read and Write thresholds are still utilized for flow control.
DEBUG	false (Boolean)	If set to true, the lower port outputs the current time, the active transaction and the channel.
Sim_Time	1.0 (Double)	This is simulation time and match the top-level Digital Simulator
Custom_Arbiter_File	"Custom_Script.txt"	Name of File containing custom arbitration in Virtual Machine script.
Custom_Arbiter_Path	"C:/VisualSim/Scripts for Windows /VisualSim/Scripts for UNIX"	Path to the custom arbitration script file.
Fixed_Priority_Array	{{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16},{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16},{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16},{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16},{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16},{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}}	Fixed Priority based on Port Numbers

Table 1 List of AXI Bus Block Parameters

1.3.5 Data Structure Field	Value (Data Type)	1.3.6 Explanation
A_Bytes	128 (Int)	Total bytes to be transferred
A_Bytes_Remaining	32 (Int)	Remaining bytes to transfer after current transaction
A_Bytes_Sent	8 (Int)	Bytes currently transferred and will equal the Width
A_Command	"Read" or "Write" (String)	Bus Operation. Internal activity will breakdown to Request, Address, Data Channel and Acknowledge
A_Message	Used internal to the Bus (String). No user setting	Indicate the type of transaction (Request , Response, Address out, Data and

	required and does not need to exist in the incoming Data Structure	ACK)
A_Priority	2 (Int)	Transaction Priority. Will reorder all queues according to priority.
A_Source	“ARM9” (String)	Master Name
A_Destination	“DRAM” (String)	Final Destination
A_Hop	“DRAM” (String)	Set to match A_Destination
AXI_Src_Arr	{“Master1”, “Master2”}	Used internally to return the transaction from Slave to the correct Master port
A_Prior_Des	1	Slave number sending the transaction
Event_Name	“Block_Event_Slave”	<p>This field contains the Event name that the sending device is waiting on.</p> <p>This is the name of the Event that the sending device is waiting on. If the Master Throttle is enabled for this port, the Master_port of the AXI will look for this field. If it does not exist, a error will be reported. If it exists and when the transaction is accepted, the AXI bus Master port will send this event. This is a notification for the device to send the next transaction. This is used for the flow control. Similarly on the AXI slave side, if the Slave Throttle is enabled, the Slave Port will wait for the Event to be received from the connected slave to send the next transaction.</p>

Table 2 List of Data Structure Fields used in the AXI Bus

Port Name	Type	Explanation
Input, input2, input3, input4, input5, input6, input7, input8, input9, input10, input11, input12, input13, input14, input15,	Multiport or input/output. Connection must be made in order- first input to this port and then output away from this port.	Each port is for one Master. Must be a Transaction (Data Structure of type Processor_DS)

input16		
output, output1, output2, output3, output4, output5, output6, output7, output8	Multiport or input/output. Connection must be made in order- first input to this port and then output away from this port.	Each port is for one Slave. Must be a Transaction (Data Structure of type Processor_DS)
Stats_Out	Output port	Statistics (Data Structures) and debugging (String) information.

Table 3 List of AXI Bus Ports

1.3.7 Master and Slave Queue Depths

The length of the Master and Slave queues can be used to throttle the transmission of transactions into the AMBA AXI bus. The thresholds are in memory arrays.

Memory Name	Value	Explanation
Read_Threshold_Array	{0,0,0,0}	Number of outstanding Read Requests. Index starts from 0, which corresponds to Slave 1. The value can be transaction or bytes depending on Threshold setting.
Write_Threshold_Array	{0,0,0,0}	Number of outstanding Write Requests. Index starts from 0, which corresponds to Slave 1. The value can be transaction or bytes depending on Threshold setting.
Read_Master_Array	{0,0,0,0,0,0,0,0}	Number of outstanding Read Requests at the Master Queues for all Slaves. Index starts from 0, which corresponds to Master 1. The value can be transaction or bytes depending on Threshold setting.
Write_Master_Array	{0,0,0,0,0,0,0,0}	Number of outstanding Write Requests at the Master Queues for all Slaves. Index starts from 0, which corresponds to Master 1. The value can be transaction or bytes depending on Threshold setting.

Table 4 Queue Depth Array Memory Locations

These are local memories inside the AMBA_AXI block. To access these memories, you need to first reference it into Virtual_Machine, Smart_Controller, Processor and Decision blocks using `X=readMemory("Model_Name.Hierarchical_Name.AXI_Block_Name.Write_Master_Array")`.

This will create a reference to the array. Define this in the initial section. After that, you can use it in your code as `X(Master_Number -1)`. Remember that array index starts from 0. To get the memory path, you can use the RegEx function called `readAllMemory`. This will provide the complete path. You can also use `local_memory` to get the hierarchical path.

1.3.8 Read and Write Request Channels

The AXI Bus supports two types of channel structure. The setting is based on the block parameter- Single_Request_Channel

- Separate Read and Separate Write channel
- Combined Read and Write channel

The Read_Threshold and Write_Threshold are used for the flow control in both cases, as described in the Flow Control section of this document.

1.3.9 Arbitration

Three types of arbitration are supported by the AXI Bus- Fixed Time Slot, Round Robin and User-Defined. The parameter “Arbiter_FIX_1_RR_2_CUSTOM_3” determines the arbitration for the Bus. 1 is for Fixed Time Slot, 2 is for Round-Robin and 3 is for Custom.

Fixed Slot Time: The fixed priority arbitration has time broken down into slots. The number of slots is equal to the AXI_Cycle * Number_of_Master parameter. Each slot is assigned to a master in the order listed. So, master 1 gets slot 1, Master 2 gets slot 2 and so on. At the respective slot, the arbiter checks the master for a transaction. If one is available, it sends it out else it waits a slot and arbitrates for the next slot. The slot time is the duration of one AXI cycle time.

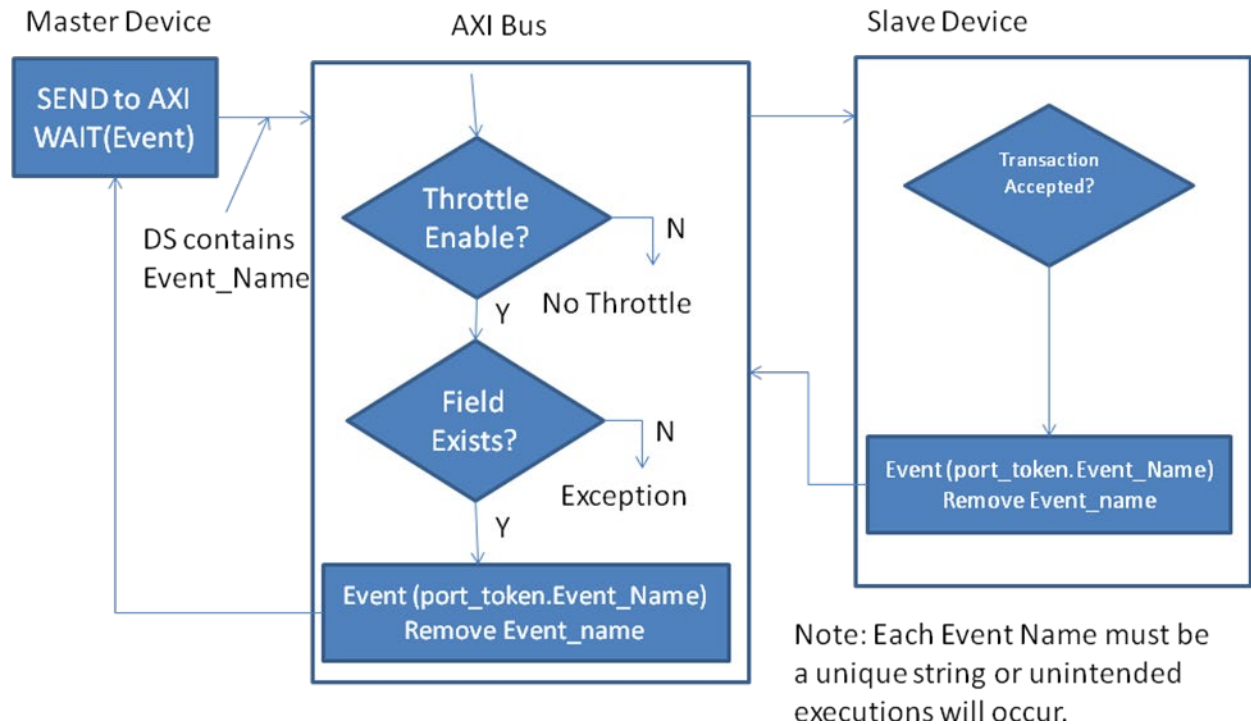
Round-Robin: This performs the round-robin between requests from the Master at every Slave. At the first clock cycle, it starts at Master 1 (input) and then scans each Master (order is based on the input port number). When it finds a request, it sends this out. The next clock cycle, it starts from the next Master after the one that was sent out last cycle. This is done at every Slave for every cycle.

Custom: The user can define a custom arbitration mechanism. The arbitration is described in the form of a Virtual Machine Script. The file name and path are set in the block parameters. Use the standard arbitration scripts as a template to construct your own algorithm.

1.3.10 Throttle Mechanism and Throttle block

The throttle mechanism is built into the AXI bus. The master and slave devices can use the event-based or TLM-style arbitration provided by the AXI throttle. Alternately, the modeler can define their own throttle mechanism by looking at the threshold memory arrays listed in Queue Depth array table above.

Standard Throttle



In the Throttle-enabled mode, the master and/or slave devices can be controlled by the AXI bus. Also, some devices can be controlled and other can be turned off based on the parameter setting. When a device sends a transaction to the AXI Master, the device goes into a Wait for Event state. The device sends a string value in the field called Event_Name to identify the unique Event expected. When the AXI master accepts the transaction, it sends the event with that name out. At the same time, the AXI master deletes the Event_Name field and sends it to the request channel. On the Slave side, the AXI adds the Event_Field with the string name and sends the transaction out. When the AXI receives the Event, it sends the next transaction out. Each master and slave device must maintain a unique name for the Event. This is done by concatenating the device name + event.

There are two arrays that keep track of the thresholds for the slaves (set to transactions):

```

    Read_Threshold_Array local {0,0,0,0} ; /* Current Read Thresholds */
    Write_Threshold_Array local {0,0,0,0} ; /* Current Write Thresholds */
  
```

Once, a request is sent from the Master Read_Master_Array to the slave, then the Read_Master_Array is decremented (one cycle typically), and the Read_Threshold_Array is incremented. When the first word of the read returns from the slave, the Read_Threshold_Array is decremented. These arrays give the user insight into pending master requests and active slave

transactions. The Read and Write Arbiters control the Read_Threshold_Array and Write_Threshold_Array internally. However, they can be monitored externally like the Read_Master_Array and Write_Master_Array.

If a user block is to be connected to the Master or Slave port of the AXI Bus, it is best to use the Flow_Control_Manager block (Hardware_Modeling->Bus_Switch_Ctrl Folder).

The Slave side throttle can be turned off by setting the "Slave_Throttle_Enable" for that slave port to false. If not, the slave will require an Event before it can send the next transaction, else will cause a lockup in the model.

1.3.11 Adding additional Master and Slave port

- i. The following steps must be followed to increase the Masters and Slaves:
 - a. Specify the number of Masters and Slaves in the parameter field called Number_Master and Number_Slaves respectively.
 - b. Modify Master_Request_Threshold to add additional indices for the new Master ports.
 - c. For Slave Ports, add additional indices for the new ports in the Slave_Speed_Mhz.
 - d. To enable or disable throttle, add an index for each additional master or slave. The master is **Master_Throttle_Enable** and slave is **Slave_Throttle_Enable**.
 - e. Update parameter "**Device_Attached_to_Slave_by_Port**" by adding the Slave_Name in Array.
 - f. To add new Master, copy the input8 port and Transaction8 block. Modify the parameter port_number to the next port number. Make the connection similar to the current method.
 - g. To add a Slave, copy Slave_4 block and output4 port. Make the connection similar to the current method. Modify the Slave_Number to the next value
 - h. The decoders are setup to support 12 Masters and 6 Slave. So, you will not have to change anything for this addition. If you go beyond this number, then you need to edit the script of Distribute_Data.
 - i. In the Distribute_Data, you will add the following lines for the definition:
 - One for each master greater than 12:
Master_13 = Bus_Name + "_Transaction_13"
 - For each slave greater than 6:
Slave_1 = Bus_Name + "_Slave_1"

- j. In the Distribute_Data, add the information for each additional master as below:

CASE: 13

SEND(Master_13, port_token, no_dup)

GTO (END)

- k. In the Distribute_Data, add the information for each additional slave as below:

CASE: 7

SEND(Slave_7, port_token, no_dup)

GTO (END)

1.3.12 AXI Bus Description

The AXI protocol is burst-based. Every transaction has address and control information on the address channel that describes the nature of the data to be transferred. The data is transferred between master and slave using a write data channel to the slave or a read data channel to the master. In write transactions, in which all the data flows from the master to the slave, the AXI protocol has an additional write response channel to allow the slave to signal to the master the completion of the write transaction. The AXI protocol enables address information to be issued ahead of the actual data transfer.

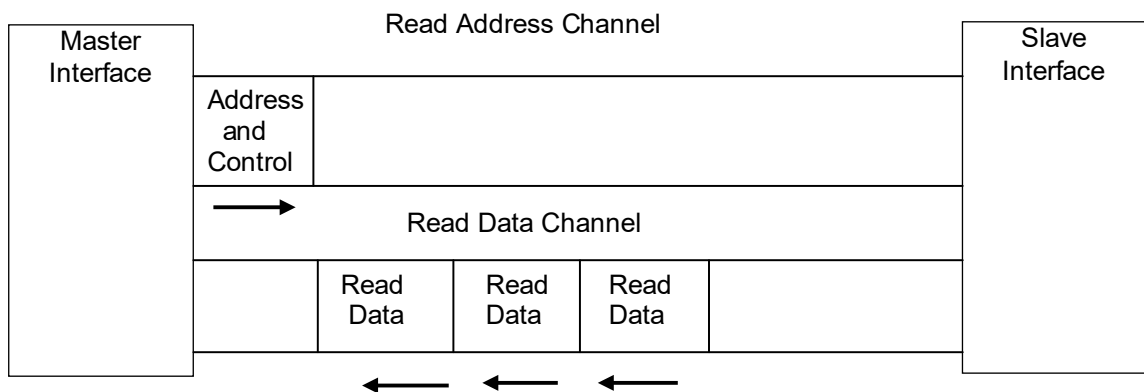


Fig. 1.2 Channel Architecture of Read Command

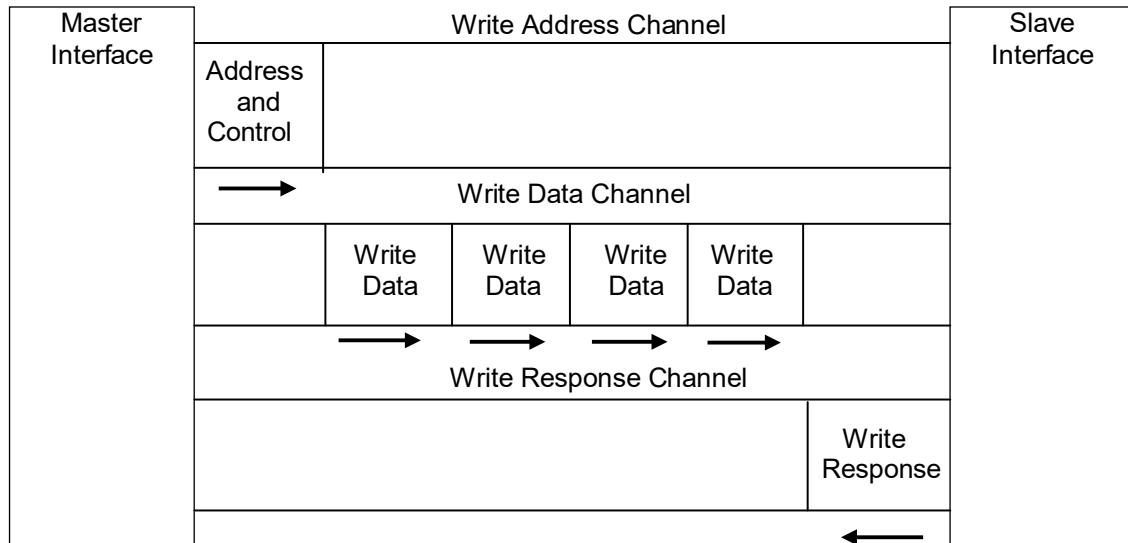


Fig. 1.3 Channel Architecture of Write Command

Read and Write Request Channels

Read and write transactions can be a combined channel or each can have their own Request channel. The appropriate Request channel carries all of the required address and control information for a transaction. The AXI protocol supports the following mechanisms are as follows: variable-length bursts, from 1 to 16 data transfers per burst, bursts with a transfer size of 8-1024 bits, and wrapping, incrementing, and non-incrementing bursts

Read Data Channel

The read data channel conveys both the read data and any read response information from the slave back to the master. The read data channel can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide. In the VisualSim AXI implementation, either the first or the last word in a burst is sent out only. The selection is based on the Master_First_Word_Flag setting. If true, the first word is sent.

Write Data Channel

The write data channel conveys the write data from the master to the slave and includes:

- The data bus, which can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide
- One byte lane strobe for every eight data bits, indicating which bytes of the data bus are valid.

Write data channel information is always treated as buffered, so that the master can perform write transactions without slave acknowledgment of previous write transactions.

Write Response Channel

The write response channel provides a way for the slave to respond to write transactions. All write transactions use completion signaling. The completion signal occurs once for each burst, not for each individual data transfer within the burst.

1.3.13 Flow Diagram

The Traffic Generator generates the token. The token field can be setup according to their requirement. The AMBA AXI Bus contain five different type of channel are as follows

Channel Name	AMBA AXI Specification Name	Description/Notes
Request_Channel	Read_Address_Channel	Address
Address_Out_Channel	Write_Address_Channel	Address
Data_Channel_1	Read_Data_Channel	Read data
Data_Channel_2	Write_Data_Channel	Write Data
ACK_Channel	Write_Response	Write Acknowledgement

Table 5 List of AXI channels implemented in the VisualSim AMBA AXI Bus Model

If the Read_Address_OK is false (by Default), and the threshold is fine, The Request is passed through Request_Channel to the respective slave. If a write operation, the Address_out is generated and then fragments the data and is passed through the Address_out Channel, Data_Channel respectively as shown in the figure below. When the Slave device status is compared to the Threshold value, if it exceed the Threshold value, negative acknowledgement will return to the Master results in resend the Request again.

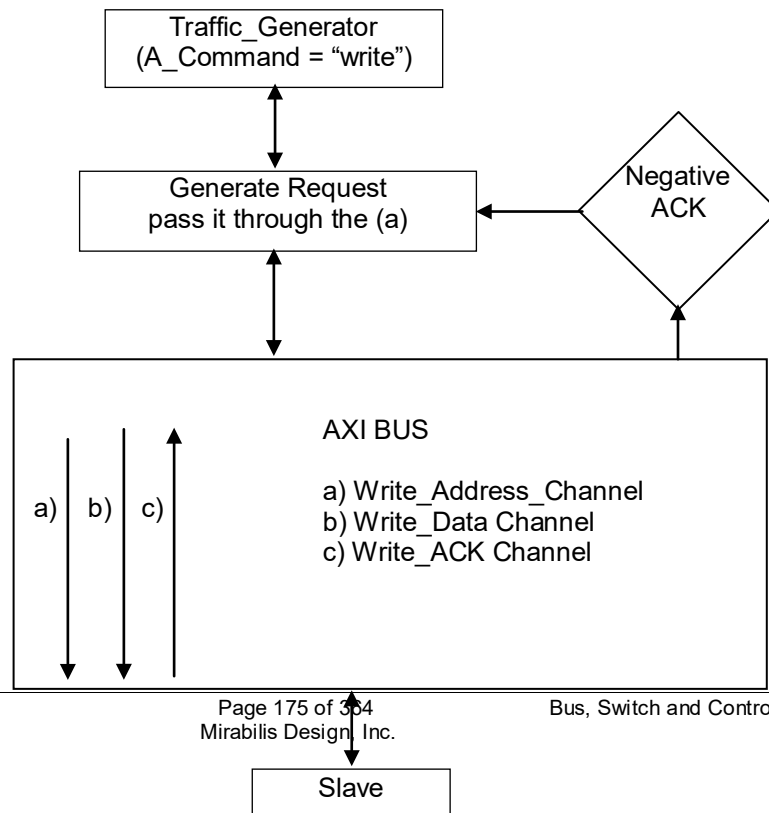


Fig. 1.4 Flow Layout of AMBA AXI Bus (Write operation)

Timing and Extra Cycle for Write Operation:

- WriteAddress = 1 cycle + Extra Delay
- WriteData = 1 cycle + Extra Delay for the first word. The transaction is sent out. Here the Data Structure fields are
 - A_Bytes=Full Data Size;
 - A_Bytes_Sent = AXI_Width;
 - A_Bytes_Remaining=(A_Bytes – A_Bytes_Sent).
- If the Write contains multiple words, there is an internal delay of 1 cycle/word for each of the remaining words. The extra cycle will not be added to these words. Say, that A_Bytes= 32 bytes and the AXI_Width = 8 bytes. The first word of 8 bytes will be delayed and output. The remaining Words will be delayed but will not be output.
- The remaining words will be in consecutive cycles. Out-of-order execution is not permitted by AMBA-AXI standard.

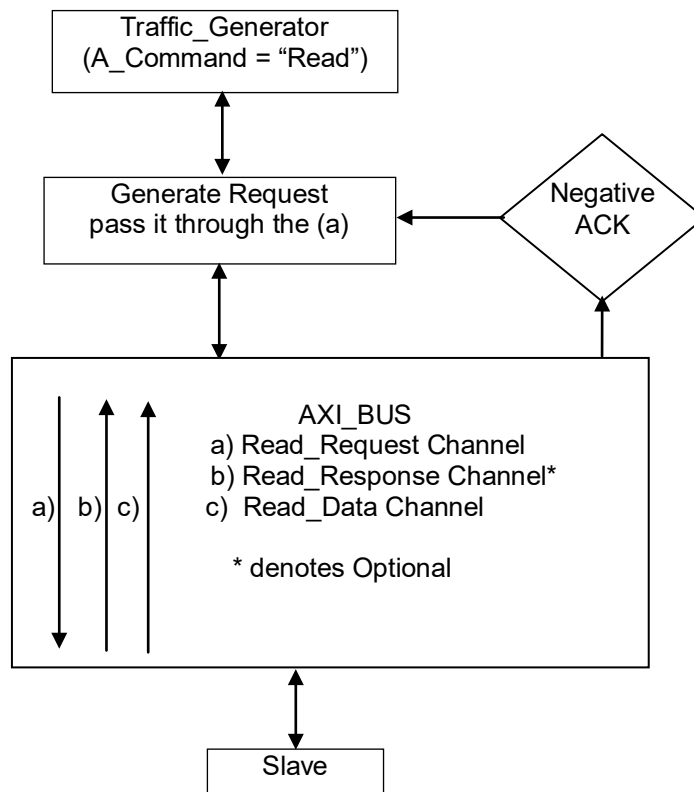


Fig. 1.5 Flow Layout of AMBA AXI Bus (Read operation)

Timing and Extra Cycle for Read Operation:

- ReadRequest = 1 cycle + Extra Delay
- ReadData = 1 cycle + Extra Delay for the first word. The transaction is sent out. Here the Data Structure fields are
 - A_Bytes=Full Data Size
 - A_Bytes_Sent = AXI_Width
 - A_Bytes_Remaining=(A_Bytes – A_Bytes_Sent).
- If the Read contains multiple words, there is an internal delay of 1 cycle/word for each of the remaining words. The extra cycle will not be added to these words. Say, that A_Bytes= 32 bytes and the AXI_Width = 8 bytes. The first word of 8 bytes will be delayed and output. The remaining Words will be delayed but will not be output.
- The remaining words will be in consecutive cycles. Out-of-order execution is not permitted by AMBA-AXI standard.

1.3.14 Bus Statistics

Standard statistics are collected for the various channels in the AXI bus:

- AXI_Top_Rd_Request_Queue: Read Request Channel
- AXI_Top_Wr_Request_Queue: Write Address Channel
- AXI_Top_Wr_Data_Channel: Write Data Channel
- AXI_Top_Rd_Data_Channel: Read Data Channel

The statistics fields are:

- Queue_Number corresponds to the Master number. In the Request channel, they correspond to a combination of Master+Slave. For example, Master 1 to Slave 1 will be Queue 1 while Master 2 to Slave 6 will be 14. The first 8 correspond to the 8 Master's going to Slave 1, second 8 are the Master's going to Slave 2 and so on.
- Total_Delay is the time across the channel for the entire transaction. The maximum size must be the Bus Width.
- Occupancy is the queue depth that is occupied. This is in number of transactions.
- Number of IOs per second
- Throughput in MB/sec

1.3.15 Latency Flow

The Latency for the Read and Write operations are shown below.

Read Command

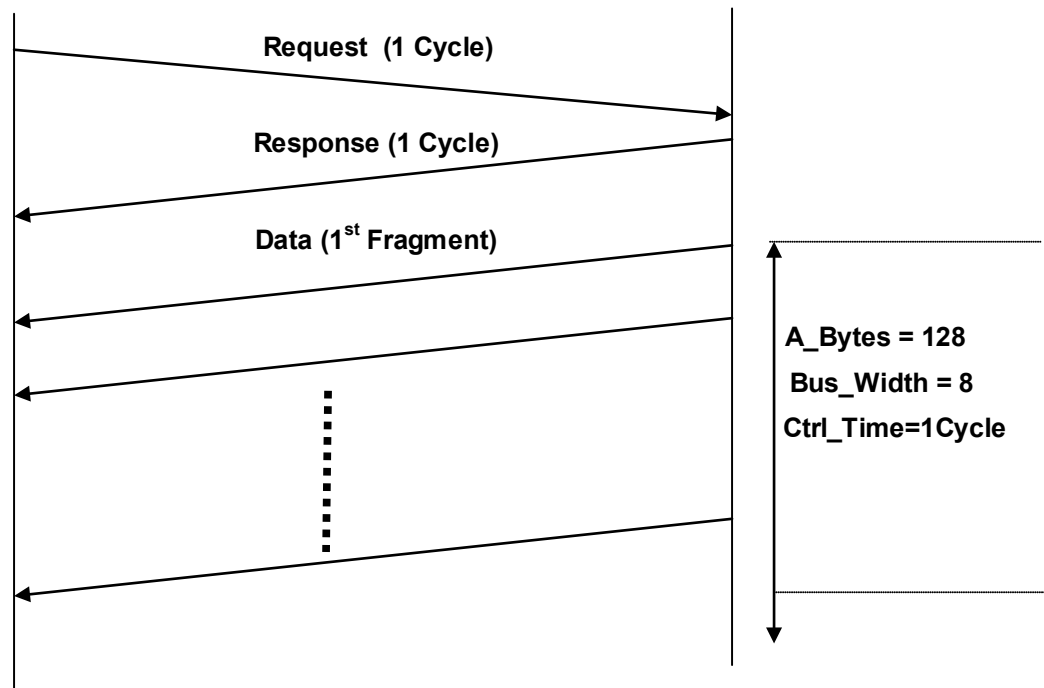


Figure 6 Total Latency for the whole READ operation = 20 Cycle

Note:

Overhead = Fragments * (Slave_Width / Slave_Speed) * (Bus_Speed / Bus_Width)
 Total_Data_Cycle = Ctrl_Time + (A_Bytes / Bus_Width) + DRAM_Latency

Write Command

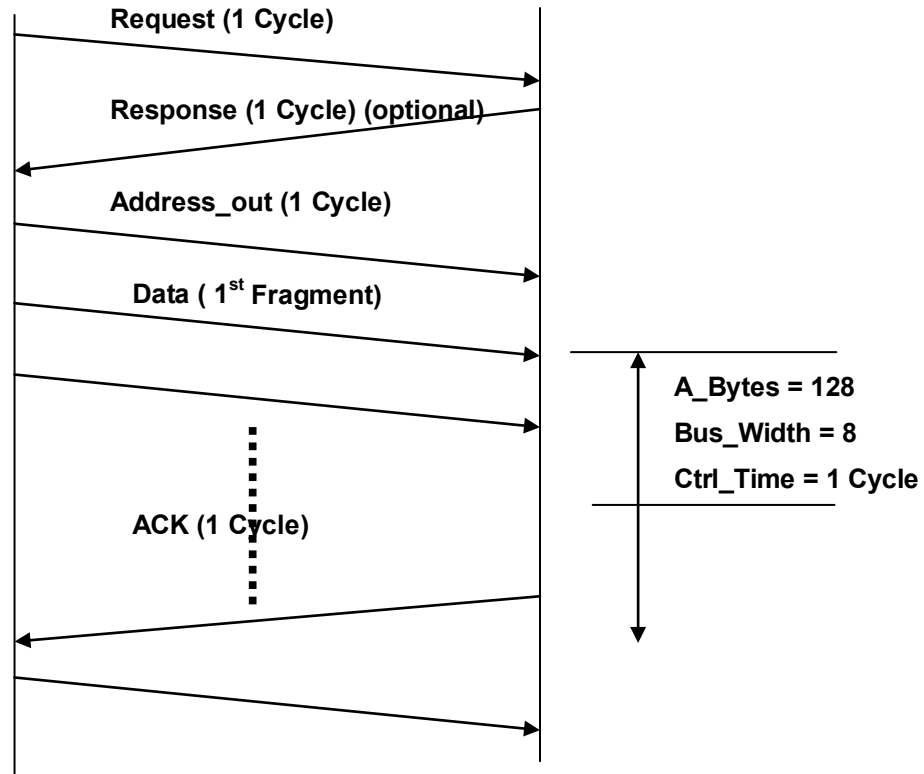


Figure 2: Total Latency for the whole WRITE operation = 22 Cycle

Note: $Total_Data_Cycle = Ctrl_Time + (A_Bytes / Bus_Width) + DRAM_Latency$

1.3.16 Operational View of the Model

All Masters and Slave transmit “Hello” messages at startup to indicate their presence and the connectivity location. This is used to build up the routing table. AXI Bus requires the listing of the Slaves that are reached through each Slave_Port. When a transaction arrives, the bus determines the type of Transaction – Read or a Write, the Size of bytes to be transferred, the source and destination blocks and whether the transmitted is Read/Write command from the source. On receiving a DS from the source, the AXI_Bus send the Request through the Request_Channel to the slave. For a Write operation, the data is fragmented to the Bus width and transmitted on the Data_Channel to the respective Slave.

1.3.16.1 Write Operation

A Write transaction is processed as follows.

1. The address/control signal (Write request) is transferred through the bus in one cycle

2. The data fragments are transferred through the bus taking several cycles, depending on the bus width and bytes transferred.
3. At the end of the transfer, the slave takes one additional cycles to write the data and return the Write Response.

Data Fragmentation – Write Transaction

{First_Word, A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Address/Control Signal:

First fragment: {true, 128, 120, 8, Write} //sends address **through Write_Request_Channel**, 8 bytes

Data Transfer:

The 128 bytes to be transferred are sent in 16 burst operations **through the Write_Data_Channel**.

First fragment: {true, 128, 120, 8, Write} //sends first word, 8 bytes
 Remaining fragment: {false, 128, 0, 120, Write} //sends remaining 15 words, 8 bytes

1.3.16.2 Read Operation

A Read transaction is processed as follows.

1. The address/control signal (Read request) is transferred through the bus in one cycle.
2. The slave now responds by performing the data Read and returns the data fragments setting the A_Command as 'Write' so the bus would return the data to the master.
3. The data fragments are transferred to the master through the bus taking several cycles, depending on the bus width and bytes transferred.

Data Fragmentation – Read Transaction

{A_First_Word, A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Address/Control Signal:

First fragment: {true, 128, 120, 8, Write} //sends address **through Read_Request_Channel**, 8 bytes

Data Transfer:

The slave DRAM reads and returns the data fragments setting the A_Command as 'Write'. The 128 bytes to be transferred are sent in 16 burst operations **through the Read_Data_Channel**.

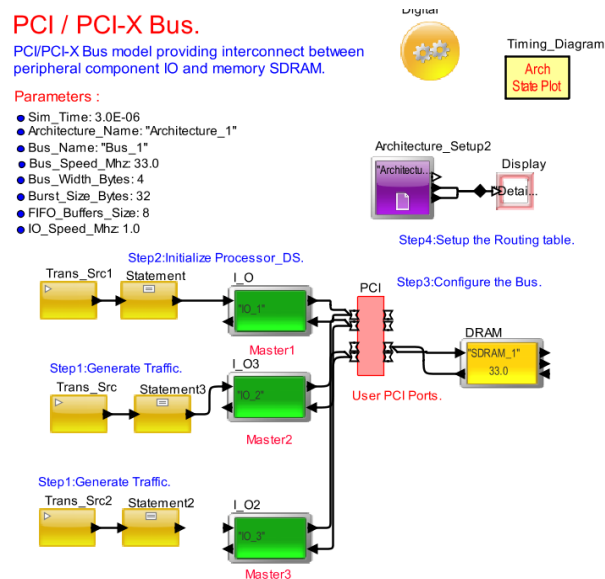
2 PCI Family of Buses

2.1 PCI and PCI-X Bus

The **PCI, PCI-X Local bus** protocol is a high performance bus for interconnecting peripheral chips to any independent processor/memory subsystems. The PCI bus block can use as interconnect between high bandwidth peripherals closer to the CPU for performance gains.

The Systems are modeled using the **Architecture library blocks**: *Processor, Cache, DRAM, I_O* device. The PCI, PCI-X buses are implemented using *Hierarchical_Block* that contains *BusArbiter* and *Linear_Port* library blocks.

2.1.1 Sample Model



2.1.2 Setup

To use the PCI Bus block, Model Parameters, Model memories, Data Structure generated and flowing into the bus (ex: Processor_DS) etc need to be setup.

2.1.3 Model Parameters

The parameter for a system that uses implementation of **32-bit PCI bus at 33 MHz**:

- Sim_Time: 3.0E-06
- Bus_Name: "Bus_1"
- Architecture_Name: "Architecture_1"
- Bus_Speed_Mhz: 33.0
- Burst_Size_Bytes: 32
- Bus_Width_Bytes: 4
- FIFO_Buffers: 8

2.1.4 Initialize the Processing Data Structure

The following are sample data structure fields to be set before a data transfer.

- A_Bytes* – total no. of bytes to be transfer
- A_Bytes_Remaining* – bus block set the field with remaining no. of data after every transfer
- A_Bytes_Sent* – bus block set the field with no. of data transferred
- A_Command* – represent bus command, the PCI bus commands can be specified as per the mapping given below:

PCI Bus Command	A_Command field
IO Read	“Read_IO”
Memory Read	“Read_Memory”
Memory Read Multiple	“Read_Multiple_Memory”
Memory Read Line	“Read_Line_Memory”
IO Write	“Write_IO”
Memory Write	“Write_Memory”

Any Read or Write command whether it is from an IO device or Memory it is handled by the bus in a similar fashion. Hence the bus looks for *A_Command* that starts with the “Read*” string to process a Read transaction and starts with the “Write” string to process a Write transaction.

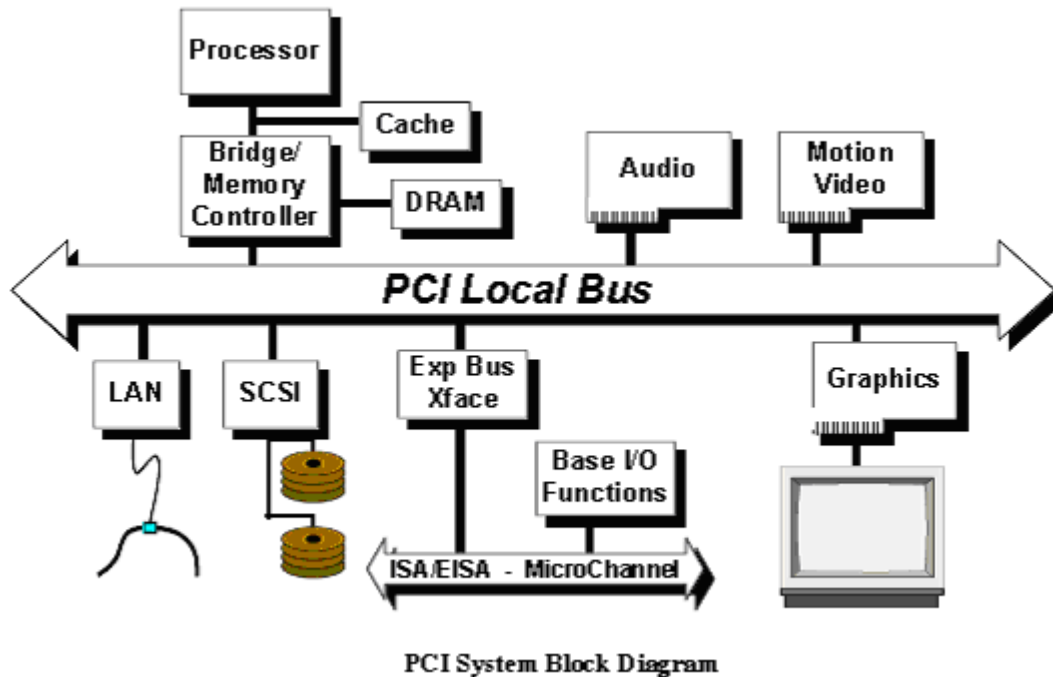
- A_First_Word* – default is true, Bus block sets to true, if the first word is sent out in a fragment, otherwise set to false.
- A_Priority* – can set the priority of the device, higher priority master gained the bus access. The multiple transactions flowing into the bus with different priorities are covered in chapter 4, Architecture Modeling.
- A_Source* – routing source, an initiator.
- A_Hop* – routing hop, bus block sets and use the fields for internal routing table.
- A_Status* – routing status, bus block sets and use the fields for internal routing table.
- A_Destination* – routing destination, a target.

A sample set of data, a user can set in fields of data structure “Processor_DS”:

Data Structure Field	Data Type	Sample Data
<i>A_Bytes</i>	int	128
<i>A_Command</i>	string	“Read_Memory”
<i>A_First_Word</i>	boolean	true
<i>A_Priority</i>	int	1
<i>A_Source</i>	string	“IO_1”
<i>A_Destination</i>	string	“SDRAM_1”

The data structure *Processor_DS* can be generated using the block *Transaction_Source* at given time. The block *Statement* is used to modify the fields of the generated data structure. The fields of data structure *Processor_DS* are also used as parameter values in block *I_O*.

2.1.5 A typical PCI, PCI-X system



In the PCI system model, the processor/cache/memory subsystem is connected to PCI bus through a *PCI bridge*. This PCI-Bridge/Memory Controller provides a low latency path through which the processor may directly access PCI devices mapped anywhere in the memory. The PCI and PCI-X buses established a central role in I/O systems, which connect the different peripherals (storage, network, graphics) and low-speed devices (keyboard, serial communications) to modern Servers and Workstation in the system. Connected to these “backbones” are several other I/O systems, such as SCSI buses. These are connected to the PCI/PCI-X system through bridge devices.

Typical PCI, PCI-X System components and the Mapping to VisualSim

The PCI, PCI-X System can build with standard library blocks available in Hardware Architecture Generator Tool Kit and Bus Modeling Tool Kit. The hierarchical block *PCI_Bus* represent the PCI and PCI-X buses. The components of subsystem processor/cache/memory can model with blocks *Processor*, *Cache*, and *DRAM*. The block *IO_Controller* can act as bridge for connection between multiple PCI, PCI-X and other IO buses. The block *I_O* can act as a port for connecting IO chips communicate to the Bus.

Each library block has documentation, which provides details on its own parameters and usage in chapter 4, Architecture Modeling.

The sample model contains one PCI bus providing interconnects between peripheral components named *IO_1*, *IO_2* and memory *SDRAM_1*.

PCI, PCI-X Bus features supported:

- Synchronous bus with operation up to 33 MHz, 66 MHz, or 133 MHz.
- Supports multiple families of processors as well as future generations of processors (by bridges or by direct integration).

- Support for up to six PCI bus masters.
- FCFS and Custom arbitration scheme.
- Preemption
- Split/Retry transactions
- Support for bus parking (Not supported)
- Support for 64-bit addressing (Not supported)

2.1.6 Routing Table

Route instructions between the blocks I_O, I_O2, SDRAM_1 connected to the bus block PCI_Bus are entered in Routing Table. The sample model has the following entries in routing table:

```

/* First row contains Column Names.                                     */
Source_Node Destination_Node Hop Source_Port ;
IO_1          SDRAM_1         PCI_Port_1  to_bus ;
IO_2          SDRAM_1         PCI_Port_3  to_bus ;
SDRAM_1       IO_1            PCI_Port_2  output ;
SDRAM_1       IO_2            PCI_Port_2  output ;

```

Bus block create its own routing map internally, so bus ports PCI_Port_1 etc., does not need to be added as sources or destinations to the routing table. However if any routing entries are missing, exceptions are thrown with saying the missing source and destination pair during the model execution.

2.1.7 Bus Statistics

The following statistics are collected in the same PCI, PCI-X bus model.

- **Delay** – Transaction delay
- **IOs_per_sec** – Input Output transactions per sec
- **Input_Buffer_Occupancy_in_Words** – Input buffer occupancy in words
- **Preempt_Buffer_Occupancy_in_Words** – Preempt buffer occupancy in words
- **Throughput_MBps** – Throughput in Mbps
- **Utilization_Pct** – Utilization percentage

The sample statistics collected in the model are given below in min, mean, stdev, and max values for all architecture resources.

```

{BLOCK                               = ".PCI_Bus_Model.Architecture_Setup",
Bus_1_Delay_Max                      = 9.999E-7,
Bus_1_Delay_Mean                     = 9.999E-7,
Bus_1_Delay_Min                      = 9.999E-7,
Bus_1_Delay_StDev                    = 0.0,
Bus_1_IOs_per_sec_Max                = 666666.6666666666,
Bus_1_IOs_per_sec_Mean               = 666666.6666666666,
Bus_1_IOs_per_sec_Min                = 666666.6666666666,
Bus_1_IOs_per_sec_StDev              = 0.0,

```

```

Bus_1_Input_Buffer_Occupancy_in_Words_Max = 32.0,
Bus_1_Input_Buffer_Occupancy_in_Words_Mean = 10.66666666666667,
Bus_1_Input_Buffer_Occupancy_in_Words_Min = 0.0,
Bus_1_Input_Buffer_Occupancy_in_Words_StDev = 15.084944665313,
Bus_1_Throughput_MBps_Max = 85.33333333333333,
Bus_1_Throughput_MBps_Mean = 85.33333333333333,
Bus_1_Throughput_MBps_Min = 85.33333333333333,
Bus_1_Throughput_MBps_StDev = 0.0,
DELTA = 0.0,
DS_NAME = "Architecture_Stats",
ID = 1,
INDEX = 0,
SDRAM_1_Delay_Time_Max = 2.3E-7,
SDRAM_1_Delay_Time_Mean = 1.15E-7,
SDRAM_1_Delay_Time_Min = 0.0,
SDRAM_1_Delay_Time_StDev = 1.15E-7,
SDRAM_1_Throughput_MBps_Max = 96.0,
SDRAM_1_Throughput_MBps_Mean = 96.0,
SDRAM_1_Throughput_MBps_Min = 96.0,
SDRAM_1_Throughput_MBps_StDev = 0.0,
TIME = 1.5E-6}

```

```

{BLOCK = ".PCI_Bus_Model.Architecture_Setup",
Bus_1_Utilization_Pct_Max = 68.6868686868687,
Bus_1_Utilization_Pct_Mean = 68.6868686868687,
Bus_1_Utilization_Pct_Min = 68.6868686868687,
Bus_1_Utilization_Pct_StDev = 0.0,
DELTA = 0.0,
DS_NAME = "Architecture_Stats",
ID = 1,
INDEX = 0,
SDRAM_1_Utilization_Pct_Max = 61.33333333333333,
SDRAM_1_Utilization_Pct_Mean = 61.33333333333333,
SDRAM_1_Utilization_Pct_Min = 61.33333333333333,
SDRAM_1_Utilization_Pct_StDev = 0.0,
TIME = 1.5E-6}

```

2.1.8 Statistics Validation comparing with PCI Specification

The PCI bus standard is validated against as the *PCI Local Bus Specification*, Revision 2.3.

Model Statistics:

```

A_Bytes : 128
Burst_Size_Bytes : 32
Bus_Width_Bytes : 4

```

Transaction	Throughput_MBps	Utilization_Pct	Clocks	Latency (us)
Burst Read	88.0	74.74	1 + 32	1.0705
Burst Write	85.33	68.6	1 + 32	1.0001

Table-1: Statistics for burst transfer

A_Bytes : 4
 Burst_Size_Bytes : 32
 Bus_Width_Bytes : 4

Transaction	Throughput_MBps	Utilization_Pct	Clocks	Latency (us)
Single Read	5.33	8.08	1 + 1	0.1312
Single Write	2.67	4.04	1 + 1	0.0606

Table-2: Statistics for Single word transfer

Specification statistics:

Latency for Different Burst Length Transfers

Data Phases	Bytes Transferred	Total Clocks	Latency Timer (clocks)	Bandwidth (MB/s)	Latency (µs)
8	32	16	14	60	.48
16	64	24	22	80	.72
32	128	40	38	96	1.20
64	256	72	70	107	2.16

The statistics obtained in the model are closely related to the statistics given in the specification.

2.1.9 Operational View of the Model

The model uses one PCI bus communicate between devices *IO_1*, *IO_2* and *SDRAM_1*. The hierarchical *PCI_Bus* block has six input/output multiports (bi-directional buses). The device *IO_1* requested the bus access for transferring 128 bytes (parameter *IO_Bytes* = *A_Bytes*) of data to the target (*IO_Destination* = *A_Destination*) *SDRAM_1*.

In the sample model assuming that neither the initiator nor the target inserts wait states during each data phase, the maximum theoretical bandwidth over a 32-bit bus (**Bus_Width_Bytes = 4**) is 132 Mbytes/second and perform continuous bursting with a 32-bit (**Burst_Size_Bytes = 32**) data transferred on each PCI clock cycle.

Transaction flow:

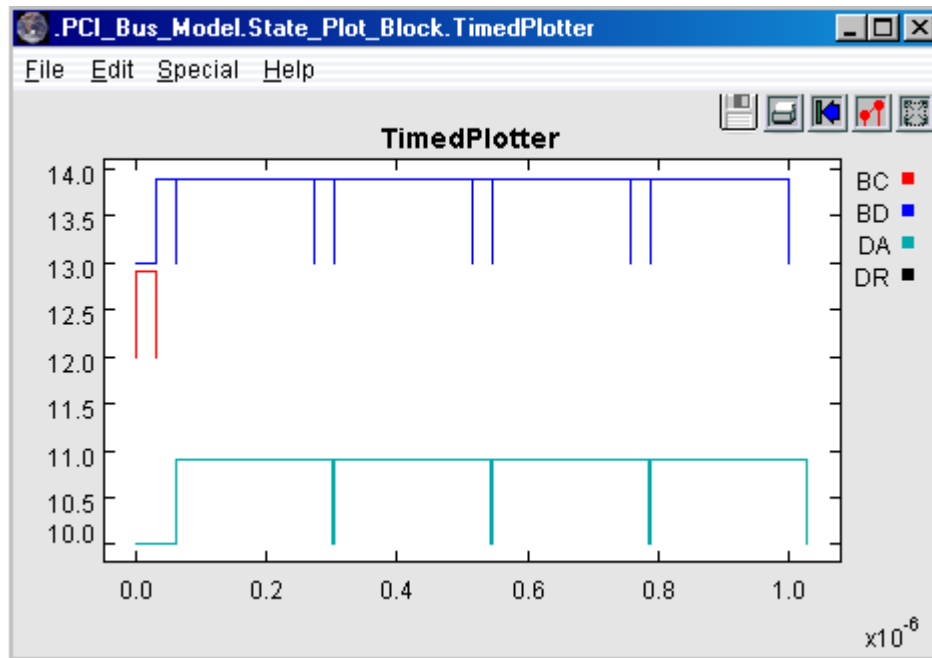
Write Transaction:

The PCI bus operation is "Write" (*IO_Command* = *A_Command*). Address and data transfers are multiplexed over the same lines on the PCI bus, the address is sent first and then the data.

In the timing diagram of sample model, Address out (**BC signal**) is completed in one cycle time, takes 30.3 ns for *Bus_Speed_Mhz* = 33.0.

{*A_First_Word*, *A_Bytes*, *A_Bytes_Remaining*, *A_Bytes_Sent*, *A_Command*}

Address/Control phase: {true, 128, 96, 32, "Write"} // PCI_Bus sent address, 4 bytes to SDRAM_1



Data Phase:

After the address out is completed bus start sending data fragments (**BD signal**) to the target SDRAM_1. In the sample model PCI bus sent 128 bytes (*IO_Bytes = A_Bytes*) in four bursts. Hence take 32 clock cycles to complete entire data transfers.

Fragmentation on Write data:

The bus sent the first word with field *A_First_Word* as true (4 bytes), and sent the remaining words with field *A_First_Word* as false (28 bytes) in all the burst fragments.

Burst #1

First fragment : {true, 128, 96, 32, "Write"} //sent first word, 4 bytes
 Second fragment: {false, 128, 96, 32, "Write"} //sent remaining words, 28 bytes

Burst #2

First fragment : {true, 128, 64, 32, "Write"} //sent first word, 4 bytes
 Second fragment: {false, 128, 64, 32, "Write"} //sent remaining words, 28 bytes

Burst #3

First fragment : {true, 128, 32, 32, "Write"} //sent first word, 4 bytes
 Second fragment: {false, 128, 32, 32, "Write"} //sent remaining words, 28 bytes

Burst #4

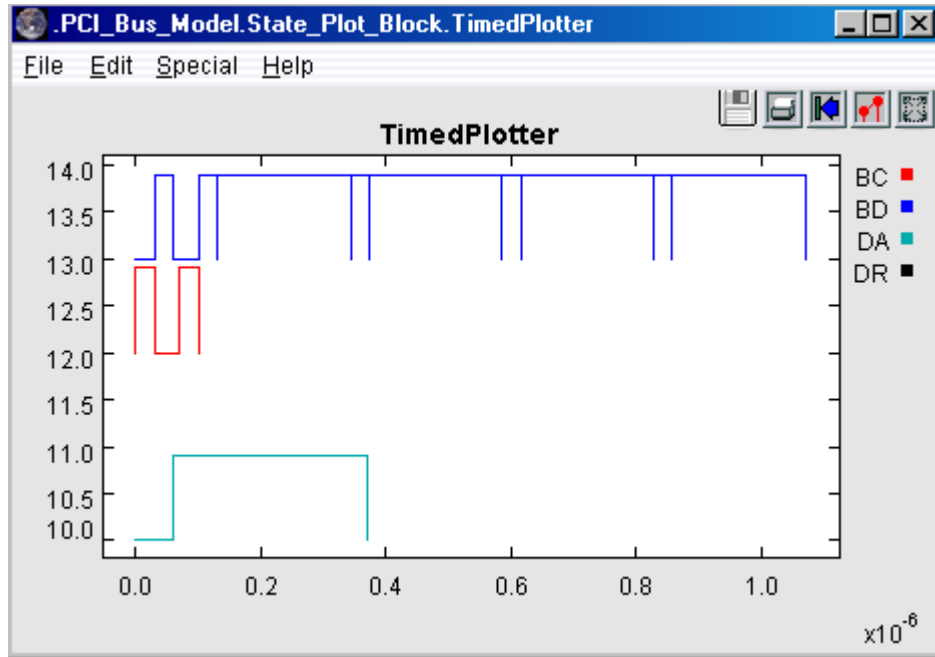
First fragment : {true, 128, 0, 32, "Write"} //sent first word, 4 bytes
 Second fragment: {false, 128, 0, 32, "Write"} //sent remaining words, 28 bytes

Read Transaction:

The bus initiated Read transfer by start transferring the Address/Control (**BC signal**) to target.

{A_First_Word, A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Address/Control phase: {true, 128, 96, 32, "Write"} // PCI_Bus sent address, 4 bytes to SDRAM_1



Data Phase:

The SDRAM_1 sent requested data to the bus block setting the field A_Command as "Write". The bus starts returning data fragments (**BD signal**) to the Initiator.

Fragmentation on Read data:

First fragment	: {true, 128, 96, 32, "Write"}	//sent first word, 4 bytes
Second fragment:	{false, 128, 96, 32, "Write"}	//sent remaining words, 28 bytes
First fragment	: {true, 128, 64, 32, "Write"}	//sent first word, 4 bytes
Second fragment:	{false, 128, 64, 32, "Write"}	//sent remaining words, 28 bytes
First fragment	: {true, 128, 32, 32, "Write"}	//sent first word, 4 bytes
Second fragment:	{false, 128, 32, 32, "Write"}	//sent remaining words, 28 bytes
First fragment	: {true, 128, 0, 32, "Write"}	//sent first word, 4 bytes
Second fragment:	{false, 128, 0, 32, Write}	//sent remaining words, 28 bytes

Arbitration Schemes

The PCI_Bus block is designed to have more than one master device. If several devices may require the use of the PCI bus to perform a data transfer at the same time, the PCI arbiter

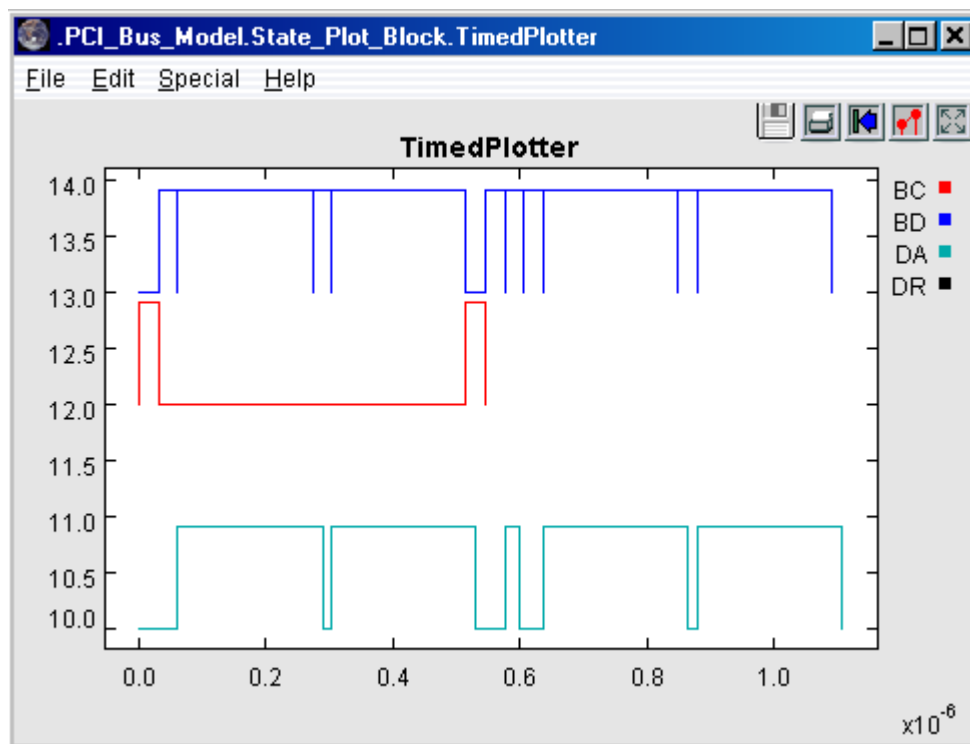
effectively control these devices in the listed arbitration schemes in *chapter 4, Architecture Modeling*.

- FCFS
- Custom Arbitration

2.1.10 Preemption while stepping in progress

When the PCI bus arbiter grants the bus to a bus master, the address fragment is started to transfer immediately. During the address out transfer, if the arbiter detects a request from a higher-priority master, it can remove the grant from the first master before start fragmenting it's data. The PCI_Bus arbiter simultaneously remove one master's grant and assert another's during the same clock cycle if a transaction is in progress. It is a rule that the arbiter cannot deassert one master's grant and assert grant to another higher priority master during the last word data transfer.

The bus received a request from Master1 ("IO_1") starts transferring the data (A_Bytes = 128). The bus operation is "Write" (IO_Command = "Write"). The priority of device "IO_1" is 1 (IO_Priority = A_Priority). The priority by default is 0.



The bus received a higher priority request from Master2 ("IO_2") while transferring the second burst data fragments. The priority of "IO_2" is 2 and A_Bytes = 8. So the bus deasserts the grant from "IO_1" and start processing the higher priority transaction of "IO_2" before completing the current transaction. After finished processing higher priority transaction bus started to process the remaining fragments of preempted transaction and other transactions if any based on the priority.

2.2 PCI-Express

PCI Express (PCIe) provides a scalable, high-speed, serial I/O bus that maintains backward compatibility with PCI applications and drivers. The PCI Express layered architecture supports existing PCI applications and drivers by maintaining compatibility with the existing PCI model. PCI Express having parallel bus topology and Multiple point-to-point connections. A switch may provide peer-to-peer communication between different endpoints and this traffic. A PCI Express link consists of dual simplex channels, each implemented as a transmit pair and a receive pair for simultaneous transmission in each direction. Each pair consists of two low-voltage, differentially driven pairs of signals. A data clock is embedded in each pair, using an 8b/10b clock-encoding scheme to achieve very high data rates.

2.2.1 Sample Model

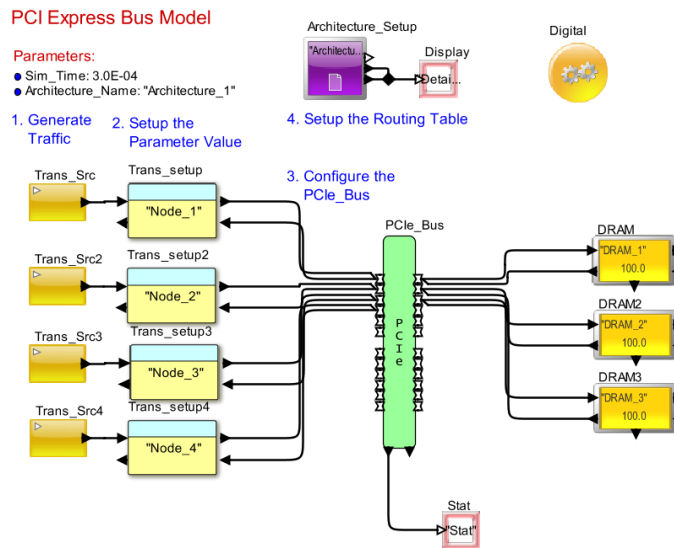


Figure 2-1 PCIe Bus Model

2.2.2 Setup

To include the PCIe block in a model, the following steps must be followed:

- PCIe Bus is available in Interfaces and Buses → PCI -> PCIe_Bus.
- Set the block parameters to match your block diagram. Refer to the Parameter selection.
- Connect each Master to the multi-ports on left-side. Connect all Slaves to the right-side. Make sure you connect the input to the PCIe first and then the output. All connections must be made to a Hierarchical block or input/output must be to a single block.
- The PCIe Bus expects the Data Structure, Processor_DS, to be used. If you are using any other Data Structure template, make sure the fields are mapped appropriately. Look at Section 1.4 for a listing of all the fields used within the PCIe block.

2.2.3 Model Parameters, Data Structure Fields and Ports

Parameter Name	Value (Data Type)	Explanation
Architecture_Name	"Architecture_1" (String)	Name of the Architecture_Setup block
Bus_Name	"PCIe_1" (String)	Unique name for this Bus. Different from all architecture blocks and global model memories.
PCIe_Channel_MHz	100.0 (Double)	Bus Speed
Number_of_Lanes	16 (Int)	Total Number of Data Lane
Slave_Buffer	512 (Int)	Buffer in bytes at each Slave interface. This is used when Flow Control with the SLave is enabled. In this case the incoming payload is placed in the Bufer until it receives an EVENT from the Slave device.
Master_Buffer	512 (Int)	Buffer in bytes at each Master interface. The Master buffer stores the payloads and the requests. When the Slave is not busy, the Request is sent out. When the Request is acknowledged, the payload is sent out.
Header_Bytes	16 (Int)	Overhead bytes. If not proprietary, DO NOT MODIFY.
Number_of_Ports	{12,12}	Two index array with the list of Master and Slave ports
BER	1.0E-11, Double	Error rate that will cause a retry between the End Point and the Root Complex
Max_Payload_Size	64 (Int)	Write data transfer size .
Max_Payload_Req_Size	64 (Int)	Size of the Read requests
PCIe_Gen_1	250.0 (Double)	PCIe Gen1 Transfer Rate
PCIe_Gen_2	500.0 (Double)	PCIe Gen2 Transfer Rate
PCIe_Gen_3	985.0 (Double)	PCIe Gen3 Transfer Rate
PCIe_Gen_4	1969.0 (Double)	PCIe Gen4 Transfer Rate
Read_to_Write_Ratio	0.5 (Double)	Range is 0.00001-1.0. If 0.0, the value is ignored. This is the ratio between the Reads and Writes sent out of a Master. This simply attempts to maintain a average ratio between Read and Write. This will work correctly if there are sufficient Reads and Writes to maintain the ratio.
Devices_Attached_to_Slaves	{{"DRAM_1"}, {"DRAM_2"}, {"DRAM_3"}, {"Dev_4"}, {"Dev_5"}, {"Dev_6"}, {"Dev_7"}, {"Dev_8"}, {"Dev_9"}, {"Dev_10"}, {"Dev_11"}, {"Dev_12"}}	Array of arrays. This is a list of slave devices that are downstream from this port. The order must match the port order.

Devices_Attached_to_Slave_Port	{"DRAM1","DRAM2"} (Array of Strings)	List of slave devices that have to pass through the Slave port 1
Devices_Attached_to_Slave_Port2	{"DRAM1","DRAM2"} (Array of Strings)	List of slave devices that have to pass through the Slave port 2
Devices_Attached_to_Slave_Port3	{"DRAM1","DRAM2"} (Array of Strings)	List of slave devices that have to pass through the Slave port 3
Root_Complex_Flow_Control	{false,false,false,false,false,false,false,false,false,false,false,false}	Array of booleans that corresponds to all theRoot Complex ports. If true, there is a Flow Control with the Master device.The order of the booleans correspond to the order of the ports
Endpoint_Flow_Control	{false,false,false,false,false,false,false,false,false,false,false,false}	Array of booleans that corresponds to all the End Point ports. If true, there is a Flow Control with the Slave.The order of the booleans correspond to the order of the ports.
DEBUG	false (Boolean)	If set to true, the lower port outputs the current time, the active transaction and the channel.
Sim_Time	1.0 (Double)	This is simulation time and match the top-level Digital Simulator
Bit_64_Mode	True	This is for addressing and header sizing only.

Table 6 List of PCIe Bus Block Parameters

Data Structure Field	Value (Data Type)	Explanation
A_Bytes	128 (Int)	Total bytes to be transferred
A_Bytes_Remaining	120 (Int)	Remaining bytes to transfer after current transaction
A_Bytes_Sent	8 (Int)	Bytes currently transferred and will equal the Width
A_Command	"Read" or "Write" (String)	Bus Operation. Internal activity will breakdown to Request, Address, Data Channel and Acknowledge
A_Message	Used internal to the Bus (String). No user setting required and does not need to exist in the incoming Data Structure	Indicate the type of transaction (Request, Response, Data and ACK)
A_Priority	1 (Int)	Transaction Priority. Used for Traffic Class.
A_Source	"Node_1" (String)	Master Name
A_Destination	"DRAM_1" (String)	Final Destination
A_Hop	"DRAM_1" (String)	Set to match A_Destination

Table 7 List of Data Structure Fields used in the PCIe Bus

<i>Port Name</i>	<i>Type</i>	<i>Explanation</i>
Input, input2, input3, input4	Multiport or input/output. Connection must be made in order- first input to this port and then output away from this port.	Each port is for one Master. Must be a Transaction (Data Structure of type Processor_DS)
output1, output2, output3	Multiport or input/output. Connection must be made in order- first input to this port and then output away from this port.	Each port is for one Slave. Must be a Transaction (Data Structure of type Processor_DS)
Port	Output port	Statistics (Data Structures) and debugging (String) information.

Table 8 List of PCIe Bus Ports

2.2.4 Traffic Queue Depths

The length of the Traffic queues can be used to throttle the transmission of transactions into the PCIe bus. The DRAM Status also in memory arrays.

Memory Name	Value	Explanation
Max_Q_Value	{100, 500, 350, 400, 462, 500, 512, 600, 700, 800}	Used for Traffic Class, the Maximum Queue Status is setup before running the model.
Current_Q_Value	{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}	Used for Traffic Class, the Current Queue Status is updated and verified with the Maximum Queue value before sending the Request
DRAM_Status	{0, 0, 0}	DRAM Status can be updated while sending the Request

Table 9 Queue Depth Array Memory Locations

These are local memories inside the PCIe block. To access these, import into Virtual_Machine using X =

readMemory("Model_Name.Hierarchical_Name.PCIe_Block_Name.Current_Q_Value"). This will create a reference to the array. Define this in the initial section. Remember that array index starts from 0. To get the memory path, you can use the RegEx function called readAllMemory. This will provide the complete path.

2.2.5 PCIe Bus Description

The fundamental PCI Express link consists of two, low-voltage, differentially driven pairs of signals: a transmit pair and a receive pair as shown in Figure 1.2 . A data clock is embedded using the 8b/10b encoding scheme to achieve very high data rates. The physical layer transports packets between the link layers of two PCI Express agents.

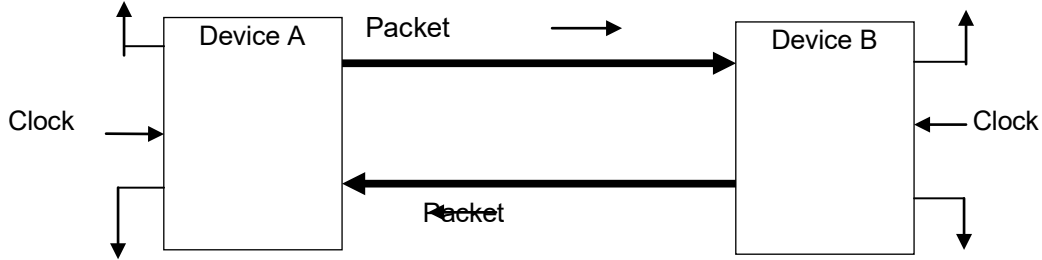


Fig. 1.2 A PCI Express link uses transmit and receive signal pairs

The bandwidth of a PCI Express link may be linearly scaled by adding signal pairs to form multiple lanes. The physical layer supports x1, x2, x4, x8, x12, x16 and x32 lane widths and splits the byte data. Each byte is transmitted across the lane(s). This data disassembly and re-assembly is transparent to other layers.

During initialization, each PCI Express link is set up following a negotiation of lane widths and frequency of operation by the two agents at each end of the link. The PCI Express architecture comprehends future performance enhancements via speed upgrades and advanced encoding techniques. The future speeds, encoding techniques or media would only impact the physical layer.

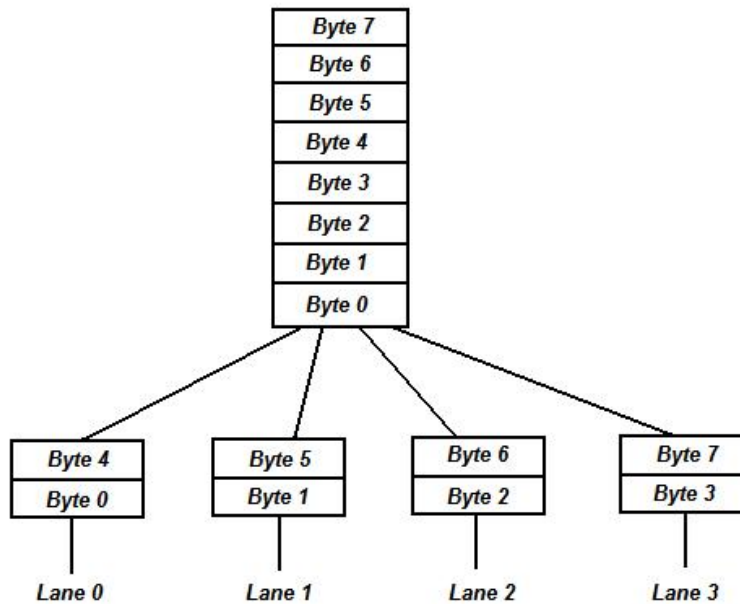


Fig. 1.3 A PCI Express Link consists of one or more lanes

2.2.6 Bus Statistics

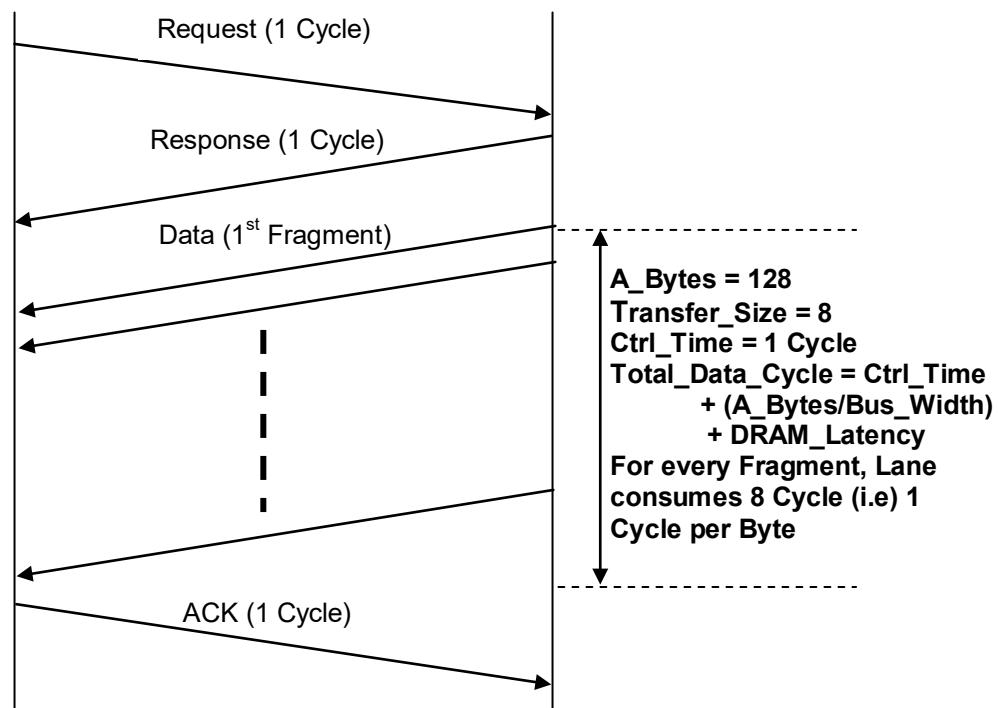
Statistics collected for the PCI Express Bus

- Throughput in Mbps
- Utilization percentage
- Input Output transactions per sec (IOs_per_second)
- Input buffer occupancy in words

2.2.7 Latency Flow

The Latency for the Read and Write operation are derived as below.

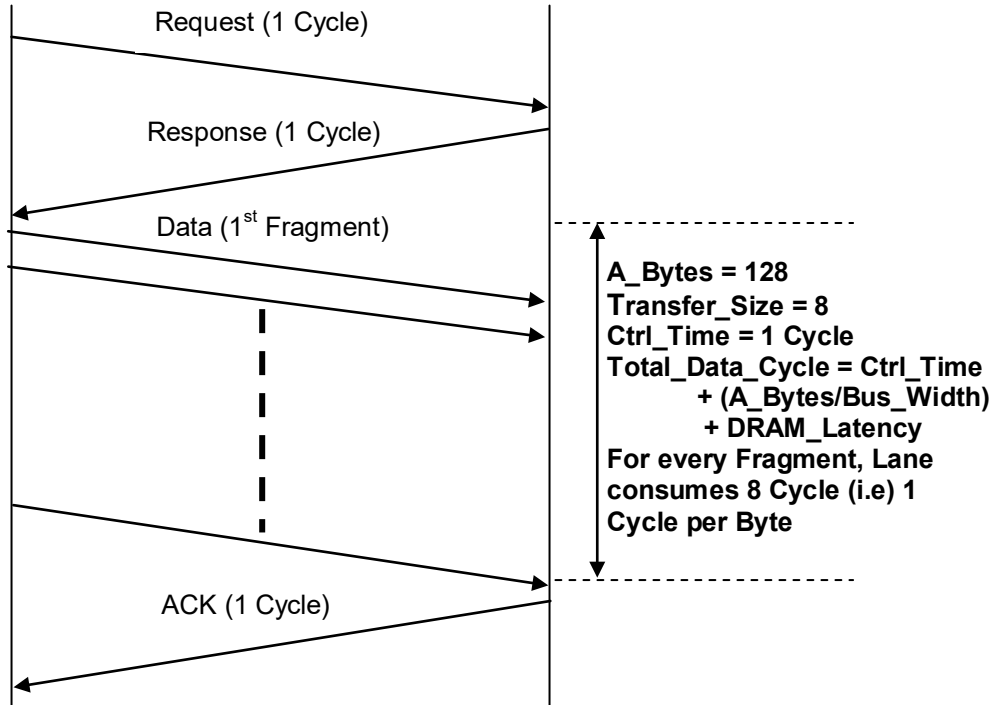
Read Command



Total Latency for the whole READ operation = 28 Cycle

Note: Cycle overhead due to Slave = Number of Fragment * (Slave_Width / Slave_Speed) * (Bus_Speed / Bus_Width)

Write Command



Total Latency for the whole WRITE operation = 28 Cycle

2.2.8 Operational View of the Model

The traffic pumps in transactions through the PCIe_Bus Block along with “Hello” Message from their respective nodes. The Processor_DS determines the type of Transaction – Read or a Write, the Size of bytes to be transferred, the source and destination blocks and A_message to denote whether the transmitted is Request or Data or Acknowledgment from the source. On receiving a DS from the source, the PCIe Bus send the Request through the x16 Lane to the slave. After getting the Response, the data are fragmented pass it through the x16 Lane Channel as per the Byte Stream concept to the respective Slave. PCIe Bus model will transfer the Data point to point to the Slave through the x16 Lane Channel.

A Write transaction is processed as follows.

1. The address/control signal (Write request) is transferred through the bus in one cycle
2. The data fragments are transferred through the bus taking several cycles, depending on the number of Bytes transferred, etc.
3. The slave now takes a cycles to write the data, returns the Write Response.

Data Fragmentation – Write Transaction

{A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Request Signal:

First fragment: {128, 120, 8, Write} //sends Request, 8 bytes

Data Transfer:

The 128 bytes to be transferred are sent in 16 burst operations **through the x16 Lane Channel** as byte by byte.

First fragment: {128, 120, 8, Write} //sends first word, 8 bytes
Remaining fragment: {128, 0, 120, Write} //sends remaining 15 words, 8 bytes

A Read transaction is processed as follows.

- The address/control signal (Read request) is transferred through the bus in one cycle.
- The slave now responds by performing the data Read and returns the data fragments setting the A_Command as 'Write' so the bus would return the data to the master.
- The data fragments are transferred to the master through the bus taking several cycles, depending on the bytes transferred etc.

Data Fragmentation – Read Transaction

{A_Bytes, A_Bytes_Remaining, A_Bytes_Sent, A_Command}

Request Signal:

First fragment: {128, 120, 8, Write} //sends Request, 8 bytes

Data Transfer:

The slave DRAM reads and returns the data fragments setting the A_Command as 'Write'. The 128 bytes to be transferred are sent in 16 burst operations **through the Channel** as Byte by Byte.

3 CoreConnect Bus

3.1.1 Introduction

The **Processor Local Bus (PLB)** protocol is a high performance bus for interconnecting processor, memory subsystems, and high bandwidth peripherals. There are IBM and Xilinx versions of the bus that match specific timing for reads and writes. The CoreConnect Bus hierarchical block supports four master and two slave ports with bi-directional ports.

This model sends two read commands from Master (2) to Slave (1) and two more read commands from Master (2) to Slave (1).

Parameters.

- Architecture_Name: "Architecture_1"
- Bus_Name: "Bus_1"
- Architecture_Bus_Name: Architecture_Name + "_" + Bus_Name
- SDRAM_Speed_Mhz: 500.0
- Bytes_per_Xfer: 100

Single Read @ 0.0

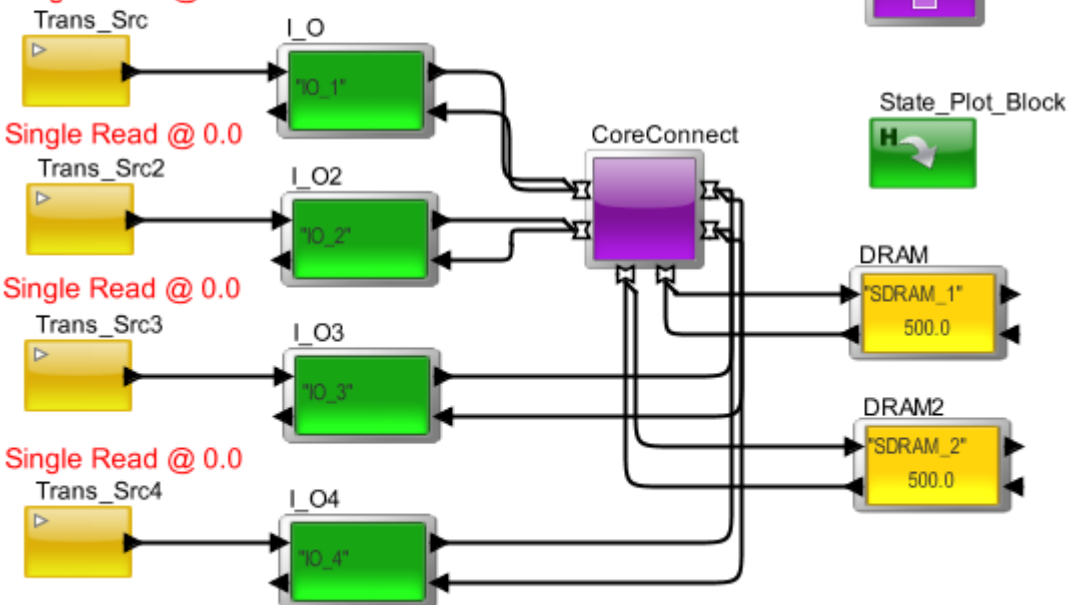


Figure 3-1 VisualSim model of a system using the CoreConnect bus

3.1.2 Setup

The CoreConnect hierarchical block requires certain Model Parameters and a specific Data Structure (Processor_DS) for proper use, once the CoreConnect hierarchical block has been dragged into a model window. The Sample Model on the previous page illustrates the model parameters, below is a more detailed description of their meaning, followed by a description of the Processor_DS.

3.1.3 Model Parameters

The key bus parameter settings for a Xilinx **32-bit CoreConnect bus at 266 MHz**:

- Bus_Name: "Bus_1"
- Architecture_Name: "Architecture_1"
- CoreConnect_Speed_Mhz: 266.0
- Burst_Size_Bytes: 32
- FIFO_Buffers: 8
- Request_Clock_Multiplier: 0.99

The Bus_Name must be unique with the Architecture_Name, which can be thought of as the platform domain. The Burst_Size_Bytes represents the CoreConnect Bus model ability to fragment large transactions into smaller transactions at a master or slave. FIFO_Buffers is the amount of buffering at the master or slave, based on the Burst_Size_Bytes of the largest bus fragment. The Request_Clock_Multiplier modifies the request response rate from the master to slave and slave to master. A Request_Clock_Multiplier of 0.99 represents one clock time per request transfer from master to slave port, or visa versa.

3.1.4 Data Structure: Processor_DS

The Processor_DS was selected at the best data structure to send through the CoreConnect bus, as this is the same data structure used by the Processor block, other Architectural Library busses. The following are specific data structure fields to be set prior to entering a master port.

- A_Bytes* – Total number of bytes to be transferred
- A_Bytes_Remaining* – A_Bytes (total) minus A_Bytes_Sent (bus width)
- A_Bytes_Sent* – bus width
- A_Command* – CoreConnect bus commands, see below:

<u>CoreConnect Bus Commands</u>	<u>A_Command field</u>
IO Read	"Read_IO"
IO Write	"Write_IO"
Memory Read	"Read_Memory"
Memory Write	"Write_Memory"

Any Read or Write command whether it is from an IO device or Memory is handled by the bus in a similar fashion. Hence the bus looks for A_Command that starts with the "Read_*" string to process a Read transaction and starts with the "Write_*" string to process a Write transaction. Read commands will return to the source node, Write commands will execute on the slave device and "not" return. One exception is if the slave device is a Processor block.

- A_First_Word* – default is true, not used by the CoreConnect Bus.
- A_Priority* – the priority of the transaction, higher priority gains bus access.
- A_Source* – routing source node, or transaction initiator.
- A_Hop* – represents the next node in the routing path.
- A_Status* – default OK, not used by the CoreConnect Bus.
- A_Destination* – routing destination node, or transaction target.

A sample set of data, a user can set in fields of data structure "Processor_DS":

<u>Data Structure Field</u>	<u>Data Type</u>	<u>Sample Data</u>
<i>A_Bytes</i>	int	64
<i>A_Command</i>	string	"Read_Memory"
<i>A_First_Word</i>	boolean	true
<i>A_Priority</i>	int	1

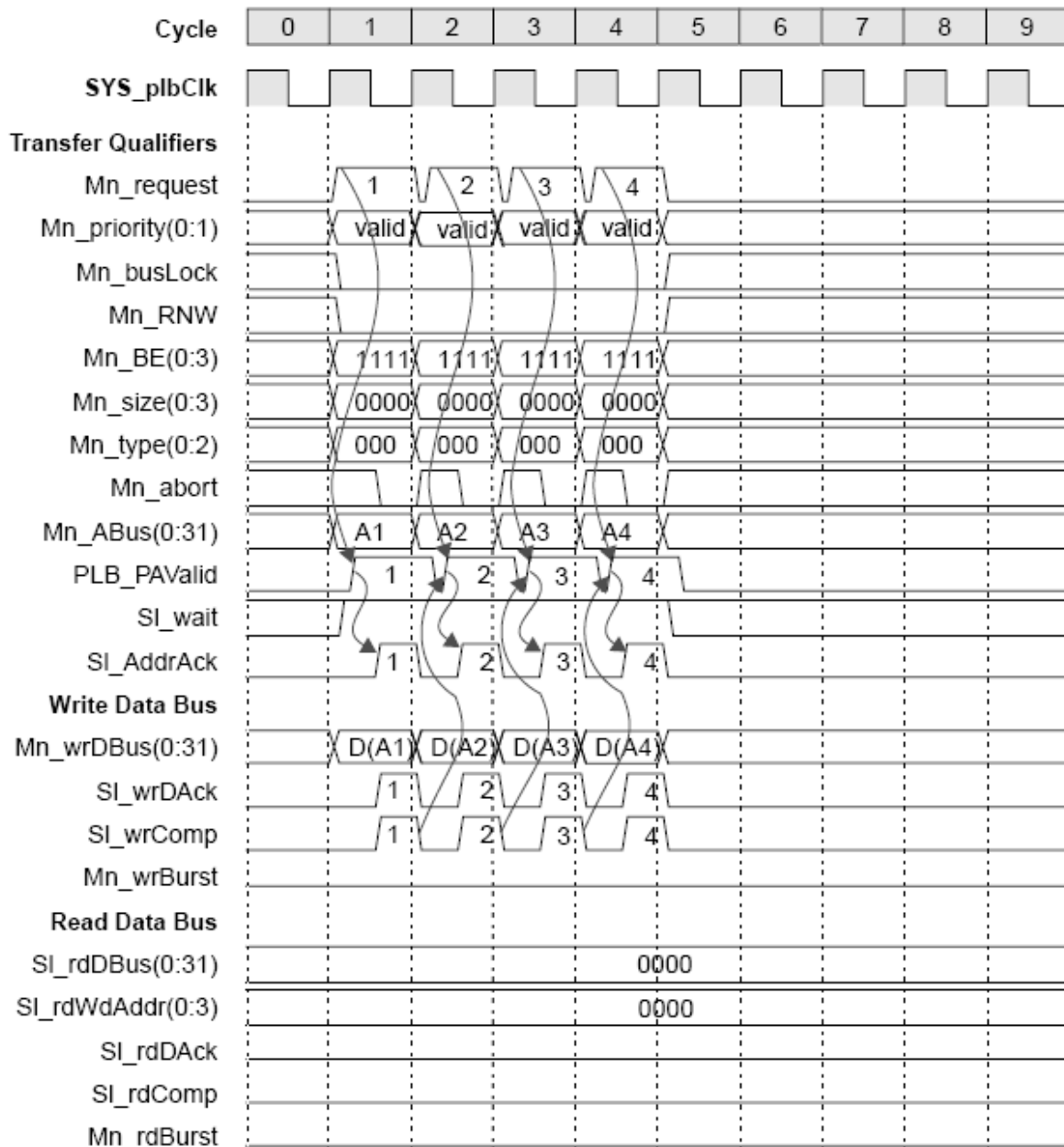
A_Source	string	"IO_1"
A_Destination	string	"SDRAM_1"

The data structure Processor_DS can be generated using the block *Transaction_Source* at given time. The block *Statement* is used to modify the fields of the generated data structure. The fields of data structure Processor_DS are also used as parameter values in block I_O.

3.1.5 Timing Diagrams

3.1.5.1 Back-to-Back Write Transfers (IBM)

IBM CoreConnect, 4 Cycles for 4 words



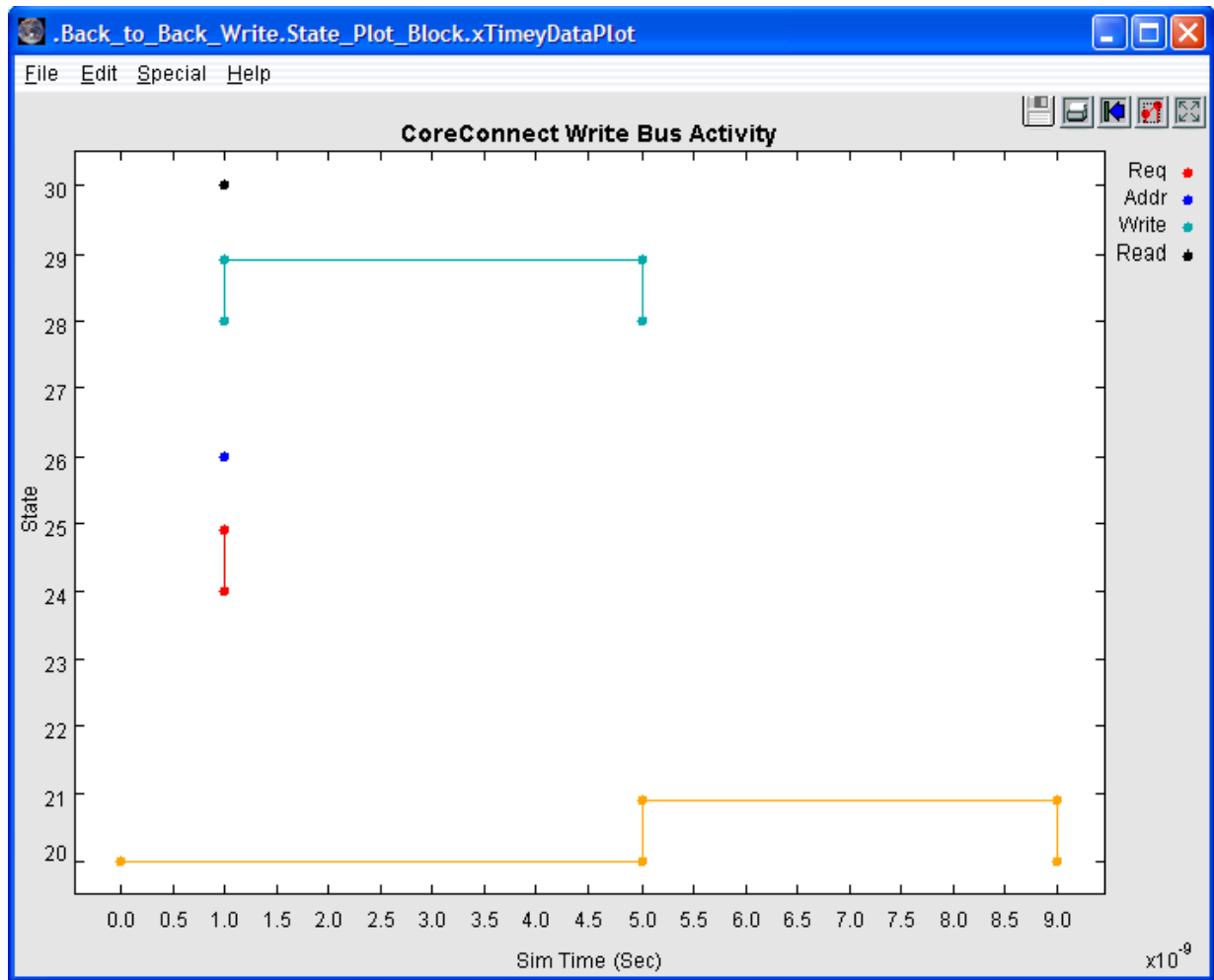
3.1.5.2 Back-to-Back Write Transfers (VisualSim)

Models IBM CoreConnect, 4 Cycles for 4 words

Request_Clock_Multiplier = 0.0

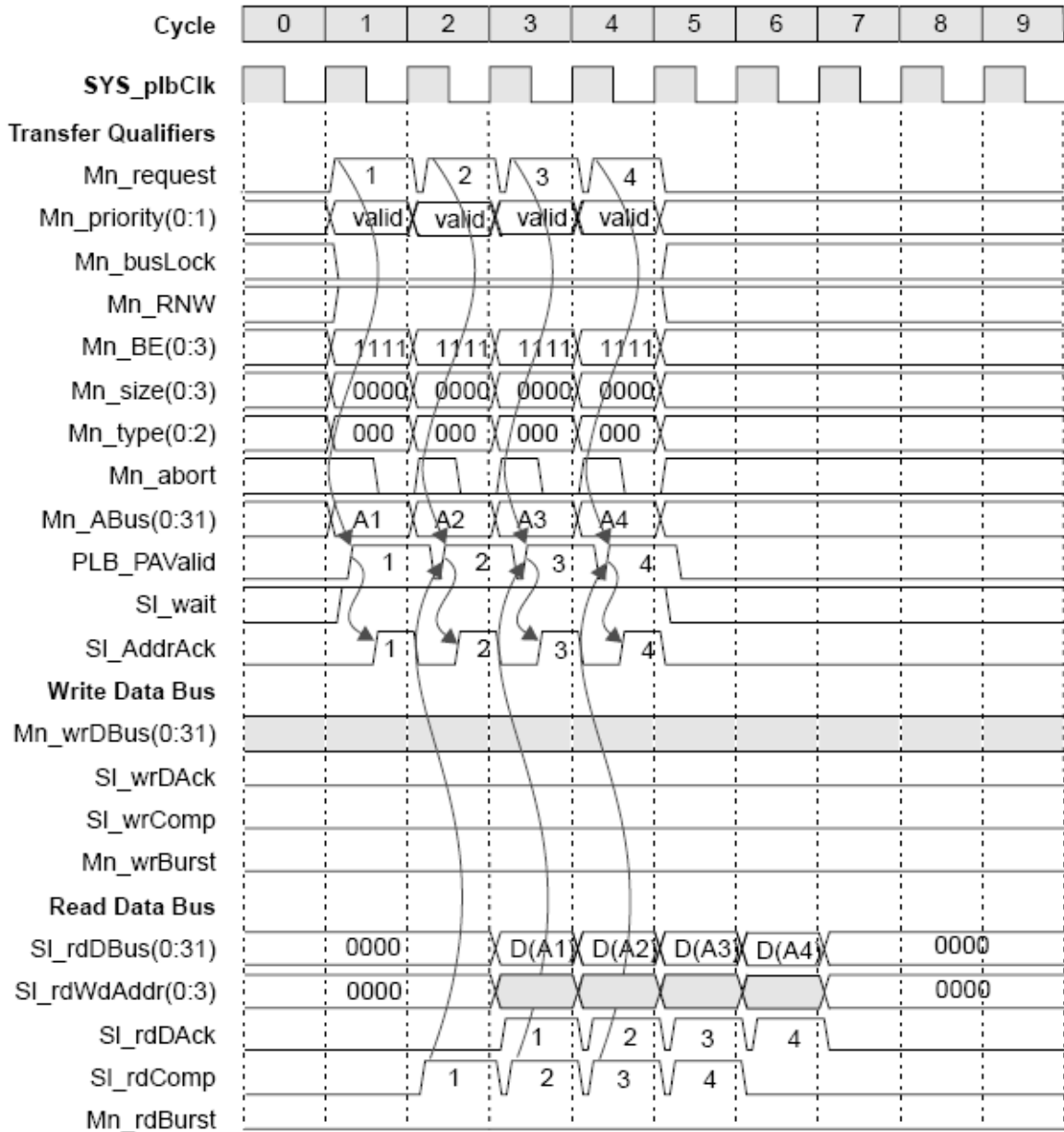
SDRAM (gold color) write shown at bottom, when bus transfer completes.

Note: To reduce SDRAM write latency, it is recommended to reduce the Burst_Size_Bytes parameter to fewer Words.



3.1.5.3 Back-to-Back Read Transfers (IBM)

IBM CoreConnect, 6 Cycles for 4 words



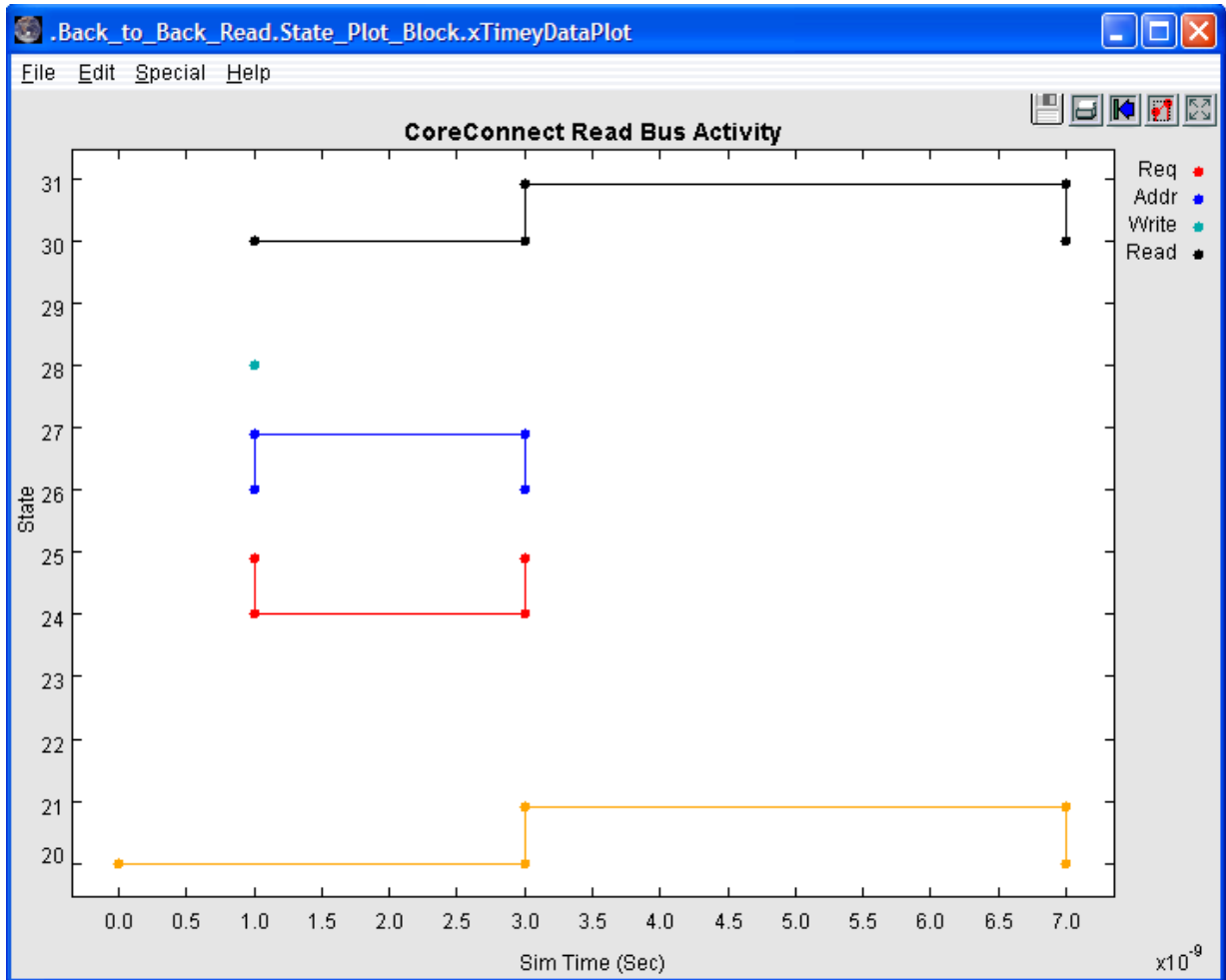
3.1.5.4 Back-to-Back Read Transfers (VisualSim)

Models IBM CoreConnect, 6 Cycles for 4 words

Address_Bytes = 8

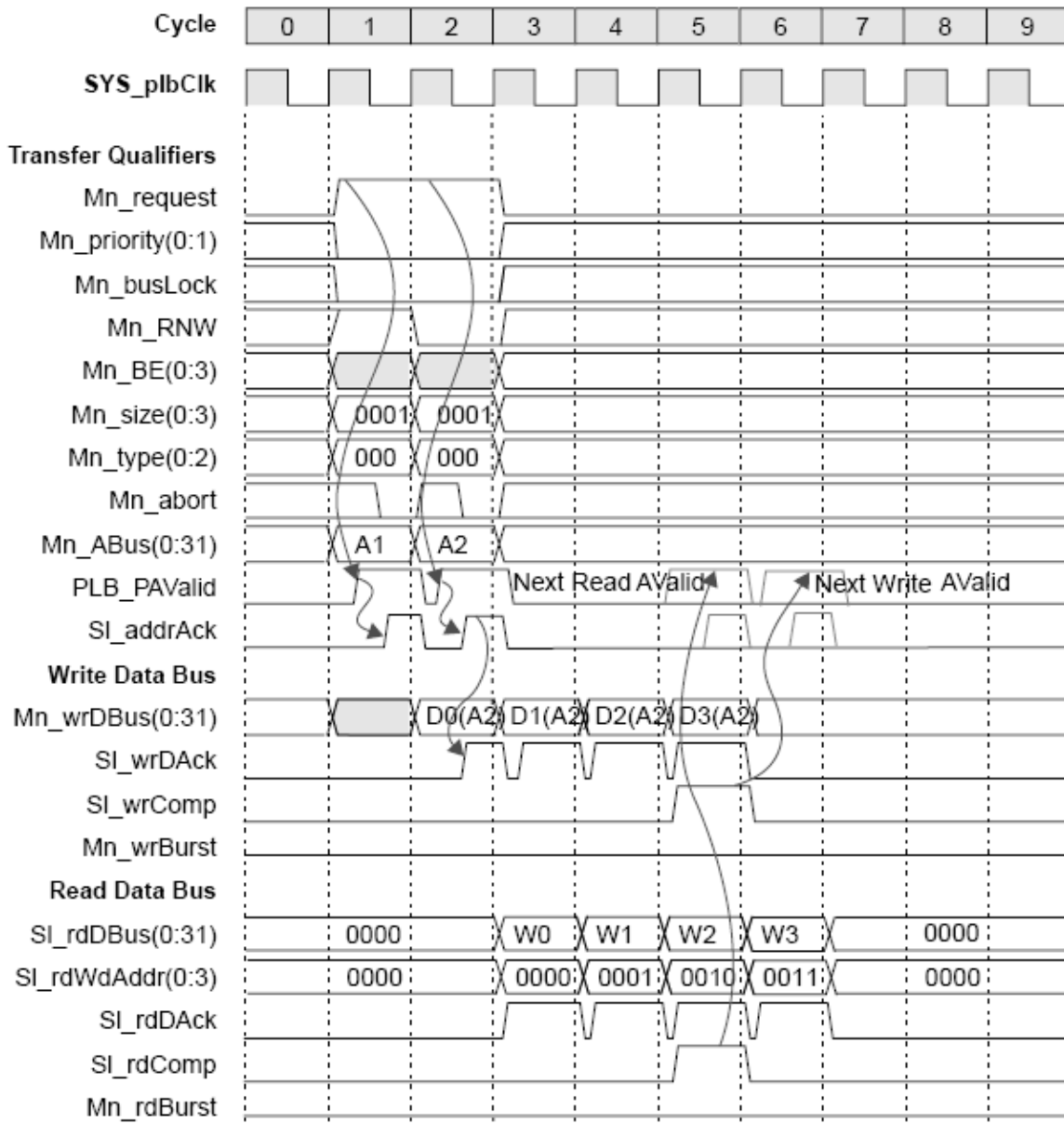
Request_Clock_Multiplier = 0.0

SDRAM (gold color) read shown at bottom, starts with completion of Address sent via Address Channel.



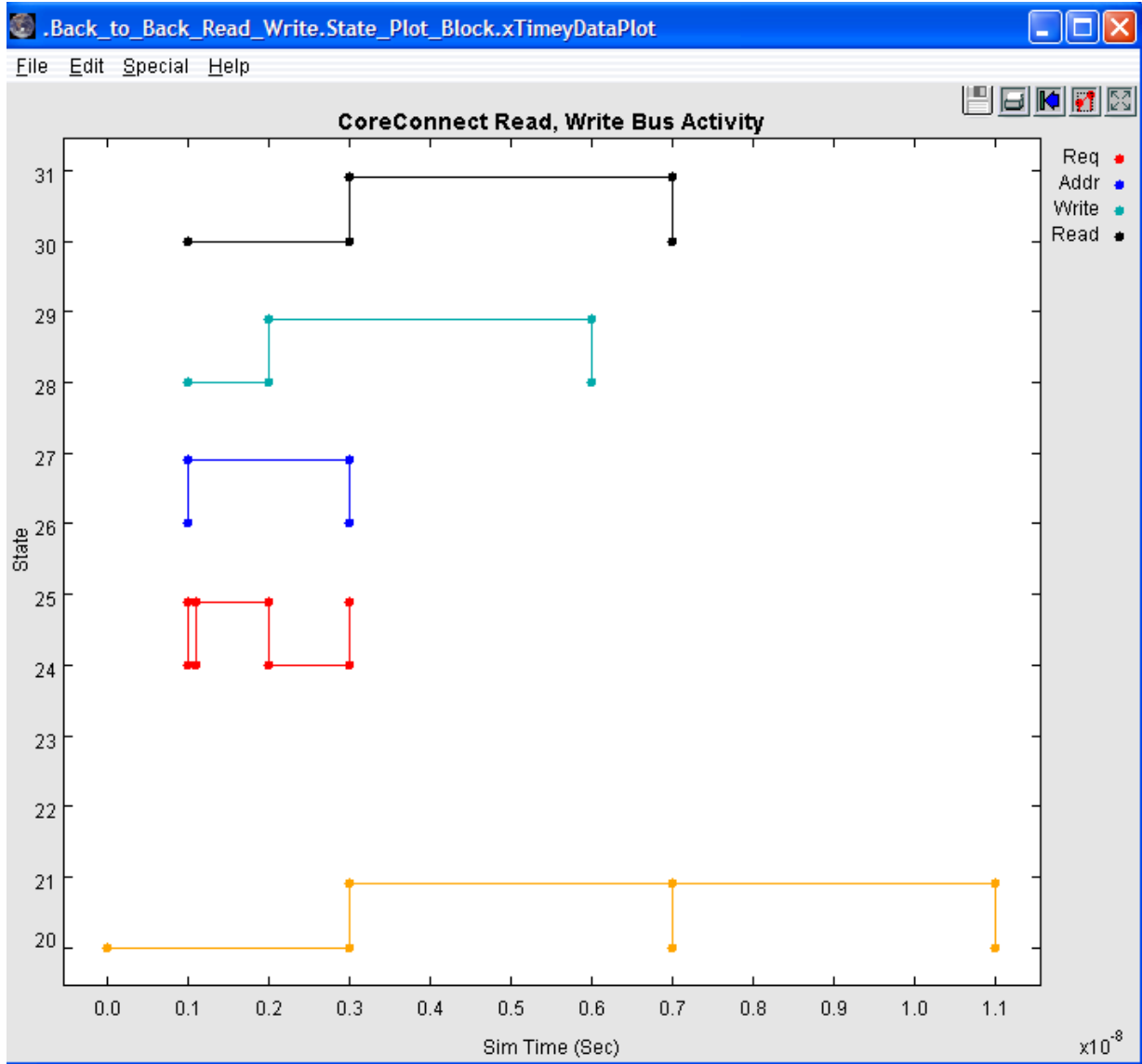
3.1.5.5 Four-word Line Read Followed By Four-word Line Write Transfers (IBM)

IBM CoreConnect, 6 Cycles for 4 word Read followed by 4 word Write



3.1.5.6 Four-word Line Write Followed By Four-word Line Read Transfers (VisualSim)

Models IBM CoreConnect, 6 Cycles for 4 word Write followed by 4 word Read
 Address_Bytes = 8
 Request_Clock_Multiplier = 0.0
 SDRAM (gold color) read and write shown at bottom, starts with completion of
 Address sent via Address Channel.



3.1.5.7 Single Read Transfer (Xilinx)

Xilinx CoreConnect, 6 Cycles for 1 word

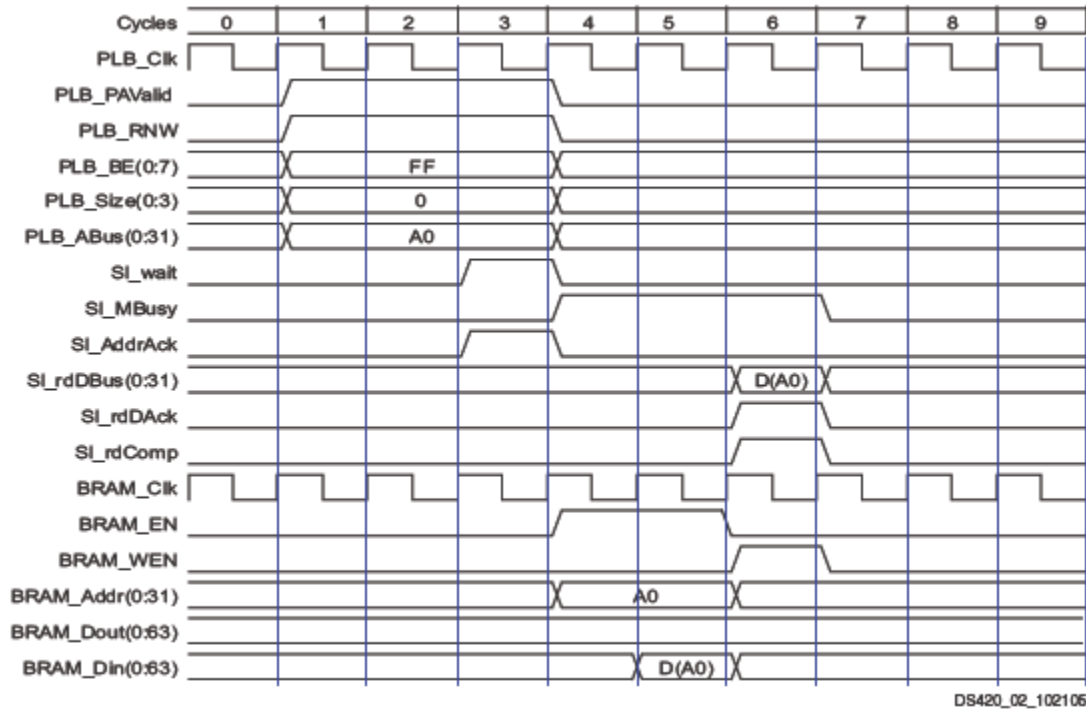
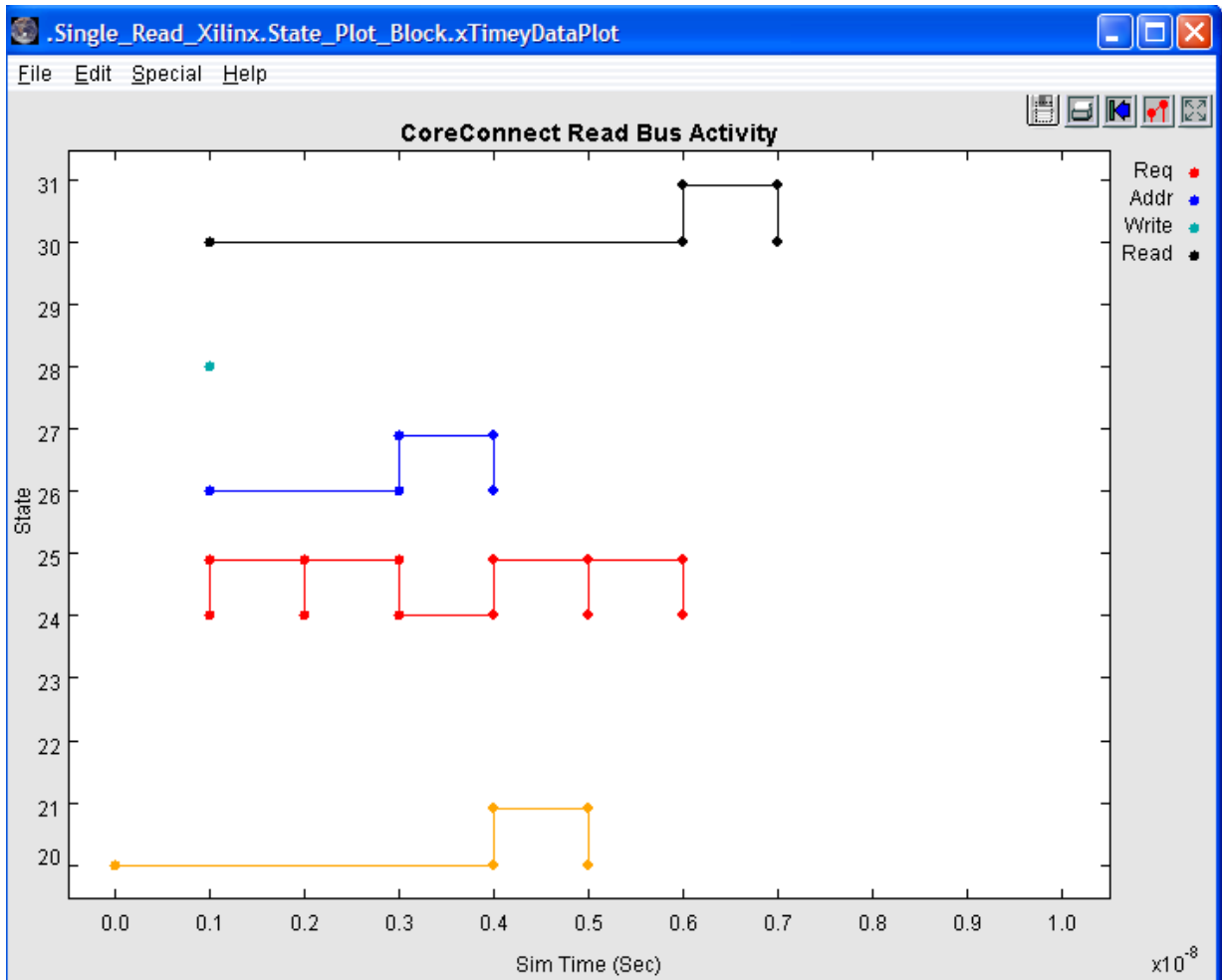


Figure 2: Single Read Transfer

3.1.5.8 Single Read Transfer (VisualSim)

Models Xilinx CoreConnect, 6 Cycles for 1 word
 Address_Bytes = 4
 Request_Clock_Multiplier = 0.99



3.1.5.9 Single Write Transfer (Xilinx)

Xilinx CoreConnect, 6 Cycles for 1 word

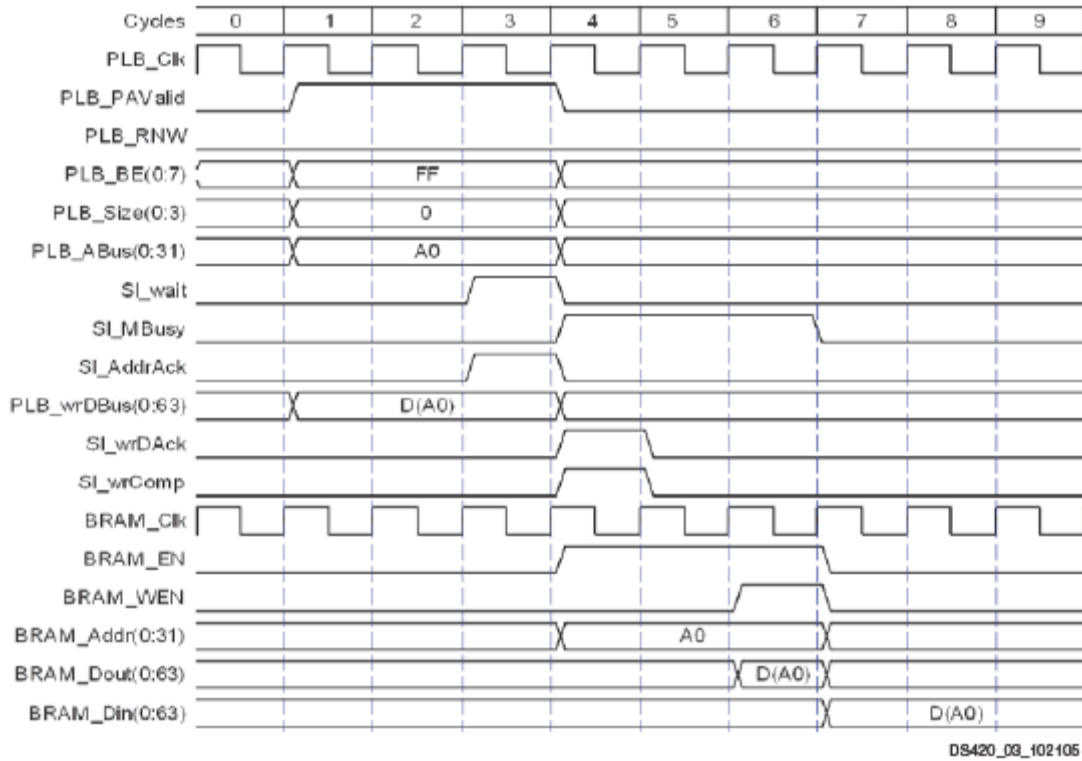
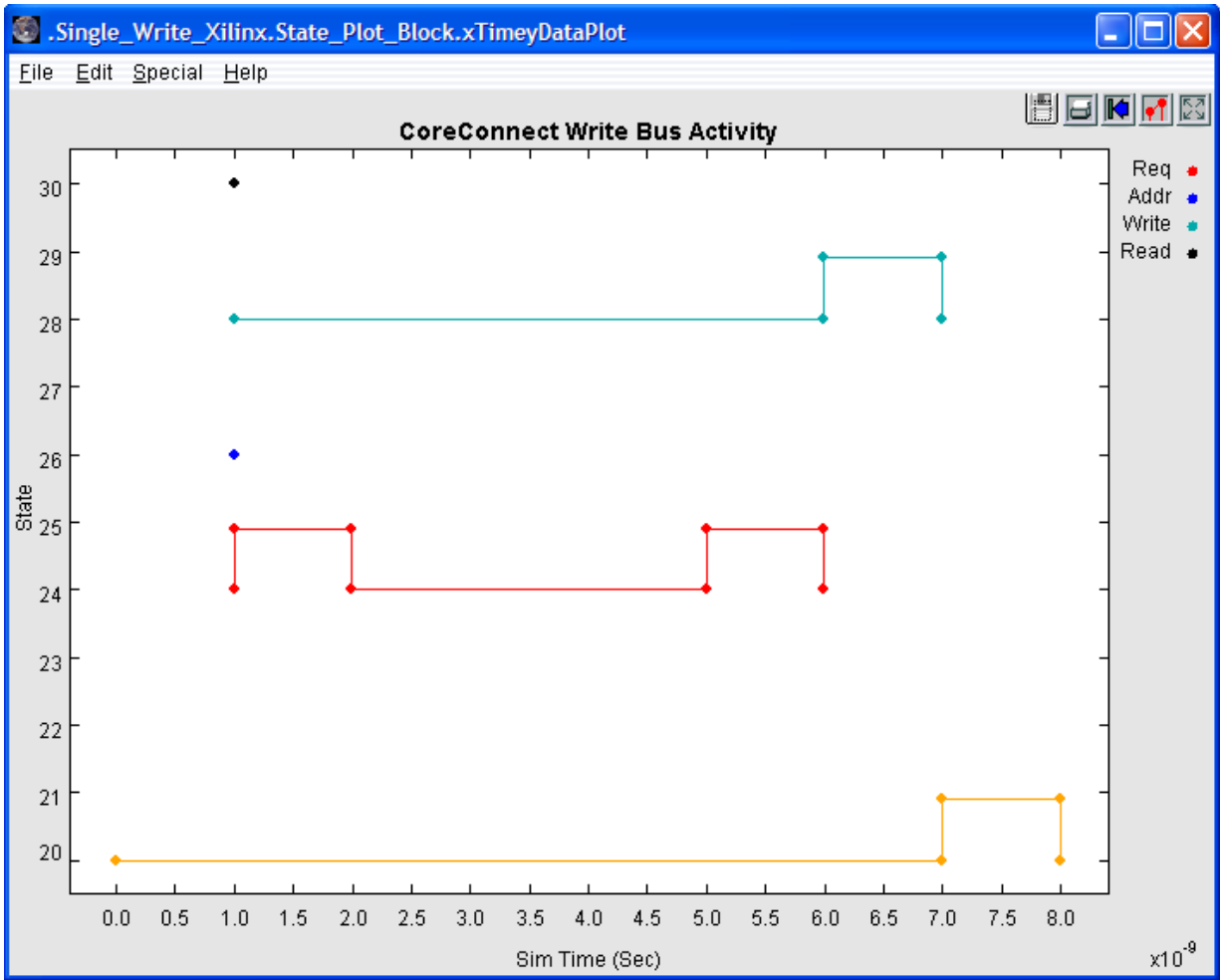


Figure 3: Single Write Transfer

3.1.5.10 Single Write Transfer (VisualSim)

Models Xilinx CoreConnect, 6 Cycles for 1 word
 Address_Bytes = 4
 Request_Clock_Multiplier = 0.99



3.1.6 Routing Table

Routing transactions between the blocks I_O, I_O2, SDRAM_1 connected to the bus block CoreConnect Bus are added into the Routing Table by the auto-routing capability within the VisualSim bus blocks. If the transactions must traverse multiple busses, or one wishes to over-ride the auto-routing, then additions can be made to the routing table:

```

/* First row contains Column Names.                                     */
Source_Node  Destination_Node  Hop          Source_Port ;
IO_1         SDRAM_1          Port_1      to_bus   ;
  
```

```

IO_2          SDRAM_1      Port_3        to_bus  ;
SDRAM_1      IO_1         Port_2        output  ;
SDRAM_1      IO_2         Port_2        output  ;

```

The third column represents the next hop in the routing, which might be the name of the bus port, I_O, Cache, DRAM, IO_Controller, Memory_Controller, or DMA_Controller. The Source_Port is the name of the port the transaction is leaving, and applies if there is more than one output port from a bus or memory block. The CoreConnect Bus has a single input and output per port, so the value of the Source_Port column is not critical to entering routing information. If any routing entries are missing, exceptions are thrown indicating the missing source and destination pair during the model execution.

3.1.7 Bus Statistics

The following statistics are collected in the same CoreConnect bus model.

- **Delay** – Transaction delay
- **IOs_per_sec** – Input Output transactions per sec
- **Node_Buffer_Occupancy_in_Words** – Input buffer occupancy in words
- **Throughput_MBps** – Throughput in Mbps
- **Utilization_Pct** – Utilization percentage

The sample statistics collected in the model are given below in min, mean, stdev, and max values for the CoreConnect bus.

```

{BLOCK = ".Req_Ack_Node_Read_Read_N_Bytes.Architecture_Setup",
Bus_1_Address_Buffer_Occupancy_in_Words_Max      = 1.0,
Bus_1_Address_Buffer_Occupancy_in_Words_Mean     = 1.0,
Bus_1_Address_Buffer_Occupancy_in_Words_Min      = 1.0,
Bus_1_Address_Buffer_Occupancy_in_Words_StDev    = 0.0,
Bus_1_Delay_Max                                  = 8.0E-9,
Bus_1_Delay_Mean                                 = 7.0E-9,
Bus_1_Delay_Min                                  = 4.0E-9,
Bus_1_Delay_StDev                                = 1.4142135623731E-9,
Bus_1_IOs_per_sec_Max                            = 6.0E6,
Bus_1_IOs_per_sec_Mean                           = 4.0E6,
Bus_1_IOs_per_sec_Min                            = 3.0E6,
Bus_1_IOs_per_sec_StDev                          = 1.309307341416E6,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Max = 0.33333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Mean = 0.33333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Min = 0.33333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Max = 0.66666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Mean = 0.66666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Min = 0.66666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_StDev = 0.0,

```



```

Bus_1_Node_7_Address_Buffer_Occupancy_in_Words_Min      = 0.333333333333333,
Bus_1_Node_7_Address_Buffer_Occupancy_in_Words_StDev    = 0.0,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_Max        = 0.666666666666667,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_Mean       = 0.666666666666667,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_Min        = 0.666666666666667,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_StDev      = 0.0,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_Max     = 0.666666666666667,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_Mean    = 0.666666666666667,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_Min     = 0.666666666666667,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_StDev   = 0.0,
Bus_1_Read_Buffer_Occupancy_in_Words_Max               = 2.0,
Bus_1_Read_Buffer_Occupancy_in_Words_Mean              = 2.0,
Bus_1_Read_Buffer_Occupancy_in_Words_Min               = 2.0,
Bus_1_Read_Buffer_Occupancy_in_Words_StDev             = 0.0,
Bus_1_Request_Buffer_Occupancy_in_Words_Max            = 4.0,
Bus_1_Request_Buffer_Occupancy_in_Words_Mean           = 4.0,
Bus_1_Request_Buffer_Occupancy_in_Words_Min            = 4.0,
Bus_1_Request_Buffer_Occupancy_in_Words_StDev          = 0.0,
Bus_1_Throughput_MBs_Max                               = 112.0,
Bus_1_Throughput_MBs_Mean                              = 112.0,
Bus_1_Throughput_MBs_Min                               = 112.0,
Bus_1_Throughput_MBs_StDev                             = 0.0,
Bus_1_Utilization_Pct_Max                              = 5.79,
Bus_1_Utilization_Pct_Mean                             = 5.79,
Bus_1_Utilization_Pct_Min                              = 5.79,
Bus_1_Utilization_Pct_StDev                            = 0.0,

```

3.1.8 Operational View of the Model

The Read Command/Read Channel and Write Command/Read Channel flows are shown on the next page.

The flow for the Master to Slave for Read Command/Read Channel starts with adding the Address to the address queue of the Master, sending a request to the Slave via the Controller. The Slave receives the request, and acknowledges the request and sends back the “acknowledge” to the Master via the Controller. Once, the “acknowledge” arrives at the Master, it sends the Address to the Slave via the Address Channel. The Slave receives the Address, and sends it to the external Memory. The external Memory returns the Read data and sends to the Request port of the Slave. The Slave then sends the memory data to the master via the Controller, until the transfer is complete. Typically, a CoreConnect transfer is less than the CoreConnect burst word size.

The flow for the Write Command/Read Channel starts with a Request/Address being sent from the Master to the Slave, via the Controller. The Slave acknowledges the request, sends back to Master, via the Controller. The Master then sends the Write Command data to the Slave, via the Controller. If the Write Command exceeds the Burst Size, then it sends a Request back to the Master to continue the Burst Write Command.

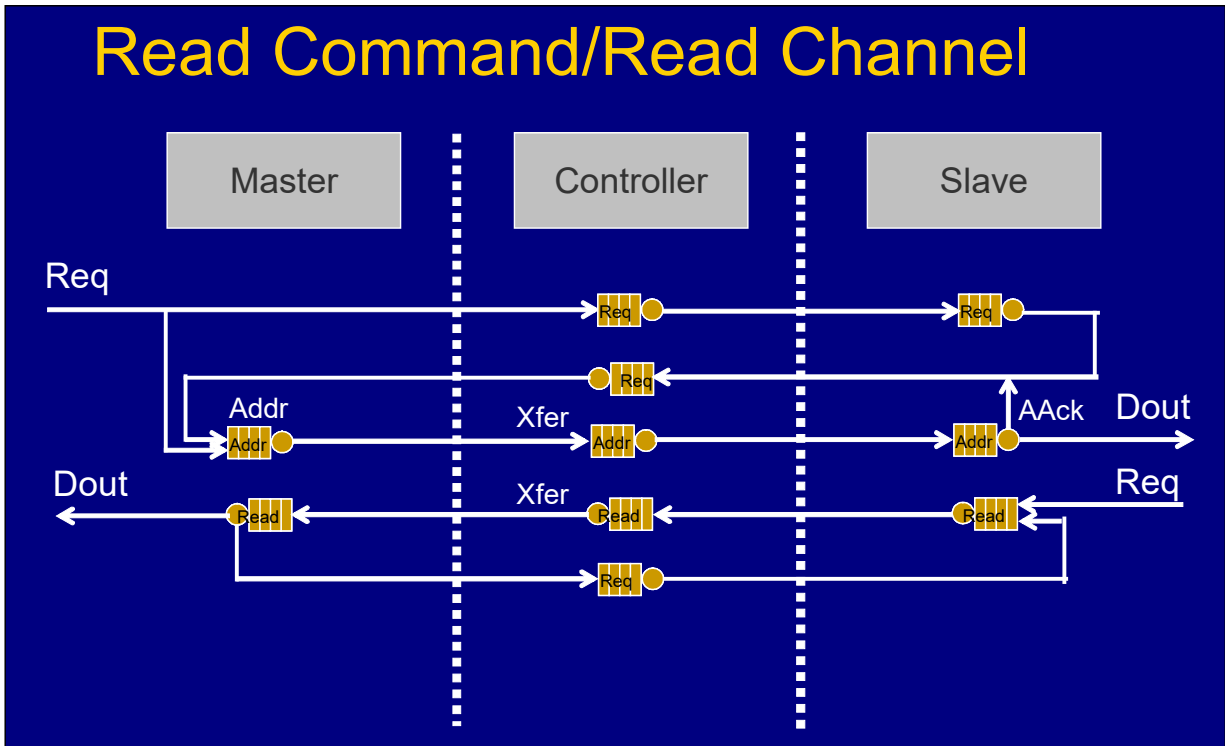


Figure 3-2 Read Command/Read Channel

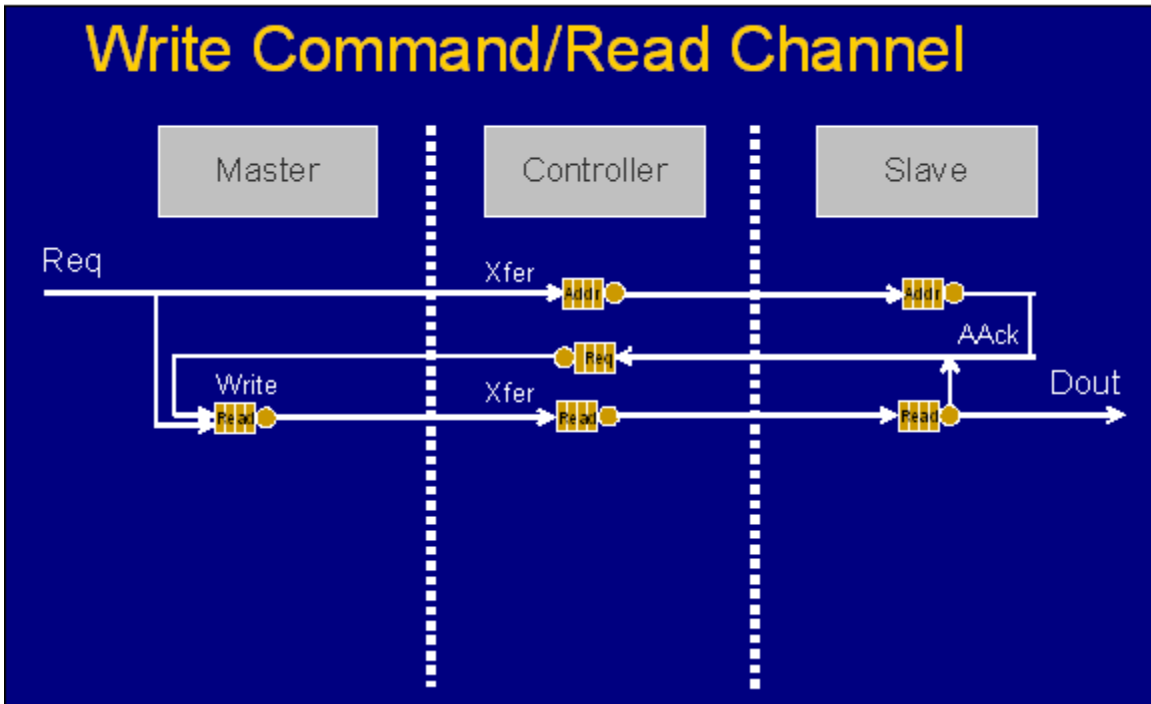


Figure 3-3 Write Channel/ Write Command

4 Switched Ethernet

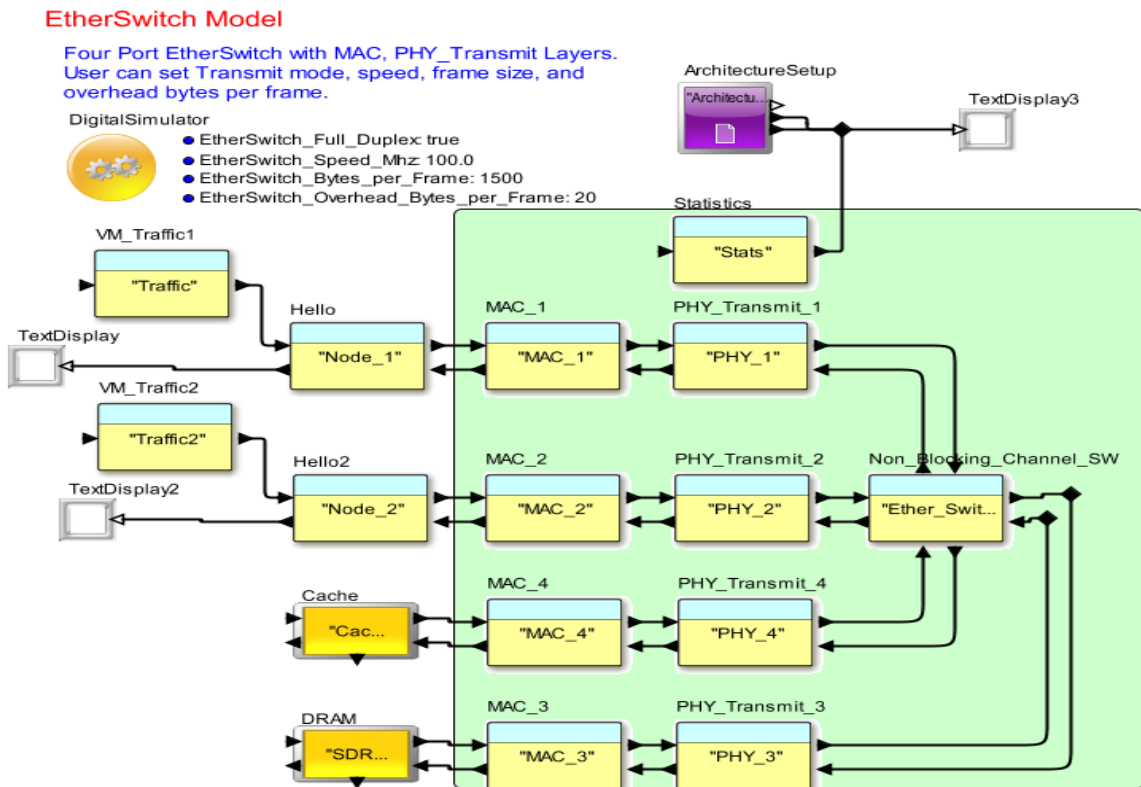
4.1.1 Introduction

Ethernet was originally based on the idea of computers communicating over a shared coaxial cable acting as a broadcast transmission medium. The common cable providing the communication channel was likened to the ether and it was from this reference that the name "Ethernet" was derived. The coaxial cable was replaced with point-to-point links connected by Ethernet hubs and/or switches to reduce installation costs, increase reliability, and enable point-to-point management and troubleshooting.

4.1.2 Model Description

The traffic pumps in transactions through the Ethernet Hierarchical Block along with "Hello" Message from their respective nodes. The Processor_DS determines the type of Transaction – Read or a Write, the Size of bytes to be transferred, the source and destination blocks and A_message to denote whether the transmitted is Request or Data or Acknowledgment from the source. The "Hello" messages are transmitted to the Destination node through the EtherSwitch Hierarchical Block. On receiving a DS from the node, the MAC_Layer verify the medium condition and the transmission mode. After fragmenting the frame, the MAC passes it to the PHY_Layer. The Switches perform as multi port Bridge.

4.1.3 Sample Model



4.1.4 Setup

To Run the Model, the Model Parameters and the transaction data structure flowing into the Ethernet_Hierarchical Block (ex: Processor_DS) etc need to be setup.

4.1.5 Initialize the Processing Data Structure

The following are sample data structure fields to be set before a data transfer.

A_Bytes – total no. of bytes to be transferred

A_Bytes_Remaining – bus block sets the field with remaining no. of data after every transfer

A_Bytes_Sent – bus block sets the field with no. of data transferred

A_Command – represent bus command,

Any Read or Write command whether it is from any node it is handled by the Ethernet Hierarchical Block

A_Priority – set to zero.

A_Source – an initiator.

A_Hop – bus block sets and use the fields for internal routing table.

A_Status – bus block sets and use the fields for internal routing table.

A_Message – indicate the whether transmitted frame is data or request

A_Destination – a target.

A sample set of data, a user can set in fields of data structure “Processor_DS” are

Data Structure Field	Data Type	Sample Data
A_Bytes	Int	128
A_Command	String	“Read_Memory”
A_Message	String	Request
A_Priority	Int	0
A_Source	String	“Block_Name”
A_Destination	String	“Cache_1”
EtherSwitch_Full_Duplex	Boolean	True
EtherSwitch_Speed_Mhz	Int	100.0
EtherSwitch_Bytes_per_Frame	Int	1500

4.1.6 A Typical Ethernet Model

Ethernet is fundamentally a shared technology where all users on a given LAN segment compete for the same available bandwidth IEEE 802.3 logical layers and their relationship to the OSI reference model. As with all IEEE 802 protocols, the ISO data link layer is divided into two IEEE 802 sub layers, the Media Access Control (MAC) sublayer and the MAC-client sublayer. The IEEE 802.3 physical layer corresponds to the ISO physical layer. Ethernet LANs consist of network nodes and interconnecting media. The network nodes fall into two major classes

- **Data terminal equipment (DTE)** -- Devices that are either the source or the destination of data frames. DTEs are typically devices such as PCs, workstations, file servers, or print servers that, as a group, are all often referred to as end stations.
- **Data communication equipment (DCE)**— Intermediate network devices that receive and forward frames across the network. DCEs may be either standalone devices such as

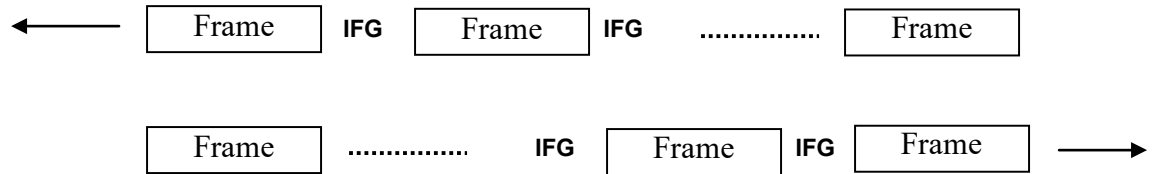
repeaters, network switches, and routers, or communications interface units such as interface cards and modems. Throughout this chapter, standalone intermediate network devices will be referred to as either *intermediate nodes* or *DCEs*. Network interface cards will be referred to as *NICs*.

The current Ethernet media options include two general types of copper cable: unshielded twisted-pair (UTP) and shielded twisted-pair (STP), plus several types of optical fiber cable.

Switches create a virtual circuit between two connected devices that want to communicate. When the virtual circuit is created, a dedicated communication path is established between the two devices. In theory this creates a collision free environment between the source and destination, which allows maximum utilization of the available bandwidth. A switch is also able to facilitate multiple, simultaneous virtual circuit connections.

4.1.7 FULL DUPLEX OPERATION

Full-duplex operation is an optional MAC capability that allows simultaneous two-way transmission over point-to-point links. Full duplex transmission is functionally much simpler than half-duplex transmission because it involves no media contention, no collisions, no need to schedule retransmissions, and no need for extension bits on the end of short frames. The result is not only more time available for transmission, but also an effective doubling of the link bandwidth because each link can now support full-rate, simultaneous, two-way transmission. Transmission can usually begin as soon as frames are ready to send. The only restriction is that there must be a minimum-length interframe gap between successive frames, as shown in Figure 2, and each frame must conform to Ethernet frame format standards.



IFG -- Inter Frame Gap

Figure 2 Full Duplex Operation Allows Simultaneous Two-Way Transmission on the Same Link

Full-duplex operation requires concurrent implementation of the optional flow-control capability that allows a receiving node (such as a network switch port) that is becoming congested to request the sending node (such as a file server) to stop sending frames for a selected short period of time. Control is MAC-to-MAC through the use of a pause frame that is automatically generated by the receiving MAC. If the congestion is relieved before the requested wait has expired, a second pause frame with a zero time-to-wait value can be sent to request resumption of transmission.

4.1.8 Operational View of the Model

The model uses one Ethernet Hierarchical model between *Node_1*, *Node_2*, *cache* and *SDRAM_1*. The Ethernet Hierarchical Model consists of *MAC_Layer*, *PHY_Layer* and



Non_Blocking_Switch. The traffic pumps in transactions through the Ethernet Hierarchical Block along with "Hello" Message from their respective nodes. The Processor_DS determines the type of Transaction – Read or a Write, the Size of bytes to be transferred, the source and destination blocks and A_message to denote whether the transmitted is Request or Data or Acknowledgment from the source. On receiving a DS from the node, the MAC_Layer verify the medium condition and the transmission mode. After fragment the frame pass it to the PHY_Layer. The Switches perform as multi port Bridge. Non Blocking Switches transfer the data to their respective node it won't broadcast signal.

5 SpaceWire

5.1.1 Introduction

The SpaceWire standard addresses the handling of payload data on-board a spacecraft. It is a standard for a high-speed data link, which is intended to meet the needs of future, high-capability, remote sensing instruments and other space missions. SpaceWire provides a unified high-speed data-handling infrastructure for connecting together sensors, processing elements, mass-memory units, downlink telemetry sub-systems and EGSE equipment.

The SpaceWire standard specifies the physical interconnection media and data communication protocols to enable data to be sent reliably at high-speed (between 2 Mbps and 100 Mbps or more) from one unit to another. SpaceWire links are full-duplex, point-to-point, serial data communication links.

5.1.1.1 Sample Model

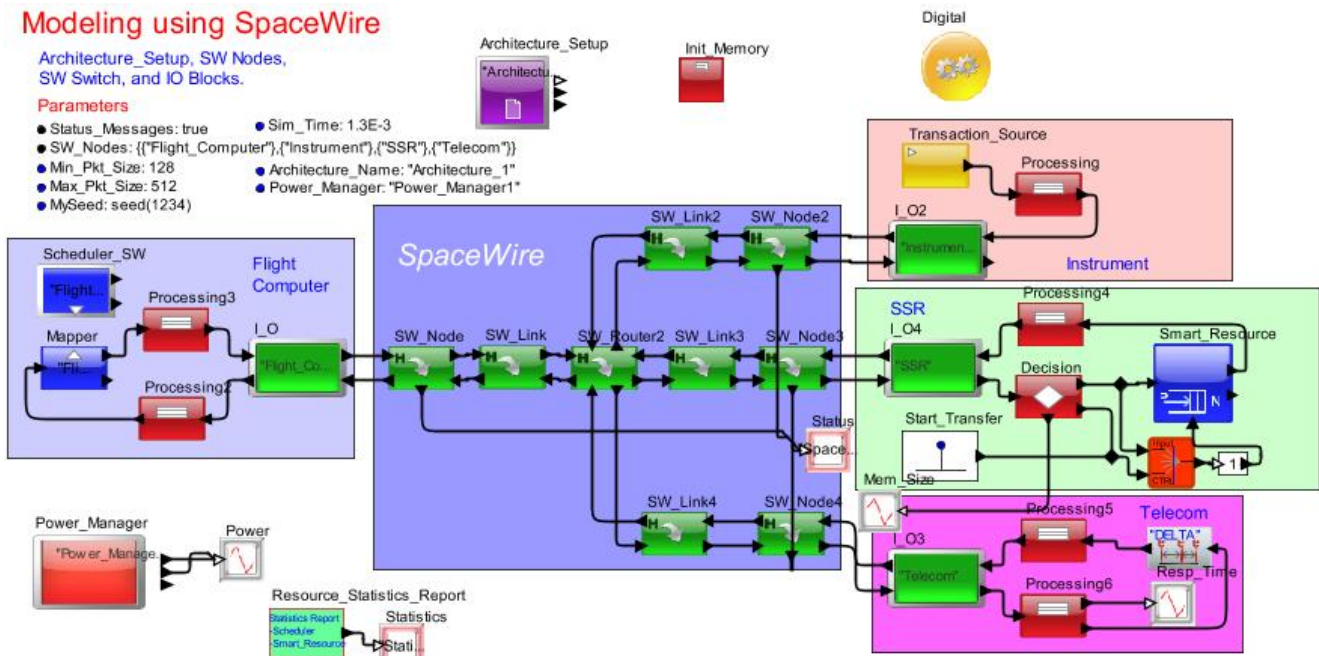


Fig.1 SpaceWire Model

5.1.2 Model Setup

The SpaceWire Blocks requires certain Model Parameters and a specific Data Structure (Processor_DS) for proper use, once the SpaceWire Node, Link and Router Block has been dragged (from Hardware_Modeling →



Advanced Buses → SpaceWire) into a model window. Each SpaceWire Node Block used in a model must designate the Architecture_Name associated with Architecture_Setup Block and Power_Manager associated with the Power_Manager Block, which also needs to be dragged into a model. To Run the Model, the Model Parameters and the transaction data structure flowing into Node, Link and Router need to be setup.

5.1.3 Model Parameter

The Parameter for a system that uses implementation of SpaceWire Bus

4. Sim_Time = 1.3E-3
5. Architecture_Name = "Architecture_1"
6. Power_Manager = "Power_Manager1"
7. Min_Pkt_Size = 1024
8. Max_Pkt_Size = 4096
9. MySeed = seed(1234)

Add the **Architecture_Setup** Block

Configure the **Power_Manager** Block as follows

for SpaceWire Node,

Architecture_Name + Node_Name (for eg, Architecture_1_SW_Node_1) and then updated the power State value individually.

for SpaceWire Link,

“Scheduler_” + Link_Name +”Forward/Reverse” (for eg, Scheduler_SW_Link_1_Forward) and then updated the power state value individually.

For SpaceWire Router,

Architecture_Name + Router_Name (for eg, Architecture_1_Router_1) and then updated the power State value individually.

Architecture_Block	Standby	Active	Wait	Idle	Cycles;
Scheduler_Flight_Computer_CPU	70.0	350.0	0.0	0.0	0 ;
Architecture_1_SW_Node_1	0.0	0.1	0.0	0.0	0 ;
Architecture_1_SW_Node_2	0.0	0.1	0.0	0.0	0 ;
Architecture_1_SW_Node_3	0.0	0.1	0.0	0.0	0 ;
Architecture_1_SW_Node_4	0.0	0.1	0.0	0.0	0 ;
Scheduler_SW_Link_1_Forward0.0	0.1	0.0	0.0	0	;
Scheduler_SW_Link_1_Reverse	0.0	0.1	0.0	0.0	0 ;
Scheduler_SW_Link_2_Forward0.0	0.1	0.0	0.0	0	;
Scheduler_SW_Link_2_Reverse	0.0	0.1	0.0	0.0	0 ;
Scheduler_SW_Link_3_Forward0.0	0.1	0.0	0.0	0	;
Scheduler_SW_Link_3_Reverse	0.0	0.1	0.0	0.0	0 ;
Scheduler_SW_Link_4_Forward0.0	0.1	0.0	0.0	0	;
Scheduler_SW_Link_4_Reverse	0.0	0.1	0.0	0.0	0 ;
Architecture_1_Router_1	0.0	0.1	0.0	0.0	0 ;

5.1.3.1 Initialize the Processing Data Structure

The standard library blocks like Traffic and transaction sequence are used to generate necessary Processor_DS. The following Processor_DS fields need to be initialized to send transaction through SpaceWire bus.

Processor_DS

A_Bytes

A_Source

A_Destination

A Processing block can be used to set the Processor_DS fields, If an I_O blocks is used, the I_O blocks fields can also be used to replace the Processor_DS field values.

5.1.4 SpaceWire Node

A SpaceWire node shall comprise one or more SpaceWire link interfaces (encoder-decoders) and an interface to the host system. A SpaceWire node shall accept a stream of packets from the host system for transmission or provide a stream of packets to the host system after reception from the SpaceWire link, or do both. In Transmitter side, Queue it up all the inputs from Traffic Source and start processing one by one using Event Mechanism. During processing the token, power is set to be active state. if there is no transaction, set the power state to standby. In Receiver side, check out all the Error occurrence such as Buffer Full Error, Bit Error and Timeout Error resend the Token by setting A_Packet = "EEP" and calculate the Statistics.

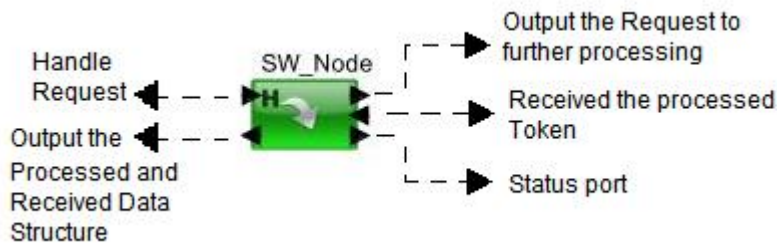


Fig. 2 SpaceWire Node Block

5.1.4.1 Parameter Setup

Parameter Name	Value (Data type)	Explanation
Node_Name	“SW_Node_1” (String)	Unique name for the Node Name
Node_Speed_Mbps	100.0 (Double)	Node Speed determines the Node Delay and Link Delay
BER_Rate	1.0E-6 (Double)	Bit Error Rate is used to verify the received token at node is within the limit, Otherwise the

		retransmission occurs
Architecture_Name	“Architecture_1” (String)	Add the Node Block to the Architecture for further power calculation
Node_Out_Buffer_Length	16 (Int)	Output buffer queue length is set to verify the Node Queue is within the specified limit, otherwise retransmission occurs.
Packet_Overhead_Bytes	16 (Int)	Extra bytes added over the data packet. used for Link_Delay calculation
Sim_Time	Sim_Time (Double)	End simulation Time in Digital Simulator
Debug	true (Boolean)	Node status are sent out (Send, Receiving, retransmission and power status)
Time_to_Init_Link	1.0E-06 / Node_Speed_Mbps	Request Token must wait till the specified time.

5.1.5 SpaceWire Link

SpaceWire nodes and SpaceWire routing switches shall be interconnected with SpaceWire links. Transfer (FULL Duplex Transfer Mode) the token based on the Link_Delay field + Link_Latency_Sec. where Link_Delay Field is calculated in the Node itself. $Link_Delay = Packet_Overhead_Bytes * ((1.0E-6 / Node_Speed_Mbps) * 8.0)$ and Link_Latency_Sec Parameter is an userdefined extra delay. Link_Length_Feet decides the Number of cycles required for processing in Scheduler.

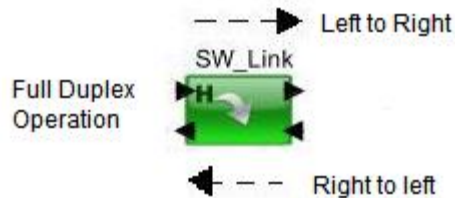


Fig. 3 SpaceWire Link Block

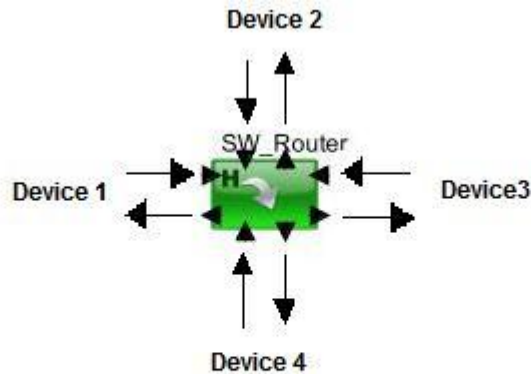
5.1.5.1 Parameter Setup

Parameter Name	Value (Data type)	Explanation
Link_Name	“SW_Link_1” (String)	Unique name for the Link Name
Link_Length_Feet	200.0 (Double)	Link_Length_Feet decides the

		Number of Clocks required for processing in Scheduler
Link_Latency_Sec	1.0E-09 (Double)	Link Latency is the extra latency added to the Link_Delay

5.1.6 SpaceWire Router

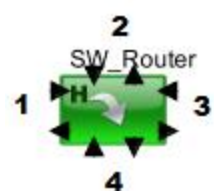
The Routing out the token to the desired destination while routing set the power state to Active. If there is no token for processing then set the power state to Standby. If a packet arriving at a routing switch has an invalid destination address then that packet shall be discarded.



Device_Connected_To_Router = {"Device_1"}, {"Device_2"}, {"Device_3"}, {"Device_4"}
(Parameter)

Fig.4 SpaceWire Router Block

5.1.6.1 Parameter Setup

Parameter Name	Value (Data type)	Explanation
Router_Name	"Router_1" (String)	Unique name for the Router Name
Router_Speed_Mbps	500.0 (Double)	Router_Speed_Mbps decides the Router Latency
Device_Connected_To_Router	{{"Node_1"}, {"Node_2"}, {"none"}, {"none"}}	Device connected to the Router is set based on the port position 

5.1.6.2 Block Diagram

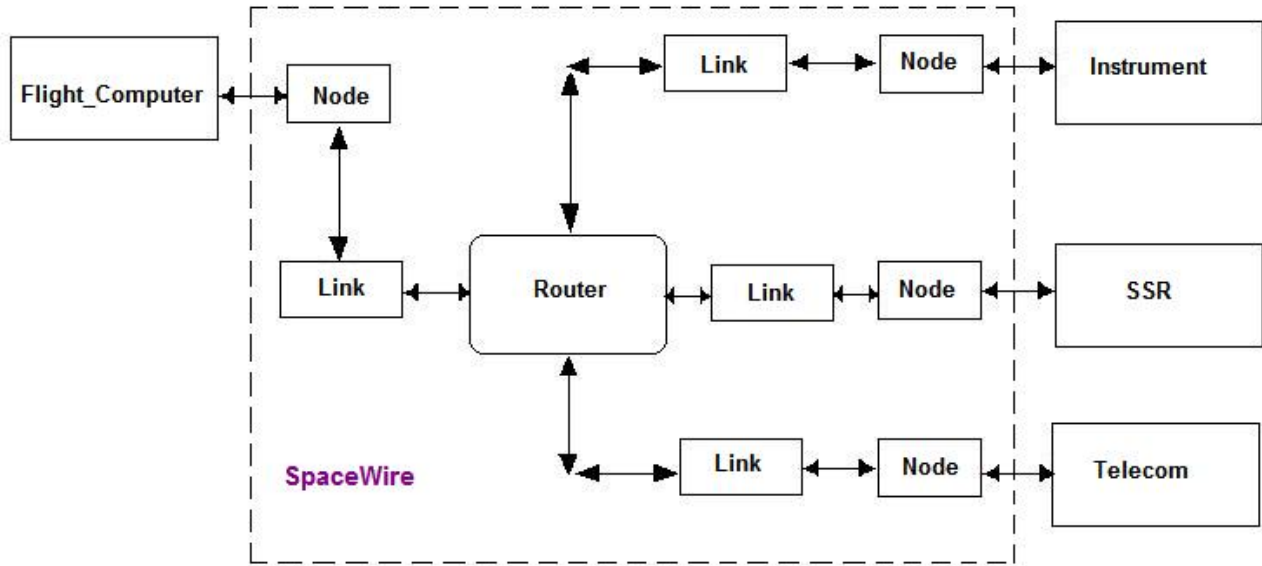


Fig.5 SpaceWire Model Block Diagram

The Data_structure (Processor_DS) generate at the Instrument source and pass over the SpaceWire Bus to reach the Flight_Computer destination. The Flight_Computer modified the destination and bytes transferred to SSR and pass over the seperate link and then the same token is routine back to the Telecom block and keeps on repeating the process till the Simulation time reaches. At the End of Simulation the power and overall Statistics are calculated.

5.1.7 SpaceWire Description

The SpaceWire standard covers the following normative protocol levels

5.1.7.1 Packet Level

It defines how data is encapsulated in packets for transfer from source to destination. The format of a packet is illustrated in Figure

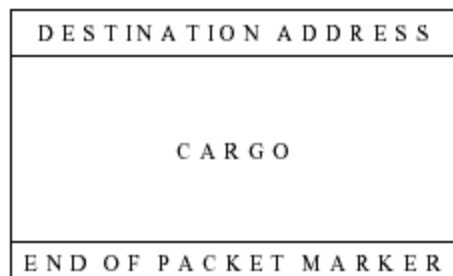


Fig.6 Packet Format

The “**Destination address**” is a list of zero or more data characters that represent the destination identity. This list of data characters represents either the identity code of the destination node or the path that the packet takes to get to the destination node.

The “**cargo**” is the data to transfer from source to destination.

The “**End of packet marker**” is used to indicate the end of a packet. Two end of packet markers are defined:

- a. EOP normal end_of_packet marker → indicates end of packet;
- b. EEP error end_of_packet marker → indicates that the packet is terminated prematurely due to a error.

5.1.7.2 Network Level

The network level defines what a SpaceWire network is, describes the components that make up, explains how packets are transferred across it, and details the manner in which it recovers from errors. A SpaceWire network is made up of a number of SpaceWire nodes interconnected by SpaceWire routing switches. SpaceWire nodes are the sources and destinations of packets and provide the interface to the application systems. SpaceWire nodes can be directly connected together using SpaceWire links or they can be interconnected via SpaceWire routing switches using SpaceWire links to make the connection between node and routing switch. A SpaceWire routing switch has several link interfaces connected together by a switch matrix, which allows any link input to pass the packets that it receives on to any link output for retransmission.

5.1.7.3 Flow Control Mechanism

Flow control is necessary to manage the movement of packets across a network from one node to another. For a node or router to receive some data there must be buffer space for that data in the receiving node or router. When the receive-buffer becomes full the receiver must stop the transmitting node from sending any more data. SpaceWire uses flow control tokens to manage the flow of data across a data link connecting one node to the next. Initially Source Node generate the Event mechanism and does not allow next transaction to flow through the network till the current transaction reaches the destination.

5.1.7.4 ERROR Scheme

The error occurrence and recovery in the SpaceWire are as follows.

a) Buffer Overflow Error

The Token in the output Queue is greater than the user-defined Parameter (Node_Out_Buffer_Length) result in initialize the retransmission by setting A_Packet field to EEP (Error End of Packet) and sent the token back to Source.

b) Bit Error Rate

The Ratio of number of bits received in error to the total number of bits sent across a link. If the Ratio exceeds the user-defined BER_Rate Parameter, initialize the retransmission by setting A_Packet field to EEP and sent the token back to Source.

c) Destination Error

A packet that arrives at a routing switch with an invalid address, i.e. an address that is not recognized by the routing switch, is discarded.

d) TimeOut Error

When an application tries to read a packet from a link interface, it can set a timeout period for reception of the packet. If the complete packet has not been received when the timeout period expires then the receiver is assumed blocked. The link interface is then disabled to cause a disconnect error and reset of the link, and then enabled again to allow the link to start and reconnect. The Error Recovery is done by retransmitting the specific token over the network.

5.1.8 Statistics

Statistics collected for the SpaceWire include

- Throughput in Mbps
- Utilization percentage
- Input Output transactions per sec (IOs_per_second)
- Input buffer occupancy in words

The sample statistics collected in the model are given below in min, mean, stdev, and max values for the SpaceWire as follows

```

BLOCK          = "SpaceWire_Model.SSR_Q",
DELTA          = 0.0,
DS_NAME        = "Queue_Common_Stats",
ID             = 2,
INDEX          = 0,
Number_Entered = 26,
Number_Exited  = 26,
Number_Rejected = 0,
Occupancy_Max  = 1.0,
Occupancy_Mean = 1.0,
Occupancy_Min  = 1.0,

```

Occupancy_StDev = 0.0,
 Queue_Number = 1,
 TIME = 0.0013,
 Total_Delay_Max = 6.915000000000244E-6,
 Total_Delay_Mean = 3.58000000000031E-7,
 Total_Delay_Min = 0.0,
 Total_Delay_StDev = 1.3896791273251169E-6,
 Utilization_Mean = 0.0

BLOCK = "SpaceWire_Model05.Flight_Computer_CPU",
 DELTA = 0.0,
 DS_NAME = "Queue_Common_Stats",
 ID = 2,
 INDEX = 0,
 Number_Entered = 26,
 Number_Exited = 26,
 Number_Rejected = 0,
 Occupancy_Max = 1.0,
 Occupancy_Mean = 0.07142857142857142,
 Occupancy_Min = 0.0,
 Occupancy_StDev = 0.2575393768188564,
 Queue_Number = 1,
 TIME = 0.0013,
 Total_Delay_Max = 7.98399999999943E-6,
 Total_Delay_Mean = 4.985384615384614E-6,
 Total_Delay_Min = 2.14400000000039E-6,
 Total_Delay_StDev = 1.974573509489743E-6,
 Utilization_Mean = 9.970769230769228

6 Rapid IO

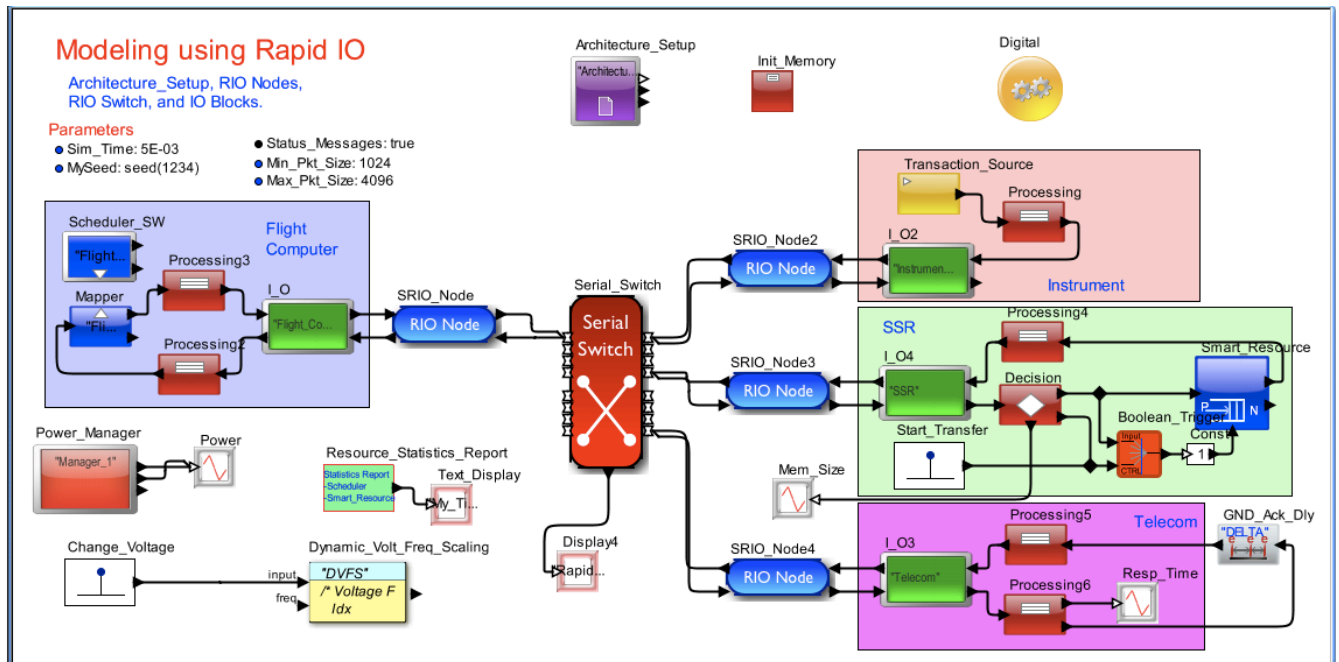
6.1 Introduction

The **Rapid IO Block Set** is a high performance RIO_Node that works in conjunction with the Serial Switch to create a Rapid IO bus that can be altered, according to specific arbitration algorithms, or other custom configurations. The RIO_Node provides the logic for master and slave node message processing, while the Serial Switch provides the channels and timing needed by the Rapid IO protocol.

In addition, one can monitor the Rapid IO operation with a port at the bottom of the Serial_Switch block, shown in the diagram as connected to a “Display” block to see if “Doorbell”, “HW_Ack”, “Retry”, “NRead”, “NWrite”, etc. messages are being sent in the expected order. One can also turn off the monitoring, once the messages have been validated using a parameter of the RIO_Node block called “Enable_Status_Messages” to false.

The RIO_Node takes transactions, or transaction fragments, at a master RIO_Node and sends messages to the designated slave Rapid_IO_Node, based on an internal routing table. This routing table is assembled in the Serial Switch and utilizes fields in the Processor_DS called “A_Source” and “A_Destination” to route through the Rapid IO bus. The Serial Switch has a parameter for the “Bus_Width” that will determine the latency through the switch based on the field “A_Bytes_Sent” in the Processor_DS.

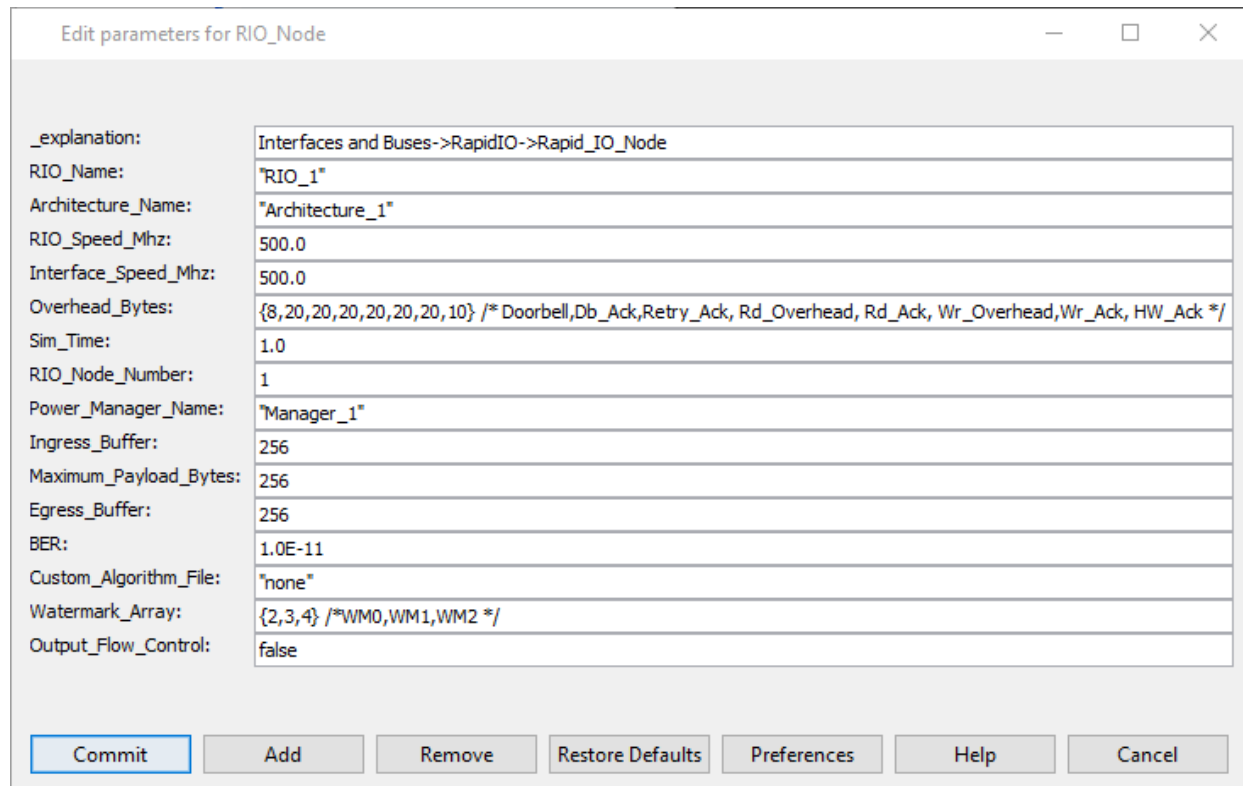
6.2 Rapid IO Blockset



Setup

The RIO_Node Block requires certain Model Parameters and a specific Data Structure (Processor_DS) for proper use, once the RIO_Node Block has been dragged into a model window. Each RIO_Node Block used in a model must designate the Block_Name and Architecture_Name associated with the Architecture_Setup Block, which also needs to be dragged into a model, if one does not exist.

6.3 RIO_Node Block Configuration Parameters



_explanation:	Interfaces and Buses->RapidIO->Rapid_IO_Node
RIO_Name:	"RIO_1"
Architecture_Name:	"Architecture_1"
RIO_Speed_Mhz:	500.0
Interface_Speed_Mhz:	500.0
Overhead_Bytes:	{8,20,20,20,20,20,10} /* Doorbell,Db_Ack,Retry_Ack, Rd_Overhead, Rd_Ack, Wr_Overhead,Wr_Ack, HW_Ack */
Sim_Time:	1.0
RIO_Node_Number:	1
Power_Manager_Name:	"Manager_1"
Ingress_Buffer:	256
Maximum_Payload_Bytes:	256
Egress_Buffer:	256
BER:	1.0E-11
Custom_Algorithm_File:	"none"
Watermark_Array:	{2,3,4} /*WM0,WM1,WM2 */
Output_Flow_Control:	false

Buttons: Commit, Add, Remove, Restore Defaults, Preferences, Help, Cancel

The key RIO_Node parameter settings:

- RIO_Name: "RIO"
- Architecture_Name: "Architecture_1"
- RIO_Speed_Mhz: 1000.0
- Interface_Speed_Mhz: 1000.0
- Overhead_Bytes: {8,20,20,20,20,20,20}
- Maximum_Payload_Bytes: 256
- Egress_Buffer: 256
- Sim_Time: 1.0
- RIO_Node_Number: 1
- Power_Manager_Name: "Manager_1"

The RIO_Name must be unique within an RIO cluster, i.e., each RIO_Node has a unique RIO_Name + "_" + RIO_Node_Number. The RIO_Speed_Mhz, Interface_Speed_Mhz, Device_Width_Bytes are for internal node timing. The RIO_Speed_Mhz is used to calculate cycle time if a slave mailbox is full, meaning the current fragment will exceed the slave mailbox size in bytes; and must wait. Interface_Speed_Mhz and Device_Width_Bytes determine the rate at which incoming data transfers fragment incoming data structures. If



the Device_Width_Bytes is set to zero, then the input fragmenter does not delay fragments generated, useful for PHY layer only transfers. Send_Status_Messages, default of true, will send status of messages processed by each RIO_Node to the Serial_Switch. User can set this flag to false, once model is running as expected.

One can set the mailbox lengths for sending and receiving by setting the Mailbox_Length_Bytes parameter. One can also set the Overhead_Bytes for the following conditions: Doorbell, Done, Retry, N_Rd_Overhead, Rd_Done, N_Wr_Overhead, or Wr_Done.

The RIO_Node Block requires certain Model Parameters and a specific Data Structure (Processor_DS) for proper use, once the RIO_Node Block has been dragged into a model window. Each RIO_Node Block used in a model must designate the Architecture_Name associated with the Architecture_Setup Block, which also needs to be dragged into a model, if one does not exist. One can also monitor power by dragging a Power_Manager block into the model, and in each RIO node duplicate the power manager name in the Power_Manager_Name parameter. This is also true for the Serial_Switch.

6.4 Connecting the RIO_Node Block in a model

The RIO_Node Block has these ports:

<i>input</i>	– Left-Hand_Side Input Port
<i>input2</i>	– Right-Hand_Side Input Port
<i>output</i>	– Left-Hand_Side Output Port
<i>output2</i>	– Right-Hand_Side Output Port

6.4.1 RIO_Node Block added to a Model

The main consideration when adding a RIO_Node block to a model is the mailbox designation and if any transaction fragmenting is necessary, as the RIO_Node block does not duplicate the functionality, of existing fragmenters in the Architectural Library. Mailbox designation is set by adding fields to the Processor_DS: RIO_Sender_Mailbox, or RIO_Receiver_Mailbox. If the user wanted to assign multiple channels, then as each transaction arrives, one could increment the mailbox designator to a new mailbox, to allocate the bandwidth desired in the protocol. Certain transactions could always have access to certain mailboxes, such as Instruction and Data cache, while other mailboxes could be allocated on a round robin basis.

There is also the notion of a transaction entering a “master” node and exiting the RIO bus topology from a “slave” node. The master node ports are the “input” and “output”. The switch side ports are “input2” and “output2”.



One of the key advantages of the RIO_Node block is a User can modify the Message Processing, as required, since it is written in the VisualSim RegEx language with If-Else conditional statements:

```

if (port_token.containsRecordTokenLabel("A_Message")) { /* Message Processing */
  SWITCH (port_token.RIO_Message)
  {
    CASE: Doorbell /* Slave */
      SEND (Send_Through, port_token)
      Source = port_token.A_Source
      port_token.A_Source = port_token.A_Destination
      port_token.A_Destination = Source
      port_token.RIO_Message = "HW_Ack" /* Slave send Ack */
      port_token.RIO_Bytes = Done_Overhead_Bytes
      GTO (Send_Return)

    CASE: HW_Ack /* Ack from Slave */
      Sender_Mailbox_ID = port_token.RIO_Sender_Mailbox
      (Sender_Length_Arr(Sender_Mailbox_ID)).decr()
      QUEUE (Sender_Mailbox_ID, length)
      if (length > 0) { /* Start new Xfer? */
        QUEUE (Sender_Mailbox_ID, pop) /* Still Four Threads */
        GTO (Send_Through) /* Master send Next */
      }
      if (Power_Manager_Flag && RIO_Power_Active) {
        if (Receiver_Length_Arr.sum() == 0) {
          if (Sender_Length_Arr.sum() == 0) {
            powerUpdate(Power_Manager_Name_, RIO_Power_Name, "Standby")
            PM_DS = powerManager(Power_Manager_Name_)
            Cycles = (PM_DS.get(RIO_Power_Name)).get("Cycles")
            if (Cycles != 0) {
              WAIT (Cycle_Time * Cycles)
            }
            RIO_Power_Active = false
          }
        }
      }
    }
  }
  GTO (END)

  CASE: Retry /* Master retry */
    Number_of_Retries = Number_of_Retries + 1
    Source = port_token.A_Source
    port_token.A_Source = port_token.A_Destination
    port_token.A_Destination = Source
    port_token.A_Priority = port_token.A_Priority + 1 /* Increase priority */
    GTO (New_Transaction)

  CASE: NRead
    Write_Flag = false
    GTO (NWrite_Continue)
  CASE: NWrite /* Slave accept data */
    Write_Flag = true
  
```

```

LABEL: NWrite_Continue
Mailbox_ID          = port_token.RIO_Receiver_Mailbox
Mbox_Length_Bytes  = Receiver_Length_Arr(Mailbox_ID)
Mailbox_Max_Bytes  = Mailbox_Length_Bytes_(Mailbox_ID)
if (!Write_Flag)   { /* Read Command */
    Packet_Bytes    = Mbox_Length_Bytes + 4
}
else {
    Packet_Bytes    = Mbox_Length_Bytes + port_token.A_Bytes_Sent
}
if (Packet_Bytes   <= Mailbox_Max_Bytes) {
    if (!Write_Flag) { /* Read Command */
        Bytes_Out_Sum = Bytes_Out_Sum + 4
        Receiver_Length_Arr(Mailbox_ID) = Packet_Bytes
        if (port_token.A_Destination == My_Master) {
            port_token.RIO_Message = "Rd_Done"
        }
    }
    else {
        Bytes_Out_Sum = Bytes_Out_Sum + port_token.A_Bytes_Sent
        if (port_token.A_Task_Flag) {
            Receiver_Length_Arr(Mailbox_ID) = Packet_Bytes
        }
        if (port_token.A_Destination == My_Master) {
            port_token.RIO_Message = "Wr_Done"
        }
    }
}
SEND (Send_Through, port_token)
port_token.RIO_Message = "HW_Ack" /* Slave send Ack */
port_token.RIO_Bytes = 0
}
else if (port_token.A_Bytes_Sent > Mailbox_Max_Bytes && Write_Flag) {
    throwMyException (Block_Name + " RIO_Message (" + port_token.RIO_Message + ") from Source (" +
port_token.A_Source + ") with Size (" + port_token.A_Bytes_Sent + ") is larger than the Mailbox (" +
Mailbox_Max_Bytes + ")")
}
else {
    port_token.RIO_Message = "Retry" /* Slave send Retry */
    port_token.RIO_Bytes = Retry_Overhead_Bytes
}
Source = port_token.A_Source
port_token.A_Source = port_token.A_Destination
port_token.A_Destination = Source
GTO (Send_Return)

CASE: Rd_Done // Write return from Read, assumes one RIO in model
Mailbox_ID = port_token.RIO_Receiver_Mailbox
Mbox_Length_Bytes = Receiver_Length_Arr(Mailbox_ID)
Packet_Bytes = Mbox_Length_Bytes - 4
Receiver_Length_Arr(Mailbox_ID) = Packet_Bytes
GTO (New_Transaction)

```

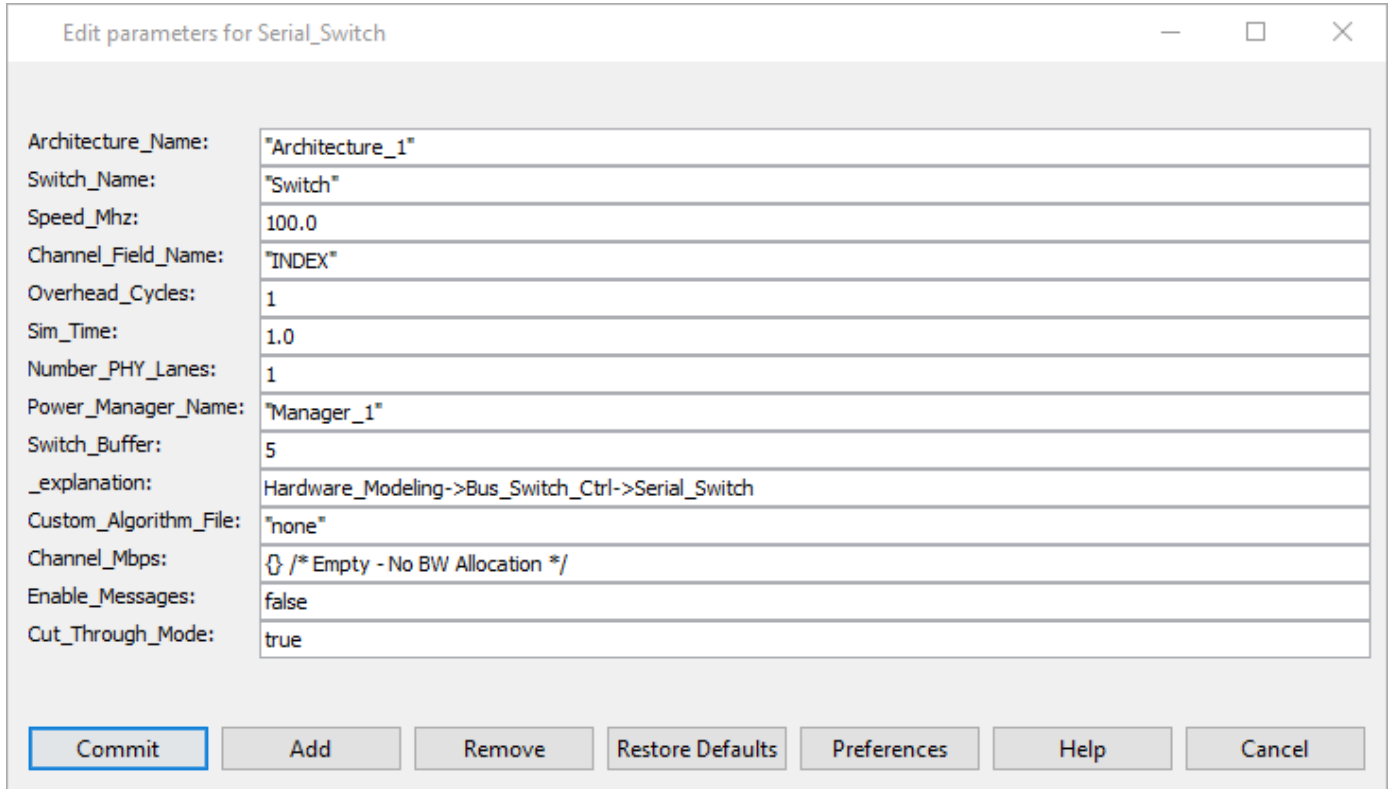
```
CASE: Wr_Done           // Read return from DMA Write, assumes one RIO in model
  Mailbox_ID           = port_token.RIO_Receiver_Mailbox
  Mbox_Length_Bytes    = Receiver_Length_Arr(Mailbox_ID)
  Packet_Bytes         = Mbox_Length_Bytes - port_token.A_Bytes_Sent
  Receiver_Length_Arr(Mailbox_ID) = Packet_Bytes
GTO (New_Transaction)

CASE: New               // Start a New Transaction
GTO (New_Transaction)

CASE: DEFAULT
  throwMyException (Block_Name + " has an unknown RIO_Message (" + port_token.RIO_Message + ")")
}
}
```

In the above, blue script designates master execution, red script designates slave execution, and black script designates master or slave execution.

6.5 Serial Switch Block Configuration Parameters



Parameter	Value
Architecture_Name:	"Architecture_1"
Switch_Name:	"Switch"
Speed_Mhz:	100.0
Channel_Field_Name:	"INDEX"
Overhead_Cycles:	1
Sim_Time:	1.0
Number_PHY_Lanes:	1
Power_Manager_Name:	"Manager_1"
Switch_Buffer:	5
_explanation:	Hardware_Modeling->Bus_Switch_Ctrl->Serial_Switch
Custom_Algorithm_File:	"none"
Channel_Mbps:	{ } /* Empty - No BW Allocation */
Enable_Messages:	false
Cut_Through_Mode:	true

Buttons: Commit, Add, Remove, Restore Defaults, Preferences, Help, Cancel

The key Serial Switch parameter settings:

- Architecture_Name: "Architecture_1"
- Switch_Name: "Switch"
- Speed_Mhz: 1000.0
- Channel_Field_Name: "RIO_Lane_ID"
- Overhead_Cycles: 1
- Sim_Time: Sim_Time
- Number_PHY_Name: 1
- Power_Manager_Name: "Manager_1"
- Custom_Algorithm_File: "None"

The Architecture_Name and Switch_Switch parameters need to be unique and match the Architecture_Setup block. The Speed_Mhz determines the internal bit cycle time. The Channel_Field_Name is the field for the channel designation in the Serial_Switch, "RIO_Lane_ID" is a field created by the RIO node block. The user does not need to add this field, as it will be added by a master RIO node and removed by a slave RIO node. Overhead_Cycles are the overhead cycles need to switch, depending on the internal switch design. If a Banyan style switch is used, then for 16 ports 4 cycles would be needed, for example. One can also monitor power by dragging a Power_Manager block into the model, and in each RIO node duplicate the power manager name in the Power_Manager_Name parameter.

The parameter Number_PHY_Name can be used to represent multiple twisted pair configurations to the Serial_Switch. The default is 1, meaning a single PHY connection to the Serial_Switch.

6.6 Connecting the Serial_Switch Block in a model

The Serial_Switch Block has 16 bi-directional ports, user needs to connect input port first, then output port:

<i>input</i>	– Left-Hand Side Input Port
<i>input2</i>	– Left-Hand Side Input Port
<i>input3</i>	– Left-Hand Side Input Port
<i>input4</i>	– Left-Hand Side Input Port
<i>input5</i>	– Left-Hand Side Input Port
<i>input6</i>	– Left-Hand Side Input Port
<i>input7</i>	– Left-Hand Side Input Port
<i>input8</i>	– Left-Hand Side Input Port
<i>input9</i>	– Right-Hand Side Input Port
<i>input10</i>	– Right-Hand Side Input Port
<i>input11</i>	– Right-Hand Side Input Port
<i>input12</i>	– Right-Hand Side Input Port
<i>input13</i>	– Right-Hand Side Input Port
<i>input14</i>	– Right-Hand Side Input Port
<i>input15</i>	– Right-Hand Side Input Port
<i>input16</i>	– Right-Hand Side Input Port
<i>output</i>	– Left-Hand Side Output Port
<i>output2</i>	– Left-Hand Side Output Port
<i>output3</i>	– Left-Hand Side Output Port
<i>output4</i>	– Left-Hand Side Output Port
<i>output5</i>	– Left-Hand Side Output Port
<i>output6</i>	– Left-Hand Side Output Port
<i>output7</i>	– Left-Hand Side Output Port
<i>output8</i>	– Left-Hand Side Output Port
<i>output9</i>	– Right-Hand Side Output Port
<i>output10</i>	– Right-Hand Side Output Port
<i>output11</i>	– Right-Hand Side Output Port
<i>output12</i>	– Right-Hand Side Output Port
<i>output13</i>	– Right-Hand Side Output Port
<i>output14</i>	– Right-Hand Side Output Port
<i>output15</i>	– Right-Hand Side Output Port
<i>output16</i>	– Right-Hand Side Output Port

6.6.1 Serial_Switch Block added to a Model

A key consideration when adding a Serial_Switch block to a model is that routing has been considered. This means that hello messages are passed from devices to each RIO Node. Typically, there is an I_O port block on each edge node of the RIO bus topology. An I_O block generates hello messages on the bus side, which are in turn passed from each RIO_Node to the Serial_Switch block. If one wants to add redundant paths, then additional Serial_Switches to the destination will have a lower priority in the routing table than fewer Serial_Switches.

6.7 Data Structure: Processor_DS

The Processor_DS was created at the used in the Architectural Library processors, busses and memories. The following are specific data structure fields to be set prior to entering the RIO_Node Block, red indicates fields that should be set.

```
{A_Address           = 100,
A_Branch             = false,
A_Bytes              = 8,
A_Bytes_Remaining   = 4,
A_Bytes_Sent         = 4,
A_Command            = "Read",
A_Data               = "MyData",
A_Destination        = "RIO_1",
A_First_Word         = true,
A_Hop                = "RIO_1",
A_IDX                = 0,
A_Instruction        = {"ADD", "ADD", "ADD", "ADD", "ADD", "ADD"},
A_Interrupt          = false,
A_Pipeline           = {0, 0, 0, 0, 0, 0},
A_Prefetch           = false,
A_Priority           = 0,
A_Proc_Return        = -1,
A_Return             = -1,
A_Source             = "MyNode",
A_Status             = "Status",
A_Task_Flag          = false,
A_Task_ID            = 1L,
A_Task_Name          = "Name",
A_Time               = 0.0,
A_Variables          = 16,
RIO_Receiver_Mailbox = 1,
RIO_Sender_Mailbox   = 1,
BLOCK                = "Traffic",
DELTA                = 0.0,
DS_NAME              = "DS_Traffic",
ID                   = 1,
INDEX                = 0,
TIME                 = 2.0E-9}
```

A_Source may represent an I_O port on the bus, or an RTOS name where the task or thread originated. It is important that A_Destination match the destination node. Internally, the Processor, Cache, DRAM, DMA



blocks will set these fields correctly. A_Priority (higher value is higher priority) can be used in conjunction with Interface_Speed_Mhz and

7 Ethernet Audio Video Bridging

Mirabilis Design provides an Audio-Video Bridging (AVB Library) as a standard add-on to the Ethernet and Networking application library in VisualSim Architect modeling environment. Using this environment, systems engineers and product architects can evaluate the impact of design decisions on the performance of their system. The AVB library has been built to the specification of the IEEE 802.1BA for time-synchronized low latency streaming services. The library supports the following standards:

- IEEE 802.1AS: Timing and Synchronization for Time-Sensitive Applications (gPTP),
- IEEE 802.1Qat: Stream Reservation Protocol (SRP),
- IEEE 802.1Qav: Forwarding and Queuing for Time-Sensitive Streams (FQTSS), and
- IEEE 802.1BA: Audio Video Bridging Systems

VisualSim AVB modeling environment supports the requirement of the automotive, consumer, and professional audio and video markets. Combined with the industry hardware, software and system library provided in VisualSim Architect, designers assemble an end-to-end system and evaluate the throughput, latency and other performance attributes. The proposed system can be tested for various fault and error conditions to evaluate the susceptibility of the system for real-world conditions.

VisualSim AVB can be used to design a completely new AVB-based network to integrate all the equipment, upgrade existing networks, and to design the electronics that are used in such networks. The AVB system can include the talkers and listeners such as video cameras, radars, broadcast systems, displays, and Electronic Control Units. The network can include AVB interfaces, bridges, switches and gateways.

7.1 Library

The AVB library is available as a feature in VisualSim. The AVB library is located in Interfaces and Buses → Audio_Video_Bridging.

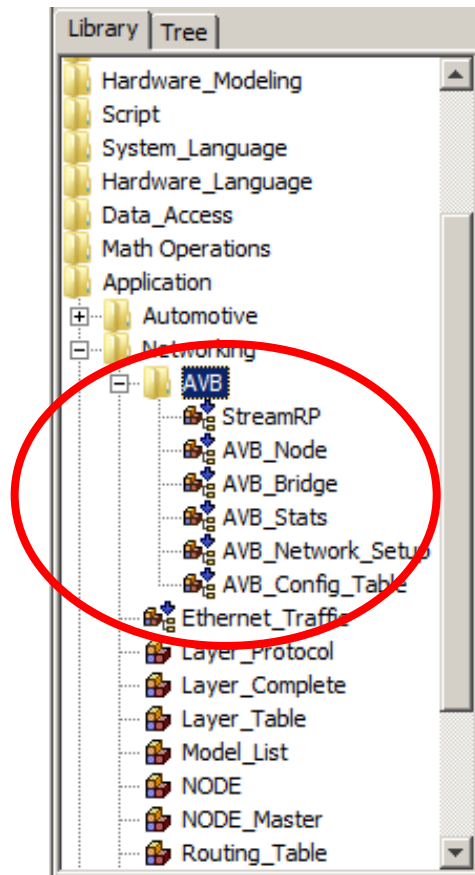


Figure 4 AVB Modeling Library

The library contains the six modeling blocks and the Ethernet Traffic block. The following blocks are located in the Application **Interfaces and Buses** → **Audio_Video_Bridging** directory.

1. StreamRP- This block manages the Talker_Advertise functionality at the Source Nodes. This is only required at the Source Nodes.
2. AVB_Node- This block connects the AVB Node to the Network. It performs the AVB Reservation protocol, traffic shaping, and clock synchronization.
3. AVB_Bridge- This block connects to both Nodes and other Bridges. It handles the Routing, traffic shaping, multicast, AVB stream reservation, and broadcast of clock synchronization messages to all the links. Note: The current implementation of the AVB Bridge can handle a maximum of eight (8) links only.
4. AVB_Stats- This is a convenient block that displays the latency plot for all streams from all the Nodes, writes the information and warning messages to a file in the model directory, and computes the latency jitters and writes it to a separate file in the Model Directory.
5. AVB_Network_Setup- One instance of this block is required in all AVB Models. This processes all the Tables in the model.
6. AVB_Config_Tables- This is a block that contains the sample of all the configuration tables required for AVB.

- Ethernet_Traffic- This is a traffic generator that can handle both Ethernet traffic and AVB talker request message.

7.2 Tutorial System

Multiple demonstration systems are provided with documentation to guide a user to learn the AVB modeling environment and create executable models.

A demonstration system is provided with explanation on the use of this library. The block diagram is provided below:

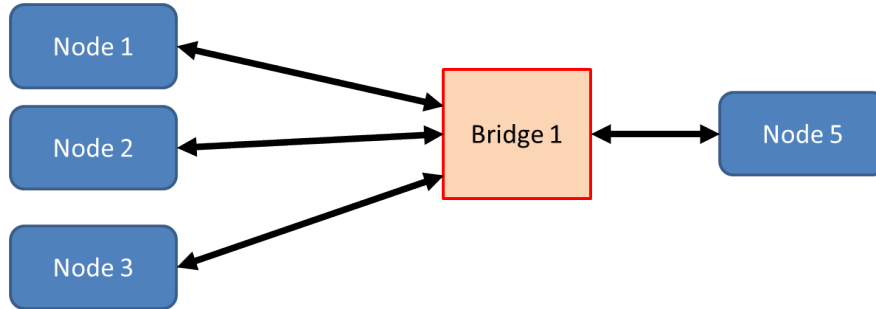
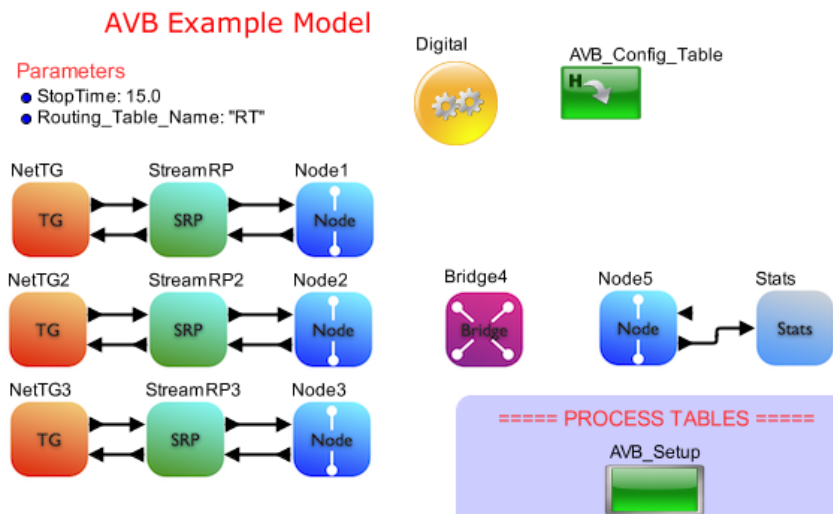


Figure 5 Block Diagram of a AVB Network

Nodes 1, 2 and 3 are Talkers. Each Node has one Ethernet, three Class A streams and one Class B stream. All the Talkers transmit to a single Listener- Node 5. All the 4 Nodes are connected to a single Bridge 1. The VisualSim model of this design is at- VS_AR/demo/networking/AVB/AVB_Example_System.xml.

Each node consists of one Ethernet_Traffic, StreamRP and Node block. The Bridge uses the AVB_Bridge block. As Node 5 is the Listener, the output of this block is connected to the AVB_Stats. The AVB_Network_Setup and the AVB_Config_Table are added to this block diagram to define the model attributes. The single parameter- Routing_Table_Name is to ensure that the Nodes, Bridges, AVB_Network_setup and the AVBP_Config_Table refer to the same Routing Table block. The Digital simulation is added with a stop time linked to both the Simulator and the AVB_Stats block.

When you run the simulation, there are four outputs. There are two plots- one displaying the latency for all the streams and the other is Histogram of the latency. There are two text files that are written at the end of the simulation- one contains information and warning messages, and the other contains the latency statistics for each stream.



8 Fibre Channel

8.1 Introduction to Fibre Channel

Fibre Channel is a technology for transmitting data between computer devices at data rates of up to 4 Gbps (and 10 Gbps in the near future). Fibre Channel is especially suited for connecting computer servers to shared storage devices and for interconnecting storage controllers and drives. Since Fibre Channel is three times as fast, it has begun to replace the Small Computer System Interface (SCSI) as the transmission interface between servers and clustered storage devices. Fibre channel is more flexible; devices can be as far as ten kilometers (about six miles) apart if optical fiber is used as the physical medium. Optical fiber is not required for shorter distances, however, because Fibre Channel also works using coaxial cable and ordinary telephone twisted pair.

There are three major fibre channel topologies, describing how a number of ports are connected together. A *port* in fibre channel terminology is any entity that actively communicates over the network, not necessarily a hardware port. This port is usually implemented in a device such as disk storage, an HBA on a server or a fibre channel switch.

- **Point-to-point (FC-P2P).** Two devices are connected directly to each other. This is the simplest topology, with limited connectivity.
- **Arbitrated loop (FC-AL).** In this design, all devices are in a loop or ring, similar to token ring networking. Adding or removing a device from the loop causes all activity on the loop to be interrupted. The failure of one device causes a break in the ring. Fibre Channel hubs exist to connect multiple devices together and may bypass failed ports. A loop may also be made by cabling each port to the next in a ring.
 - A minimal loop containing only two ports, while appearing to be similar to FC-P2P, differs considerably in terms of the protocol.
 - Only one pair of ports can communicate concurrently on a loop.
 - Maximum speed of 8GFC.
- **Switched fabric (FC-SW).** All devices or loops of devices are connected to fibre channel switches, similar conceptually to modern Ethernet implementations. Advantages of this topology over FC-P2P or FC-AL include:
 - The switches manage the state of the fabric, providing optimized interconnections.
 - The traffic between two ports flows through the switches only; it is not transmitted to any other port.
 - Failure of a port is isolated and should not affect operation of other ports.
 - Multiple pairs of ports may communicate simultaneously in a fabric.

8.2 About VisualSim Fibre Channel Library Package

Fibre Channel library is provided as a standard library in VisualSim Architect modeling environment. Using this configurable library, designers can architect next generation networking system for high performance and mission assurance systems by conducting early design space exploration, power and performance analysis. Fibre Channel library is built to the specifications and supports Fibre Channel Arbitrated Loop and Fibre Channel Switch Fabric topologies. Fibre Channel library also extends its functionality as Fibre Channel Framing and Signaling (FC-FS) and Fibre Channel - Avionics Environment – Anonymous Subscriber Message standards.

8.3 Library Blocks

Fibre Channel Library package composed of mainly 4 library components, namely

2. FC_N_Port
3. FC_Switch
4. FC_Link
5. FC_Config

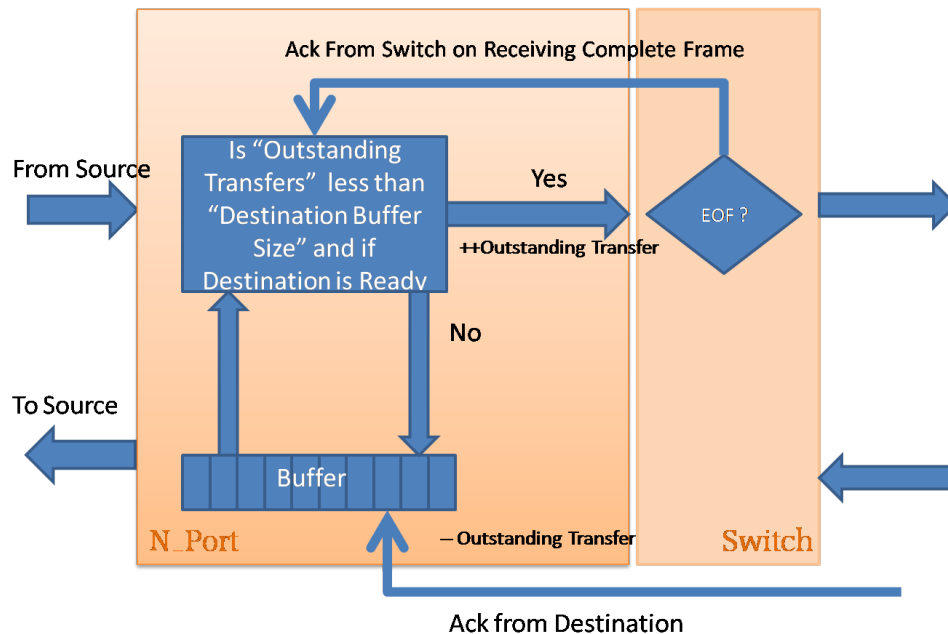
Each library components are built to Fibre Channel Specification and meets performance requirements.

8.4 FC_N_Port

FC_N_Port interfaces to the device on one side and the Fibre Channel switch on the other side. This block handles end-to-end acknowledgement for Class 1 and 2. In addition, it handles the ack between the Ingress Port of the switch and this N_Port. Make sure to connect an I_O block on the device side to handle all the routing requirements.

Current version of Fibre Channel Library requires each N_Port ID should be given as 1,2,3....16. N_Port with ID 1 should be connected to Master Device/Slave Device with name "Device_1", N_Port with ID 2 should be connected to Master Device with name "Device_2" and so on.

8.4.1 Flow Diagram



8.4.2 Data Structure Fields

N_Port requires few mandatory fields included in the A_Source, A_Destination, A_Command, A_Message (initialized with Class1, Class2 or Class3), A_Response are the required fields. These fields must be added to the incoming data structure, typically using a Processing or Decision block

Example:

input.A_Source = "Device_1"



```

input.A_Destination = "Device_9"
input.A_Command    = "Read" /*"Write"*/
input.A_Message    = "Class1" /*"Class2" or "Class3" */
input.A_Response   = "false"
input.A_Bytes      = 15000 /* Bytes */

```

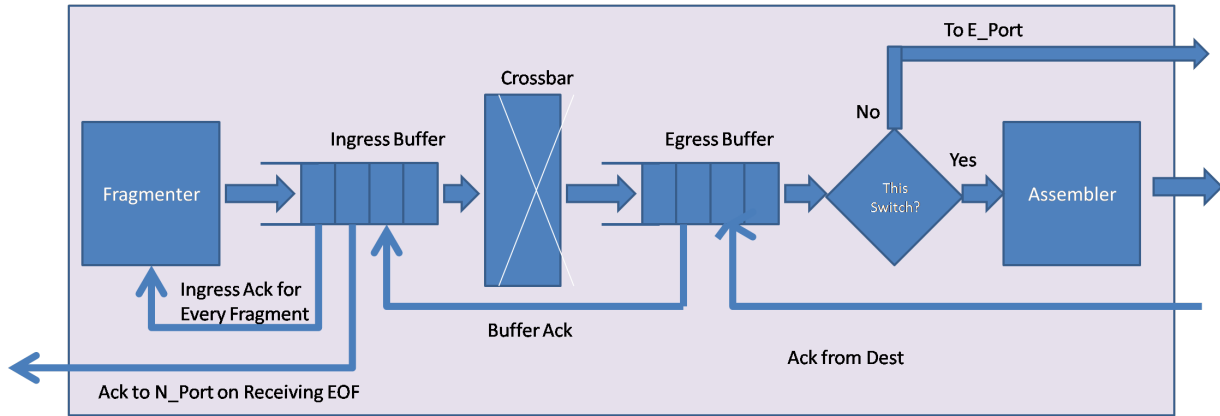
8.4.3 Parameters

Name	Type	Description
ID	Integer	Unique Identifier for the Node Port
Enable_Debug	Boolean	Enable or Disable Debug Messages
Architecture_Name	String Ex: "Architecture_1"	This is the name of the Architecture_Setup block that this Block is associated. The Architecture_Setup block maintains the routing table and statistics collection

8.5 Fibre Channel Switch

The purpose of switch block is to support fragmentation, assembly and quick data movement. Fibre Channel Switch will Fragment the incoming frame to the fragment size and forward it to ingress buffer. Each fragment is acknowledged by the Ingress Buffer before the next fragment is sent out. The Fibre Channel Switch can support up to 16 or 96 F Ports to connect N_Port and 4 E Ports to connect to Swiches . There is a unique fragmenter block per port. Ingress buffer has a dedicated queue for each connections and forwards transactions from respective queue to Egress buffer based on the flow control. Class1 messages will be sent out as soon as the egress buffer is available, however class2 and class3 implements buffer-to-buffer flow control. Each port will have a dedicated counter that monitors the number of transactions sent. This counter is used to test for available buffer space in the Egress Buffer. For every transaction sent, counter will be incremented by one. For class 2 and Class 3, ingress buffer will wait for an acknowledgement from the egress buffer. On receiving the Acknowledgement counter will be decremented by one for the corresponding port. Crossbar switch has 16*16 or 96*96 dedicated channels to make sure that there are no collisions. The Crossbar has a delay associated with it. The delay is based on the fragment size and Switch_Speed_MHz. Egress Buffer acknowledges every fragment sent from ingress buffer. If the transaction arrived is to a device connected to the current fabric block, then the transactions will be assembled else transactions will be sent to corresponding switch via E_Ports. Once all fragments are received at the Egress port, the fragments will be assembled into the original transaction. Egress receives the Ack from the Ingress of the next switch or the Destination N_Port. Once it receives Ack from next switch or destination N_Port, it transmits the next transaction

8.5.1 Flow Diagram



8.5.2 Flow Control

Flow control depends on class of service. Flow control is unique for each Class. Class 1 and Class 2 keep track of the number of buffer locations available at the destination before transmitting the next transaction. Class 3 does not require any end-to-end Ack. Class 1 will transfer from Ingress to Egress within a Switch. Class 2 and 3 requires buffer-to-buffer flow control.

8.5.3 Data Structure Fields

FC_Switch requires few mandatory fields included in the A_Source, A_Destination, A_Command ("Read"/"Write"), A_Message (initialized with Class1, Class2 or Class3), A_Bytes, A_Bytes_Remaining, A_Bytes_Sent are the required fields.

Example:

```
input.A_Source      = "Device_1"
input.A_Destination = "Device_9"
input.A_Command    = "Write" /*"Write"*/
input.A_Message    = "Class1" /*"Class2" or "Class3" */
input.A_Response   = "false"
input.A_Bytes      = 15000 /* Bytes */
input.A_Bytes_Sent = 2112
input.A_Bytes_Remaining = 12888
```

A_Sourcce field represents the name of Source device, A_Destination field represents Destination device name, A_Command is to identify whether the transaction is a Read/Write transfer. A_Message helps one to set type of service, please note that the Field values can be either "Class1", "Class2" or "Class3". Purpose of A_Response field is to indentify if a transaction is a response from Destination device or a transaction from Source device. A_Bytes field has total size of the packet, A_Bytes_Sent field get updated with Fragment Size plus Overhead Bytes in Fibre Channel Switch. A_Bytes_Remaining has the details about remaining fragment size waiting for transmission.

8.5.4 Parameters

Name	Type	Description
Switch_Name	String Ex: "FC_Fabric_1"	Name of Fibre Channel Switch
Ingress_Buffer_Size	Integer	Size of Ingress Buffer at each port

<i>Egress_Buffer_Size</i>	<i>Integer</i>	<i>Size of Egress Buffer at each port</i>
<i>Architecture_Name</i>	<i>String</i>	<i>Name of Arch_Setup</i>
<i>Fragment_Size</i>	<i>Integer</i>	<i>Size of Frame Fragment</i>
<i>Overhead_Bytes</i>	<i>Integer</i>	<i>Overhead Bytes</i>
<i>Enable_Debug</i>	<i>Boolean</i>	<i>Enable or Disable Debug Message</i>
<i>Switch_ID</i>	<i>Integer</i>	<i>Unique Identifier for Fabric Switch</i>

8.6 FC_Link

FC_Link block models the physical communication medium. User can define the length in meters and also delay associated with the type of communication medium.

8.6.1 Parameters

Name	Type	Description
Length_In_Mtrs	Integer Ex: 2	Length of Communication Medium in Meters
Dly_Per_Mtr	Double Ex: 5.0e-9	Fibre Channel Delay per meter
FC_Link_Dly	Double Ex: Length_In_Mtrs * Dly_Per_Mtr	Computed Fibre Channel Link Delay. Do not Change this parameter

8.7 FC_Config

FC_Config block is part of Fibre Channel library. Block handles routing information between one end system to another. By default Source and Destination Device names are defined as Device_1, Device_2...Device_16.

8.7.1 Parameters

Name	Type	Description
Device_Configs	Data Structure. Type: String	The content can be placed directly in the window, in a file (TXT or CSV) or a reference to another database block (extern). Each row defines which Destination device connected to which Fabric Switch
Switch_Configs	Data Structure. Type: String	Number of rows will be based on number of Fabric Switches used in the System. Switch_Idx defines the Switch

		<p><i>ID, the field Connected_Switches is a two dimensional array. As Switch supports upto 4 Switches connected to a Switch block, max length of the array would be 4. If there are multiple switches are connected to a single E-Port by cascading, then the connected switches are defined here. Ex is shown below</i></p> <pre> Switch_IDx Connected_Switches E_Port_Address ; 2 {{1,3}} {{4,1}} ; 1 {{2},{3}} {{1,1},{4,1}} ; 3 {{1}} {{1,1}} ; </pre>
--	--	---

8.8 Tutorial

Multiple demonstration system models are provided with documentation to guide a user to adopt VisualSim Fibre Channel Library for conducting early system exploration of Fibre Channel based system. Purpose of the following tutorial is to introduce VisualSim Fibre Channel libraries and understanding basic rules that needs to be followed during model construction.

Block diagram of a simple Fibre Channel based system with single Fibre Channel Switch and two end systems is shown below



Figure 1.0: System Block Diagram

Here we have two end systems, Node-1 and Node-2 connected over a Fibre Channel network. Node-1 acts as the source (Ex: Host) and Node-2 acts as a destination device (Ex: Storage). Requests can be either Read request or Write request, can be Class1, Class2 or Class3 type of service. Fiber can be either single-mode or multi-mode variant and based on the type of variant user can define the length and delay associated with fiber. Once the model construction is over, user can vary the type of service, packet size, Request type, Link distance, Switch Speed etc and analyze system behavior under different configurations.

VisualSim model of the proposed system model is shown in figure below

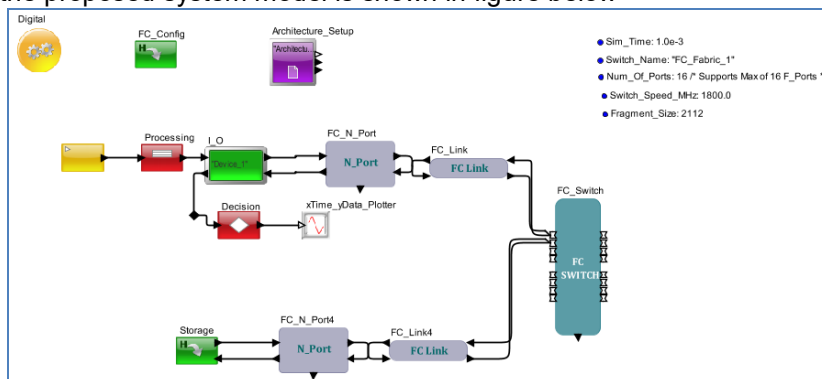


Figure 2.0: VisualSim Model

8.8.1 Basic Rules

1. All the Source and Destination FC_N_Port ID's should be in accordance to the connected port number of FC_Switch. If Device_1 is connected to FC_Switch port number 1, then FC_N_Port ID should be configured as 1.
2. Each Source Device should have following Data Structure fields. Processor_DS Data Structure is recommended
 - input.A_Source = "Device_1"
 - input.A_Destination = "Device_9"
 - input.A_Command = "Write" /*"Write"*/
 - input.A_Message = "Class1" /*"Class2" or "Class3" */
 - input.A_Response = "false"
 - input.A_Bytes = 15000 /* Bytes */
3. FC_Config block should be present in all Fibre Channel related models. Source and Destination Names should be updated in End_Point_Table if the names are others than Device_1, Device_2....Device_16.
4. Architecture_Setup block should be present

8.8.2 Construction Steps

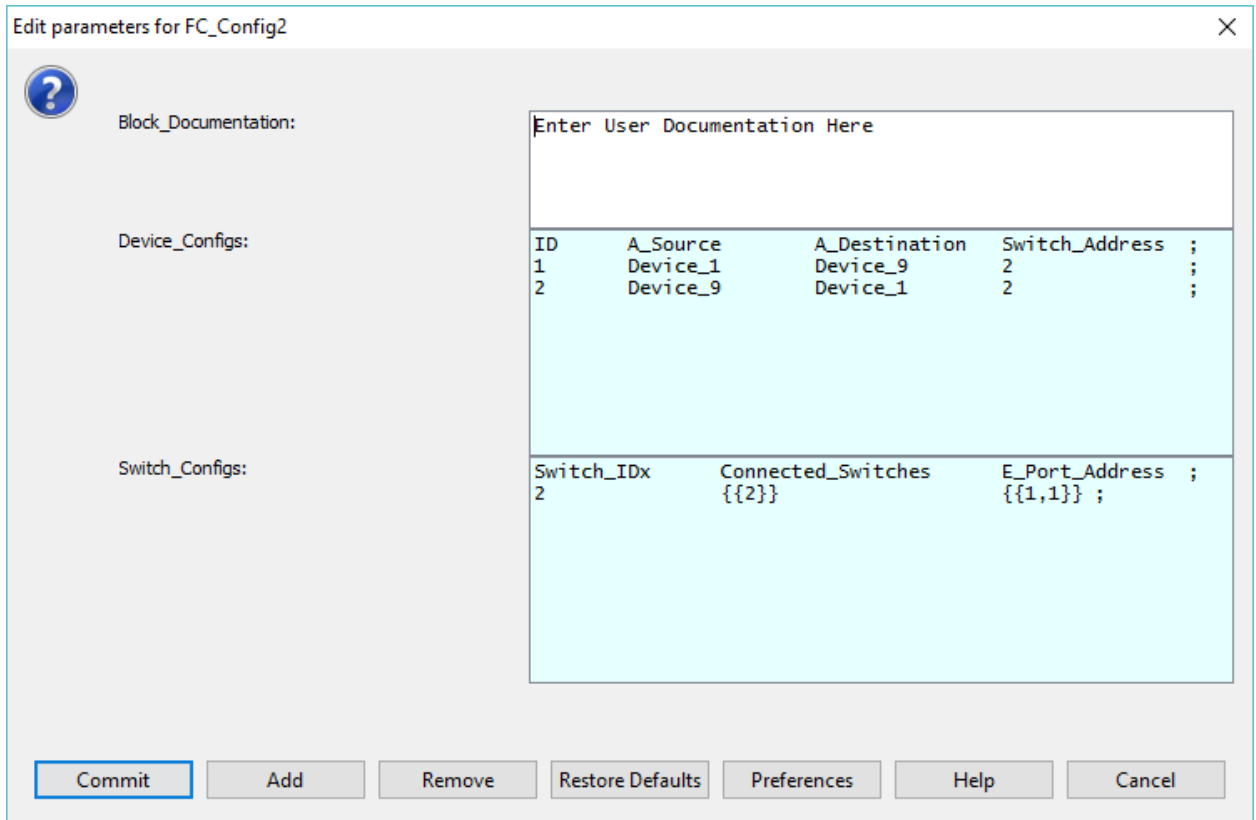
To construct a Fibre Channel Network model, there are five steps

1. Instantiate Config and Update the Device_Configs and Switch_Configs tables
2. Instantiate and configure FC_TG, FC_N_Port, FC_Link and FC_Switch blocks
3. Connect FC_TG block to the N_Port. Configure FC_TG block, N_Port, Switch and Config blocks
4. Run simulations and Analyze Reports

8.8.2.1 Step1

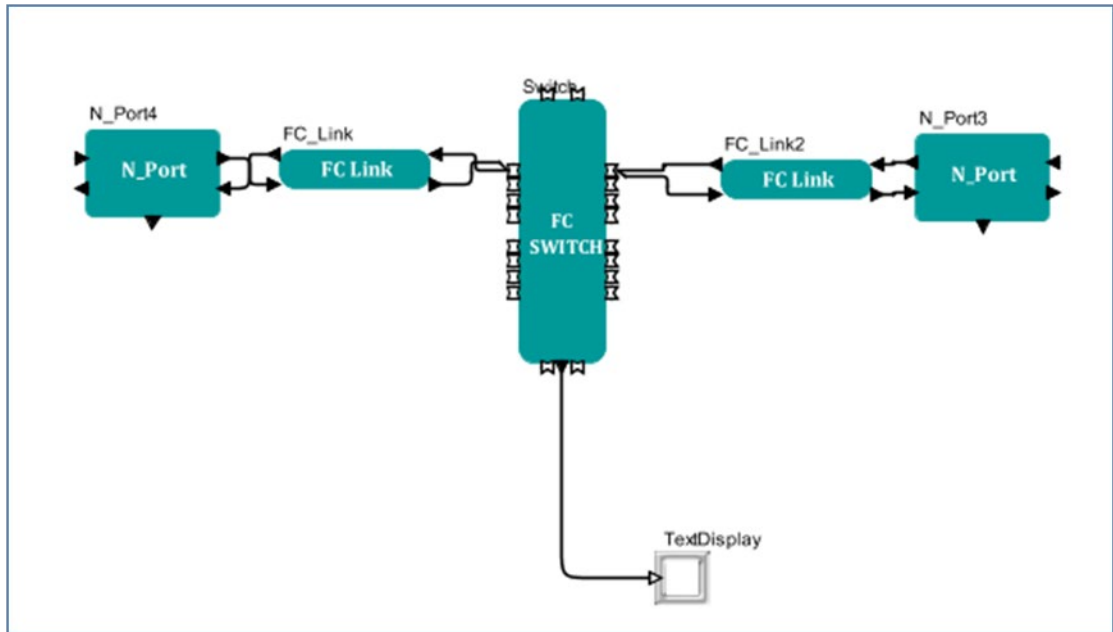
1. Instantiate FC_Config block from Hardware_Modeling → Emerging Bus Standards → Fibre Channel → FC_Config

Please update Device_Configs and Switch_Configs as shown below



8.8.2.2 Step2

1. In this step we shall add FC_N_Port, Switch and Link blocks
2. Drag two instances of FC_N_Port from the library pane Interfaces and Buses → Fibre_Channel → FC_N_Port
3. Drag two instances of FC_Link from the library Interfaces and Buses → Fibre_Channel → FC_Link
4. Drag one instance of FC_Switch from the the library pane Interfaces and Buses → Fibre_Channel → FC_Switch
5. Connect the Blocks as shown below



6. Double Click on N_Port (N-Port4 in the above figure) block and set ID as 1, double click on N_Port2 (N_Port3 in the above figure) block and set ID as 9.
7. Double Click on FC_Switch block and configure the block as below

Edit parameters for Switch

Block_Documentation:	Enter User Documentation Here
Switch_Name:	"FC_Fabric_1"
Ingress_Buffer_Size:	256
Egress_Buffer_Size:	256
Architecture_Name:	"Architecture_1"
Fragment_Size:	2112
Overhead_Bytes:	36
Enable_Debug:	true
Switch_ID:	2

Commit Add Remove Restore Defaults Preferences Help Cancel

8.8.2.3 Step3

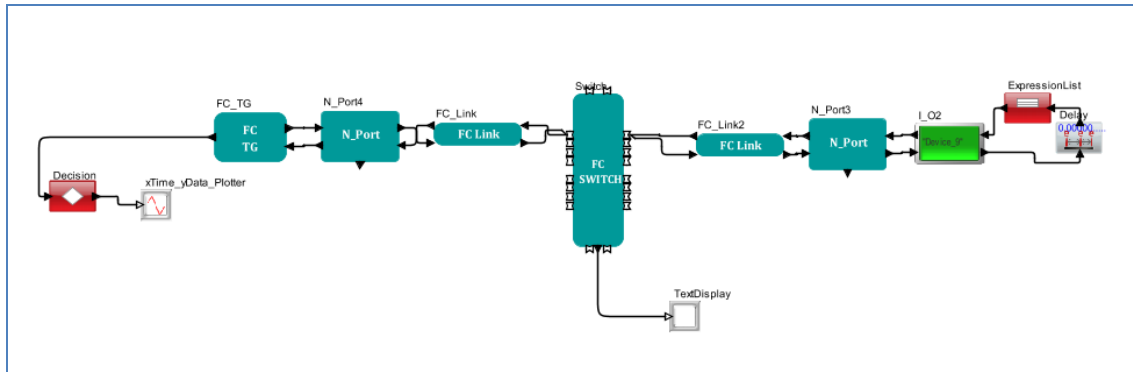
In this step we will be connecting traffic generator and modeling destination devices using VisualSim basic building blocks.

1. Instantiate FC_TG from the library pane Interfaces and Buses → Fibre_Channel → FC_TG
2. Double Click on FC_TG block
 - Configure the block parameter Device_Name as "Device_1"

- Configure the parameter Frame Details as below table

ID	A_Source	A_Destination	A_Message	A_Bytes	Trig_Start_Time
	Stop_Time	A_Command	A_Response	Mbps ;	
1	Device_Name	"Device_9"	"Class1"	6600	0.0
3	"Write"	false	1000.0 ;		1.0e-
2	Device_Name	"Device_9"	"Class1"	6600	1.0e-3
	10.0e-3 "Write"	false	1000.0 ;		
3	Device_Name	"Device_9"	"Class2"	6600	10.0e-3
	15.0e-3 "Write"	false	1000.0 ;		
4	Device_Name	"Device_9"	"Class3"	6600	15.0e-3
	20.0e-3 "Write"	false	1000.0 ;		

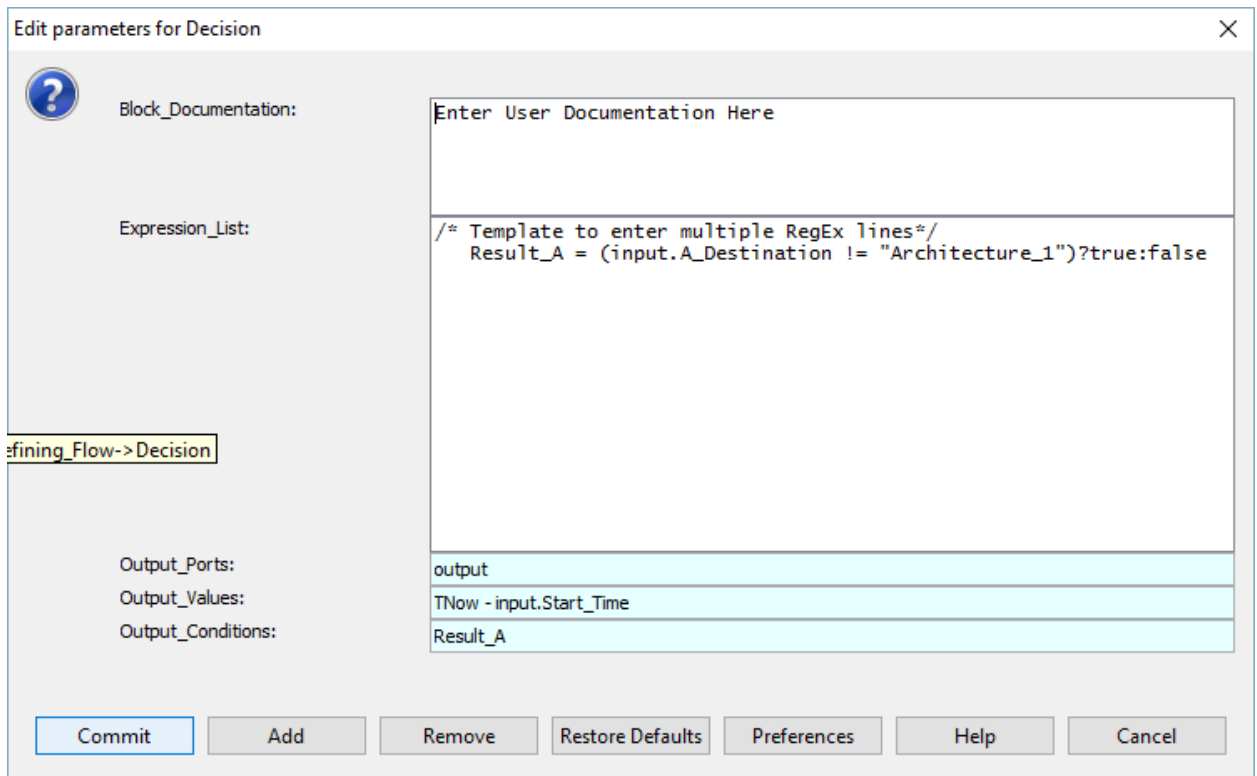
3. Instantiate Delay block from the library pane Delay and Utilities → Delay. Double click on the block and enter Delay_Value as 5.0e-8
4. Instantiate Expression_List or Processing block from the library pane. Define the expression as below under Expression
`input.A_Command = (input.A_Command == "Read")?"Write":"Read"`
5. Connect Delay block to output port of I_O2 block and input port of Expression_List or Processing block to Delay block and output port of Processing block to input port of I_O2 block.
6. Connect to_bus and frm_bus ports of I_O2 block to Device_In and Device_Out ports of FC_N_Port2 block.
7. Once all the connections are successful, VisualSim model will be as below



8.8.2.4 Step4

In this step we will configure model for capturing statistics and end-to-end latency

1. Connect data_out port of FC_TG block to a Decision block and configure the Decision block as below



2. Instantiate xTime_yData Plotter from the library pane Results → Plotter → xTime_yData plotter
3. Connect output port of Decision block to input port of plotter

8.8.2.5 Reports

Throughput Statistics

```

VisualSim Architect - file:/E:/VisualSim/VisualSim1540b_Jan...emo/Bus_Std/...
File Help
DISPLAY AT TIME ----- 9.97668440 ms -----
Buffer Ack Received @ 0.0099766844

DISPLAY AT TIME ----- 9.97668440 ms -----
Buffer Ack Received @ 0.0099766844

DISPLAY AT TIME ----- 9.97668440 ms -----
Delay Across FC_Fabric_1 is 6.7439999999926E-7

DISPLAY AT TIME ----- 9.97674440 ms -----
Fragment of Size 44 (with Overhead Bytes of 36 ) is sent to Ingress from Port 9 c

DISPLAY AT TIME ----- 9.97674880 ms -----
Buffer Ack Received @ 0.0099767488

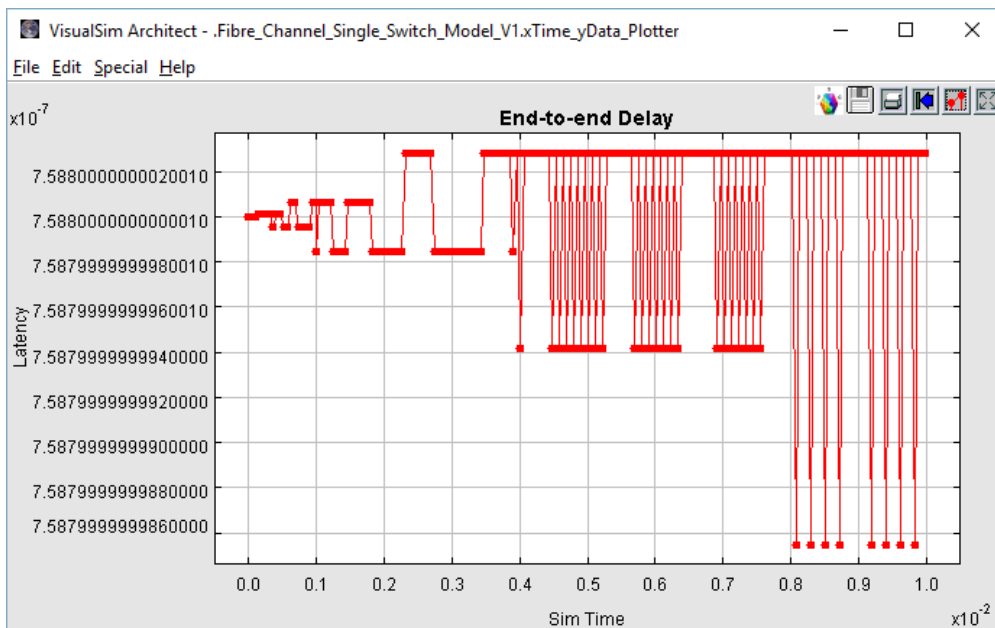
DISPLAY AT TIME ----- 9.97674880 ms -----
Buffer Ack Received @ 0.0099767488

DISPLAY AT TIME ----- 9.97674880 ms -----
Delay Across FC_Fabric_1 is 4.3999999997657E-9

DISPLAY AT TIME ----- 10.00000000 ms -----
{BLOCK              = "FC_Fabric_1_Stats",
 DELTA               = 0.0,
 DS_NAME            = "Switch Buffer Summary",
 ID                 = 1,
 INDEX              = 0,
 IOs_per_Second     = 95500.0,
 Mbs_per_Second     = 1008.4799999999999,
 TIME               = 0.01}

```

8.8.2.6 End-to-End Latency



8.8.2.7 Analysis

1. Change type of service by changing Class1 → Class2 → Class3
2. Increase/Decrease Packet size by changing A_Bytes value
3. Vary FC_Link Dly_Per_Mtr and Length_In_Mtrs
4. Modify FC_Switch Switch_Speed_MHz, Buffer Size, Overhead_Bytes

9 TTEthernet

9.1 Introduction

TTEthernet is a scalable, open real-time Ethernet platform used for safety-related applications primarily in transportation industries and industrial automation. TTEthernet sets new standards for flexibility, modularity and scalability in Ethernet-based systems. It is compatible to IEEE 802.3 Ethernet and integrates transparently with Ethernet network components.

TTEthernet has been designed for safe and highly available real-time applications, cyber-physical systems and unified networking. This technology offers deterministic real-time communication and TCP/IP Ethernet traffic in parallel on the same network. TTEthernet simplifies the design of fault-tolerant and high availability solutions. Its innovative technology consolidates experiences and proven mechanisms from aerospace system design, automotive electronics and industrial automation.

Three message types are provided:

1. **Time-Triggered** - Messages are sent over the network at predefined times and take precedence over all other message types. The occurrence, temporal delay and precision of time-triggered messages are predefined and guaranteed. The messages have as little delay on the network as possible and their temporal precision is as accurate as necessary. However, "synchronized local clocks are the fundamental prerequisite for time-triggered communication".
2. **Rate-Constrained** – Messages are used for applications with less stringent determinism and real-time requirements. These messages guarantee that bandwidth is predefined for each application and delays and temporal deviations have defined limits.
3. **Best-Effort** – Messages follow the usual Ethernet policy. There is no guarantee whether and when these messages can be transmitted, what delays occur and if messages arrive at the recipient. Best-effort messages use the remaining bandwidth of the network and have lower priority than the other two types.

VisualSim TTEthernet library is completely compliant with TTEthernet specification. TTEthernet library package includes TTEthernet node, Traffic Generators, TTEthernet Bridge and Statistic generators. In addition to the standard library modules VisualSim also has a preconfigured TTEthernet Configuration tables which includes routing table, Bandwidth allocator, stream table, VLAN table and Traffic generator table.

VisualSim TTEthernet library can be used to design a completely new TTEthernet based system or integrate with a system in which multiple different protocols are existing.

9.2 About TTEthernet Library

TTEthernet implements time-triggered communication mechanisms to provide a deterministic communication service over Ethernet. These time-triggered mechanisms establish and maintain a global time, which is realized by the synchronization of the local clocks of all TTEthernet devices. The global time is used as basis for implementing temporal partitioning, precise diagnosis, efficient resource utilization, or composability. The global time service of TTEthernet can be compared with the IEEE 1588 Precision Time Protocol, but uses multiple active "grandmaster clocks" to form a fault tolerant synchronization group with no need for reelection in case of an active grandmaster clock failure.

In order to support integration of applications with different real-time and safety requirements in a single network, TTEthernet supports three different traffic classes:

- Time-Triggered (TT) traffic – is sent in a time-triggered way, i.e. each TTEthernet sender node has a transmit schedule, and each TTEthernet switch has a receive and forward schedule. This traffic is sent over the network with constant communication latency and small and bounded jitter.
- Rate-Constrained (RC) traffic – is sent with a bounded latency and jitter ensuring lossless communication. Each TTEthernet sender node gets a reserved bandwidth for transmitting messages with the RC traffic. No clock synchronization is required for RC message exchange.
- Best-Effort (BE) traffic – traffic with no timing guarantees. BE traffic class compatible with the IEEE 802.3 standard Ethernet traffic.

The TTEthernet frame format is compatible with the standard Ethernet (IEEE 802.3) frame format. TTEthernet operates at the OSI model Layer 2, and allows the usage of existing layer 3 and upper layer protocols on top of TTEthernet.

TT messages are used for deterministic communication. All TT messages are sent over the network at predefined times and take precedence over all other traffic types. TT messages are optimally suited for communication in distributed real-time systems, specifically for safety related by-wire systems that require running of tight control loops over the network. TT messages allow designing of strictly deterministic distributed systems, where the timing properties of data flows are known in advance. Switches in TTEthernet have the central role of handling of communication data. TT messages are handled in the switch according to a predefined schedule. Precise planning at the time of system design precludes resource conflicts at runtime. Due to the predefined transmission time of a message, it is possible to reserve the medium and avoid even minimal delays of transmission if this is required for a specific TT message.

RC messages are used for applications with less stringent determinism and real-time requirements. RC messages limit the usage of bandwidth for each application (sender), and thus allow determining upper bounds for message delays and jittering. RC messages can be used for critical applications that depend on highly reliable communication and have moderate communication latency and jitter requirements. Typically, RC messages are also used for audio/video (streaming) applications.

In contrast to TT messages, RC messages are not sent with respect to a system-wide synchronized time base. Hence, different communication controllers may send RC messages at the same point in time to the same receiver. As a consequence, the RC messages may queue up in the network switches, leading to increased transmission jitter. As the transmission rate of the RC messages is bound a priori and controlled in the network switches, an upper bound on the transmission jitter can be calculated off-line and message loss due to buffer overflow or message timeouts is prevented.

If TT messages are to be transmitted via the same outgoing port of a switch at the same time, the TT messages take priority over the RC messages. TT messages can delay RC messages. RC messages are transmitted when no planned transmission of TT messages is pending and the sender observes the minimal transmission distance. The switch is responsible for arranging RC messages queued at an outgoing port. BE messages follow a method that is well-known in classical Ethernet networks. The network handles the messages in the best-effort manner, and therefore there is no guarantee whether and when these messages will be transmitted. BE messages use the remaining bandwidth of the network and have less priority than TT and RC messages. All legacy Ethernet traffic (e.g. internet protocols) can be mapped to this service class. RC and TT messages take precedence over BE messages at the same outgoing port of a switch. The switch uses the remaining bandwidth for BE messages if no TT or RC messages are to be transmitted. BE messages are transmitted after all pending RC messages, thus the remaining bandwidth is exploited in an optimal way. Tools are used to design and verify a TTEthernet system in advance. This ensures that the bandwidth for TT and RC messages is always sufficient according to the requirements of the application and interrupts are reduced to a minimum. Later incremental changes of the system configuration are possible.

9.3 Synchronization

Clock synchronization among all participants is crucial for the transmission of TT messages. TTEthernet components always transmit clock synchronization messages to keep the clocks of the end systems and switches in synchronization. For this purpose TTEthernet relies on a redundant hierarchical master-slave method that has a distributed fault-tolerant majority of master nodes and master switches to provide the time in the system. This method is unique for TTEthernet and can be combined with other mechanisms such as IEEE 1588. TTEthernet takes a two-step approach to synchronization. In the first step, the synchronization masters send protocol control frames to the compression masters. The compression masters then calculate an averaging value from the relative arrival times of these protocol control frames and send out a new protocol control frame in a second step. This new protocol control frame is then also sent to synchronization clients. TTE_Node block acts as the synchronization master and TTE_Bridge block acts as compression master block in the network.

9.4 System Level Model

System Level Simulation model of a TTEthernet based system can be classified as three subsystems. A Subsystem that defines the TTEthernet End System, a subsystem that defines the TTEthernet Switch and a Subsystem to capture analysis reports to make design decisions. VisualSim TTEthernet Library package provides these subsystem modules as preconfigured parameterized library blocks to quickly assemble complex network architecture with multiples of End Systems and Switches. To illustrate this here we have a graphical Representation of a TTEthernet based system is shown in figure1.

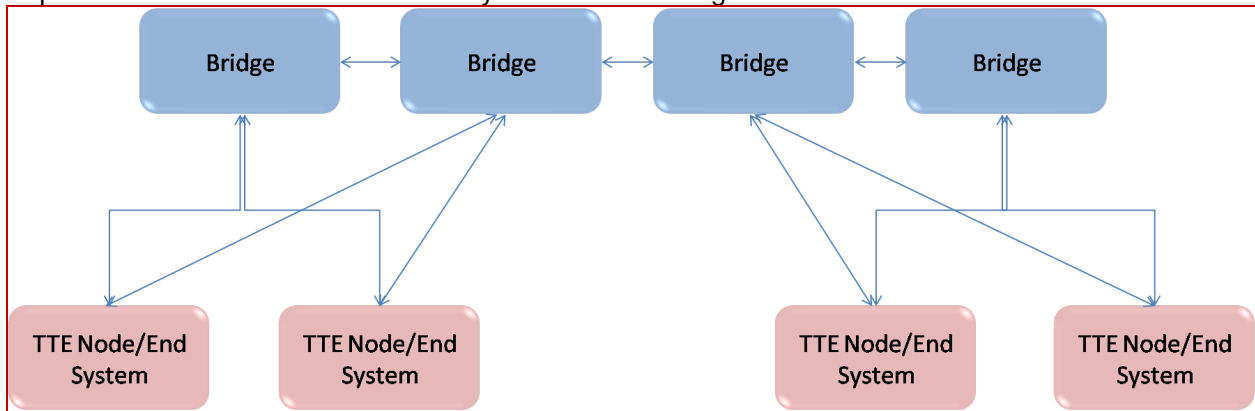


Figure 1: Block Diagram of a TTEthernet based System

VisualSim model of the proposed system architecture as defined in figure 1 is shown in figure 2 below.

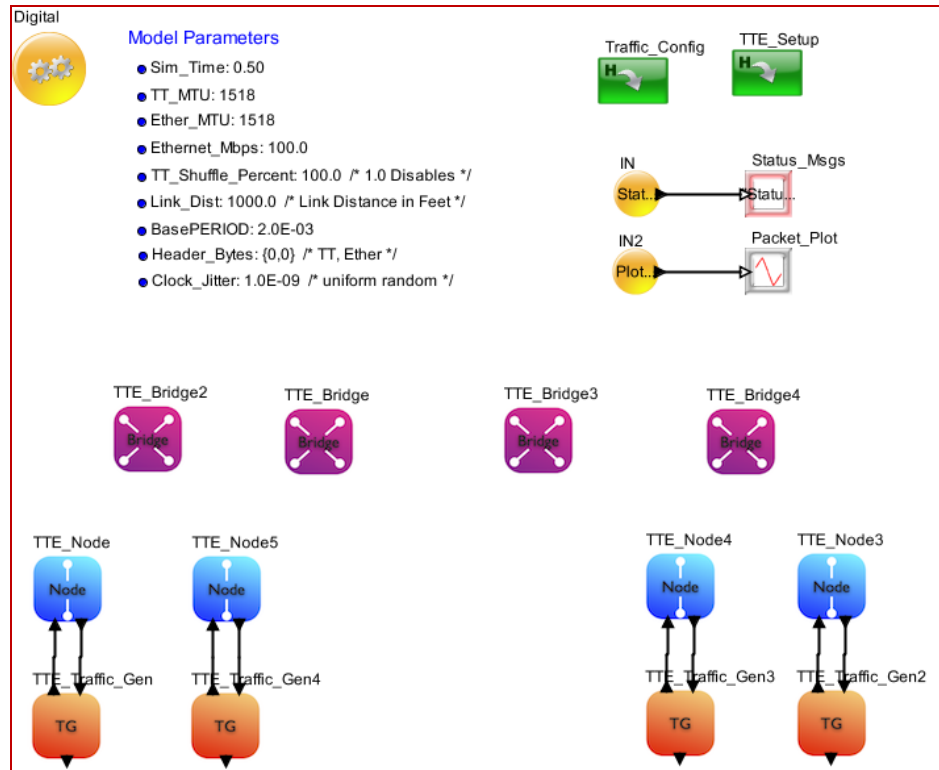


Figure 2: VisualSim Model

Please note that TTE_Node and TTE_Bridge blocks are connected virtually using named mapping.

9.5 Model Parameters

During model construction, TTE libraries require few Model Parameters to define Bandwidth, Base Period, Jitter and Shuffle percent. List of parameters are provided below

TT_MTU = 1518

Ether_MTU = 1518

Ethernet_Mbps = 100.0

TT_Shuffle_Percent = 100.0

Link_Dist = 1000.0

BasePERIOD = 2.0e-3

Header_Bytes = {0,0}

Clock_Jitter = 1.0e-9

TT_MTU :- TT Ethernet Maximum Transmission Unit. TT MTU is the size (in bytes) of the largest TT protocol data unit that the layer can pass onwards. Here we have set the maximum size as 1518 Bytes

Ether_MTU:- Ethernet Maximum Transmission Unit. Ethernet MTU is the size (in bytes) of the largest protocol data unit that the layer can pass onwards. Here we have set the maximum size as 1518 Bytes

Ethernet_Mbps :- Data rate for Ethernet is set to 100 Mbps

TT_Suffle_Percent: - Suffle percentage for TT Messages.

Link_Dist :- TTEthernet Link distance in feet

BasePERIOD:- Base period determines the communication cycle for the slots

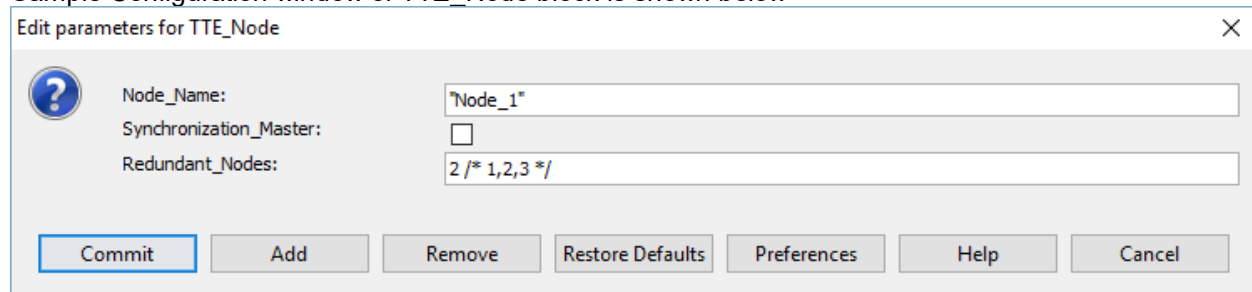
Header_Bytes:- TTEthernet and Ethernet header bytes. First array position is for TTEthernet and the second position is for Ethernet.

Clock_Jitter:- Clock Jitter value

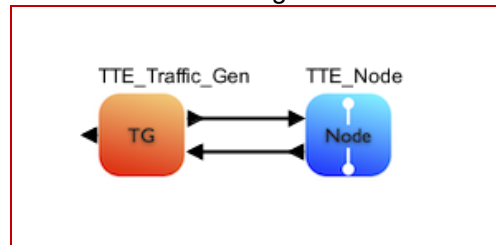
9.6 TTEthernet Node

TTEthernet Node block models TTEthernet End Systems and Synchronization master in the network. This block manages triple redundancy, multicast, MAC and Phy layers. Time-Triggered Ethernet functionality is implemented based on the SAE AS6802 specification of the Layer 2 Quality-of-Service (QoS) enhancement for Ethernet networks. The library provides capability for deterministic, synchronous, and congestion-free communication, unaffected by any asynchronous Ethernet traffic load. The block supports the isolation of the synchronous time-critical dataflows from other asynchronous Ethernet dataflows. This implementation of the standard enables designers to test, define the topology and validate their architecture for performance latency, throughput, jitters and other intrusions such as large, high priority packets, rate-controlled data streams and virtual LANs. This means that distributed applications with mixed time-criticality requirements (e.g., real-time command and control, audio, video, voice, data) can be integrated and coexist on one Ethernet network. Traffic generators can be connected to TTE Node to model an end system connected to network.

TTE Node block can be configured using its three parameters namely; Node_Name, Synchronization_Master enable/Disable, Redundant_Nodes. Node name must always start with Node_ superseded with integer value. Name of the Node must be unique. Designer can enable or disable synchronization master using the check box or Boolean conditions. As TTE Supports Redundancy, designer can select how many redundant connections a Node can have with a Bridge, ViusalSim TTE Node block supports upto 3 levels of redundancy. Sample Configuration window of TTE_Node block is shown below



Time Triggered or Rate Constrained or Best Effort traffic generators can be connected to TTE Node as below.



9.7 TTE Bridge

TTEthernet Bridge is responsible for performing partitioning among time-triggered, rate-constrained, and best-effort Ethernet traffic. High-priority time-triggered messages are routed through the bridge according to a predefined schedule with pre-definable constant latency and jitter in the sub-microsecond range. Rate-constrained messages are passed on according to the respective guaranteed bandwidth reservations. Finally, best-effort Ethernet messages are forwarded when bandwidth is available.

Based on the contents of the Type Field of an incoming message, the Bridge decides whether an incoming message is a standard Ethernet (BE) message or TT Ethernet message. BE Ethernet messages and TT Ethernet messages are handled differently by the bridge. Arriving standard Ethernet (BE) messages are stored in a BE-message queue of the bridge. The message, which is at the end of the message queue, is forwarded to the specified receiver address whenever the outgoing channel to this receiver is free. If an outgoing BE message is in the way of an incoming TT message, then the bridge immediately clears the channel (preempts the BE message) for the pending TT message. Immediately after the TT message has terminated, the bridge retransmits the (previously preempted) BE message, and, if the transmission was successful, releases the message buffer occupied by this message for a new incoming BE message. This autonomous TTEthernet

protocol mechanism uses any bandwidth that becomes free for the immediate transmission of BE-messages. It optimizes the throughput without any need for an explicit scheduling action and thus simplifies the schedule design and the system operation.

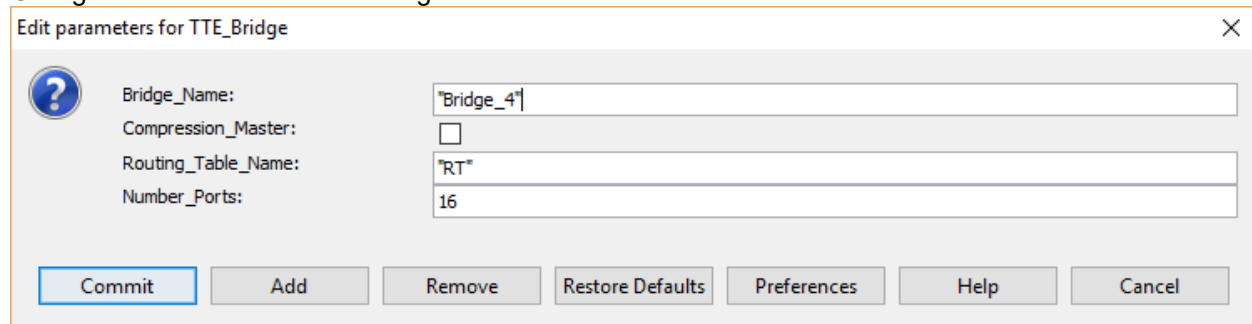
Arriving TT Ethernet messages are not stored in the bridge. They are delayed by the bridge for a defined number of μ seconds (clearance delay, which is the time needed to clear the transmission path in the case of a preempted BE message) and then forwarded in to the addressed receivers.

If the TTE Bridge block is configured as compression master, then the compression master calculate an averaging value from the relative arrival times of these protocol control frames and send out a new protocol control frame in a second step. This new protocol control frame is then also sent to synchronization clients.

TTE_Bridge is provided with an option to consider itself as a compression master or not. Compression master consolidates all the timing signals sent by Synchronization masters and then it sends the message back to all the nodes connected to it on the timing. This ensures that all devices and switches on the network have the same clock.

Name of the TTE Bridge block should be unique and it should always begin with Bridge_ and superseded with integer value. This block takes three parameters, Bridge Name, Compression Master (Enable/Disable), Routing Table Name and Number of End System Connections. Maximum of 16 nodes can be connected.

Configuration window for TTE Bridge is shown below



9.8 TTE Config

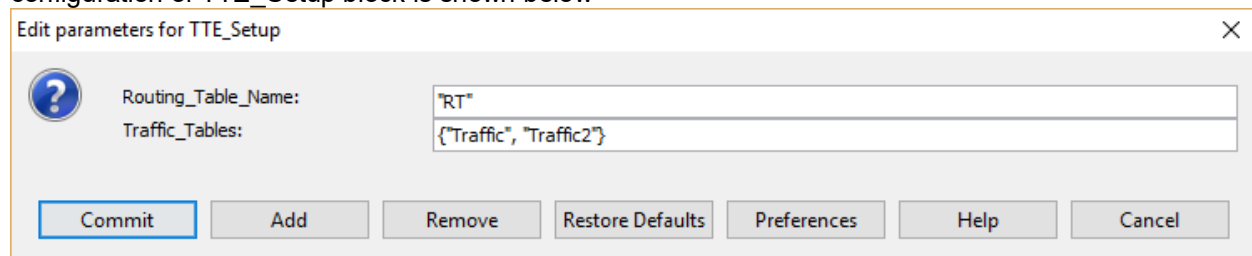
TTE Config block contains required Configuration Tables and local memory variables to setup a successful TTEthernet network and. Designer can add this block to the model and modify the values of each Table.

Designer can also add as many of the Traffic Table blocks to generate TT or BE or RC traffic.

While configuring the configuration table users must edit these tables by performing Open Instance on TT_Config block as opposed to Open Block. For Detailed explanation of each configuration tables please refer the [tutorial](#) section

9.9 TTE_Setup

TTE Setup block is required to processes all the data and transfers within each Node and Bridge block. Block has just two parameters, parameter RT should be configured with the name of the Routing Table available in TTE_Config block and the parameter Traffic_Tables should have the list of Traffic tables as a array. Sample configuration of TTE_Setup block is shown below



9.10 TTE_Stats

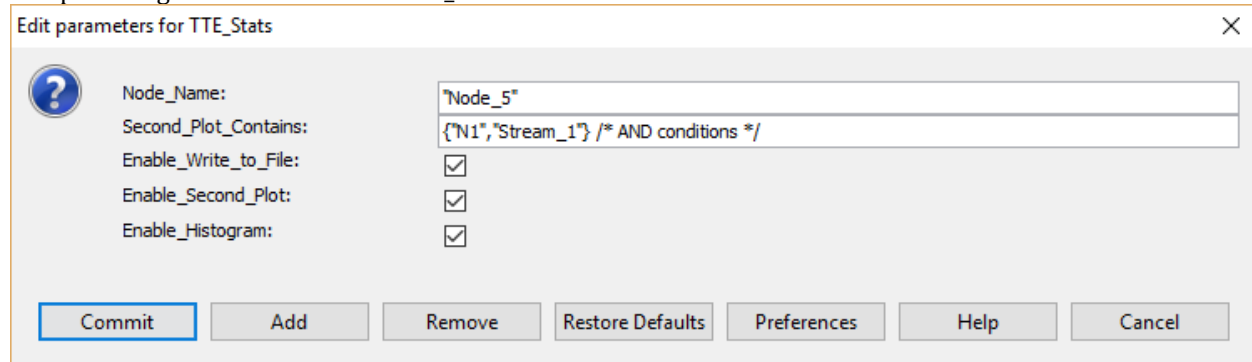
This block provides the statistics for all streams terminating at this Node. This block can be connected either to the Node block or the Traffic Generator.

This block provides 2 types of statistics- two files that contain the trace and statistics for all the streams, and 3 possible plots- latency of all streams, histogram of all the streams and a single stream plot.

Both text files are written into the directory containing the model. The first text file is named; TTE_Example_Stream_Plot.Stats.Node_5_Stream_Latency.txt and the second is Information_Warning.txt. The first file contains the minimum, maximum, mean and standard deviation of the latency, and the throughput for all the streams terminating at this Node.

The all stream latency is displayed by default. The legend on right is of the format- Start Node, End Node and Stream or Ethernet number. The Ethernet number is the order in the Traffic Table. The stream number is the ID. The histogram plot shows the histogram for each stream. The single stream latency plot can be enabled by setting Enable_Second_Plot parameter to true. The second plot will plot a single stream that is set in Second_Plot_Contains parameter.

Sample configuration window for TTE_Stats block is shown below



9.11 TTE_Traffic

The block accepts a traffic table and uses this to generate data. If the Identifier field contains Ethernet for Best Effort Traffic, "TT1" or "TT2" or "TT3" and so on for Time Triggered Traffic and RC1, RC2, RC3 and so on for Rate Constrained traffic, all data structures after the first one are sent out without waiting for a response. If the Identifier field has any other value, the block sends a data structure, waits for a response and then starts transmitting. After this first acknowledgment, no additional packets need to be acknowledged. All traffic is sent out via the net_tg_out port. All responses and traffic destined for this Node are sent to the net_tg_in port.

The data_out port sends out all data structure that terminate at this block. This can be connected to a plotter for latency plotting. This block will also support UDP Multicast traffic generation.

For Detailed explanation TTE_Traffic block please refer the [tutorial](#) section

9.12 Tutorial

Multiple demonstration systems are provided with documentation to guide a user to learn the TTEthernet modeling environment and create executable models.

A demonstration system is provided with explanation on the use of this library. The block diagram is provided below:

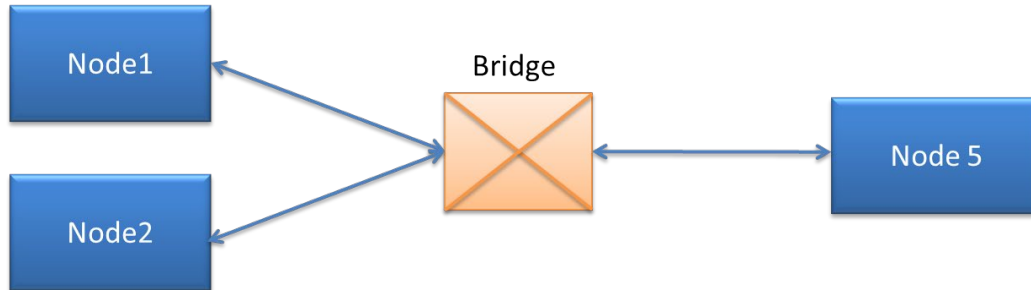


Figure 1.0 Block Diagram of Simple TTEthernet Network

Node 1 and Node 2 are the synchronization masters, Node 5 acts as a Synchronization client and Bridge acts as a compression master in figure 1.0. Each Source node's (Node1 & Node2) or Synchronization masters can generate messages of Time-Triggered (TT), Rate-Constrained (RC) or Best-Effort (Ethernet). Node 1 and Node 2 transmit messages to a single client Node5 which is connected by a Bridge in between.

The VisualSim model of this design is at **VS_AR/demo/networking/TTEthernet/TTE_Simple_Example.xml**

Each node consists of one traffic generator which can generate messages of TT, RC and Ethernet; and a node block. The bridge uses the TTE_Bridge block. As Node_5 is a client, we have connected TTE_Stats to it to generate network statistics and stream latency. The TTE_Config_Table and TTE_Setup blocks are added to this block diagram editor to define network attributes. Users have to define few top level model parameters and are given below

Ethernet_Mbps = 100.0

Link_Dist = 1000.0 /* Link Distance in Feet, user can modify this */

BasePeriod = 2.0e-3

Header_Bytes={0,0}

TT_Shuffle_Percent = 100.0

Ethernet_MTU=1518

TT_MTU=1518

The Digital simulation is added with a stop time. When you run the simulation, there are four outputs. There are two plots- one displaying the latency for all the streams and the other is Histogram of the latency. There are two text files that are written at the end of the simulation- one contains information and warning messages, and the other contains the latency statistics for each stream.

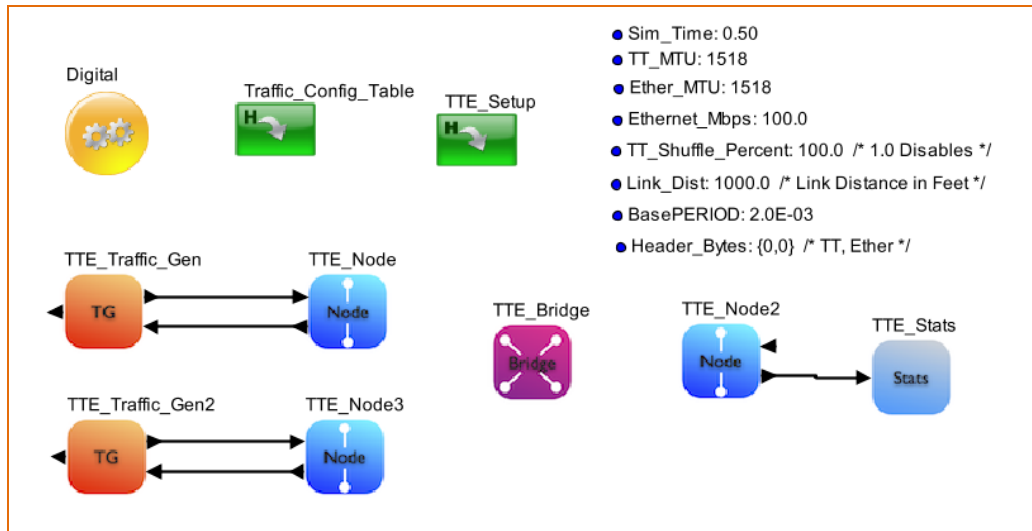


Figure 2.0 VisualSim Model

9.12.1 Basic Rules

1. All the Source and Destination node names must start with Node_ followed by a number
2. All Bridges must start with Bridge_ followed by a number
3. Each Traffic Generator will have a separate Traffic Table
4. Bandwidth credit and Class type for all streams from all nodes are listed in a single Stream Table
5. TTE_Setup block must be present in all models using TTEthernet libraries
6. The TTE_Config_Table is only an example of the required Tables. The user must modify all the Tables to setup the required configuration. Additional Traffic Tables must be added to cover all the Nodes and need to make sure TT_Config_Table is configured correctly.

9.12.2 Construction Steps

To construct a TTEthernet model, there are five steps

1. TTE_Config_Table configuration
2. Pre-Process block called TTE_Setup
3. Instantiate Nodes and Bridges
4. Add the traffic for each Nodes and define mode of message transmission.
5. Attach Statistics block to each of the nodes that are receiving the data.

9.12.2.1 Step 1- Model Configuration1

1. Drag **Interfaces and Buses ->TimeTriggeredEthernet -> TTE_Config_Table** into block diagram editor this provides the list of sample Tables that are required for the TTEthernet model.

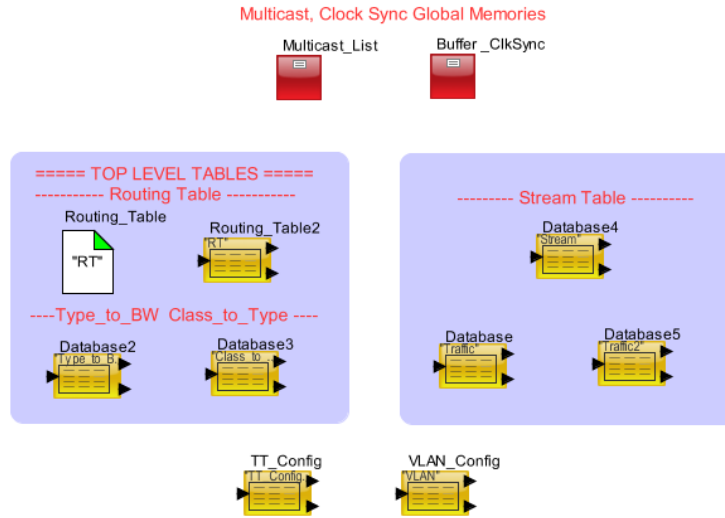


Figure 3: TTE_Config_Tables

1.1 **Roting_Table (RT)**: Configure based on your network requirements. This is required to handle the routing between nodes, compute latency on the links and maintain statistics. For this tutorial configure RT as below

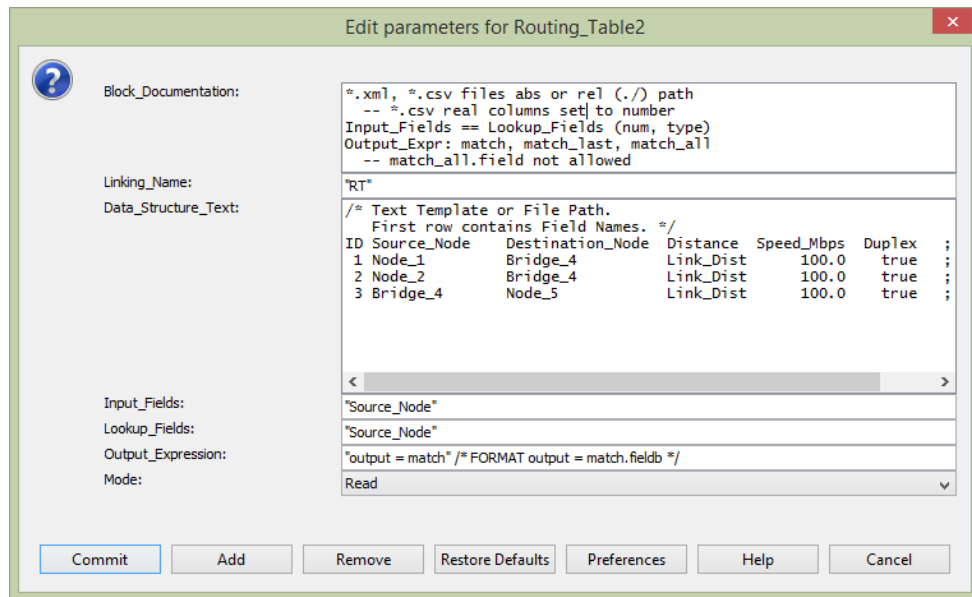


Figure 4: Routing Table Configuration

1.2 **Buffer_ClkSync**: No change required. Buffer_ClkSync is a Memory_Init block. This contains a set of required parameters for TTE. The following is the list of variables.

Percent_BW_global 0.75 ; ---Maximum allocation to Class A or B of the total bandwidth for that Type

Default_ProcTime_global 1.0E-06; Default Processing time for node processing.

Max_Buffer_Pkts_global 256 ; Maximum buffer packet size.

Max_Pkt_Bytes_global 1500 ; ---Maximum packet size *Max_Buffer_Pkts_global 256* Maximum number of buffers at each Node. Each buffer can hold one Data Structure or packet



Grand_Master_global "Node_1"; ---Node Name of the Master for the Clock Sync.
Grand_Master_Rate_global 200.0E-03; ---Rate of Clock Synchronization messages from the Grand Master Node

1.3 **Multicast_List:** Please make sure that you have required source and destination names in the table

1.4 **Traffic Table:** One table is required for each Node. Enter the list of TT, RC and Ethernet streams. One sample table is provided. Copy this instance to generate the required number of traffic tables. Make sure to give each Table a unique Name for the Linking_Table_Name. The list of all the Traffic_Tables must be provided to the TTE_Setup block.

ID	Identifier	Task_Source	Task_Destination	Mbps	Task_Size	Start_Time	Stop_Time
1	"TT1"	Node_1	Node_5	TT_Mbps	TT_Bytes	3.0e-12	0.5
	UDP 1	7 ;					
2	"RC1"	Node_1	Node_5	0.5	64	1.5E-03	0.5
	UDP 2	6 ;					
3	"RC2"	Node_1	Node_5	0.5	128	2.0E-03	0.5
	UDP 3	6 ;					
4	"Ethernet"	Node_1	Node_5	0.3	64	2.5E-03	0.5
						UDP	4 5 ;

ID: Increasing sequence number

Identifier: This determines the message stream type

Task_Source: Name of this Source Node.

Task_Destination: Name of the final destination or listener.

Mbps: Generation rate of this traffic stream

Task_Size: Packet data size. Does not include headers or trailers

Start_Time: Time in seconds after the start of the simulation

Stop_Time: Time in seconds after the start of the simulation

Protocol: UDP or TCP

VLAN: This is VLAN ID and it enables users to select BAG

Type: Type of Class. Can be 0 to 7.

1.5 **Stream:** This table contains the Class and the requested bandwidth for all TTEthernet stream from all the nodes.

ID	Mac_ID	Identifier	SR_Class	Mbps ;
1	"a0:36:9f:0c:77:38"	"TTE1"	A	0.2 ;
2	"a0:36:9f:0c:77:38"	"RC1"	A	0.2 ;
3	"a0:36:9f:0c:77:38"	"RC2"	B	0.5 ;
4	"a0:36:9f:0c:77:38"	"Ethernet"	A	0.3 ;
5	"a0:36:9f:0c:77:39"	"TTE2"	A	0.2 ;

6	"a0:36:9f:0c:77:39"	"RC3"	A	0.2 ;
7	"a0:36:9f:0c:77:39"	"RC4"	B	0.5 ;
8	"a0:36:9f:0c:77:39"	"Ethernet"	A	0.3 ;
9	"a0:36:9f:0c:77:3A"	"TTE3"	A	0.2 ;
10	"a0:36:9f:0c:77:3A"	"RC5"	A	0.2 ;
11	"a0:36:9f:0c:77:3A"	"RC6"	B	0.5 ;
12	"a0:36:9f:0c:77:3A"	"Ethernet"	A	0.3 ;

ID: Increasing sequence number

MAC_ID: Physical address of the Node

Identifier: Unique ID for each TTE stream. This must match the Identifier in the Traffic Table.

SR_Class: Stream Reservation class of this TTE stream- A or B.

Mbps: Requested bandwidth

1.6 **Type_to_BW:** For each Bridge and Node in the model, the amount of bandwidth allocated to each Type is specified in this Table. Top Level parameter Bandwidth_Mbps determines Ethernet Speed and is common for all nodes.

Sample Format is as below

ID	Type	Mbps ;
0	0	0.0 ;
1	1	0.0 ;
2	2	0.0 ;
3	3	1.0 ;
4	4	1.0 ;
5	5	33.0 ;
6	6	25.0 ;
7	7	40.0 ;

ID: Increasing sequence number starting from 1.

Type: Type of Class. The values are 0 to 7

Mbps: Bandwidth allocation. The sum of all the rows of this column must not exceed the total bandwidth of the links.

Full Format is:

ID	Type	Node_1	Node_2	Node_3	Bridge_4	Bridge_5	Node_5 ;
0	0	0.0	0.0	0.0	0.0	0.0	0.0 ;
1	1	0.0	0.0	0.0	0.0	0.0	0.0 ;
2	2	0.0	0.0	0.0	0.0	0.0	0.0 ;
3	3	1.0	1.0	1.0	1.0	1.0	1.0 ;
4	4	1.0	1.0	1.0	1.0	1.0	1.0 ;
5	5	25.0	25.0	25.0	33.0	33.0	33.0 ;
6	6	25.0	25.0	25.0	25.0	25.0	25.0 ;

```
7      7      40.0  40.0  40.0      40.0      40.0      40.0 ;
```

ID: Increasing sequence number starting from 1.

Type: Type of Class. The values are 0 to 7.

Node_1, Bridge_1...: Bandwidth allocation. The sum of all the rows of each column must not exceed the total bandwidth of the link.

Please make sure that you have included all the available nodes in the model

1.7 Class_to_Type- For each Bridge and Node in the model, the mapping of Type to Class and B.

Sample table is as shown below

Common Format:

```
ID Class  Type  ;
0  A     4     ;
1  B     3     ;
```

ID: Increasing sequence number starting from 1

Class: A or B

Type: Type of Class. The values are 0 to 7. Common to all Nodes and Bridges in the model.

Full Format is:

```

ID  Class Node_1 Node_2 Node_3 Bridge_4 Bridge_5 Node_5 ;
0   A    4      4      4      4      4      4  /* Background */
1   B    3      3      3      3      3      3  /* Best Effort */

```

ID: Increasing sequence number starting from 1

Class: A or B

Type: Type of Class. The values are 0 to 7. Common to all Nodes and Bridges in the model.

1.8 TT_Config_Table: TT_Config table provides the timing precision for Time-Triggered messages. StartTime is computed based on the task size and the Network Speed. User should enter details for the complete flow, for example if Source node is Node_1 and the destination is Node_5 then all the Nodes and bridges involved in communication flow should be defined in the table.

Sample TT_Config_Table is shown below

```

Node      TTName  StartTime  BasePeriod  ProcTime ;
"Node_1"  "TT1"    1.25E-03  BasePERIOD  ProcTIME ;
"Node_2"  "TT1"    2.0E-03   BasePERIOD  ProcTIME ;
"Node_3"  "TT3"    3.0E-03   BasePERIOD  ProcTIME ;
"Bridge_4" "TT1"    2.5E-03   BasePERIOD  ProcTIME ;
"Bridge_4" "TT2"    3.0E-03   BasePERIOD  ProcTIME ;

```

```

"Bridge_4" "TT3" 3.5E-03 BasePERIOD ProcTIME ;
"Bridge_5" "TT1" 3.0E-03 BasePERIOD ProcTIME ;
"Bridge_5" "TT2" 3.5E-03 BasePERIOD ProcTIME ;
"Bridge_5" "TT3" 4.0E-03 BasePERIOD ProcTIME ;
"Node_5" "TT1" 3.5E-03 BasePERIOD ProcTIME ;

```

1.9 **VLAN:** No change is required. In this table based on the VLAN ID BAG is allocated. For the Rate Constrained traffic stream this table decides the allocated bandwidth for a given VLAN. User can have as many VLAN's and associated with BAG and LMax time. Sample VLAN table is shown below;

ID	VLAN	BAG	LMax ;
1	1	1.0e-6	256 ;
2	2	1.5e-6	512 ;
3	3	2.0e-6	1024 ;
4	4	2.0e-6	1024 ;
5	5	2.0e-6	1024 ;

9.12.2.2 Step2 – Model Configuration 2

In this step we will add the blocks required for pre-processing the tables. This block must be added to all TTE models or models that use the TTE blocks. This block does require the existence of the Traffic tables, Routing Table, Link Setup table, Type_to_BW table, Class_to_Type, TT_Config_Table table and Multicast Memory_Init block.

- Instantiate the **Interfaces and Buses ->TimeTriggeredEthernet->TTE_Setup block**.
- Link the `Routing_Table_Name` parameter to the top-level. This ensures that all blocks will use the same name. Example name is "RT"
- In the parameter 'Traffic_Tables', list all the traffic tables names as strings in array.

9.12.2.3 Step3- End Systems and Switches

Each transmitting node or synchronizing master node requires two blocks – **TTE_Traffic_Gen** and **Node block**. The assembly of the two blocks is as below

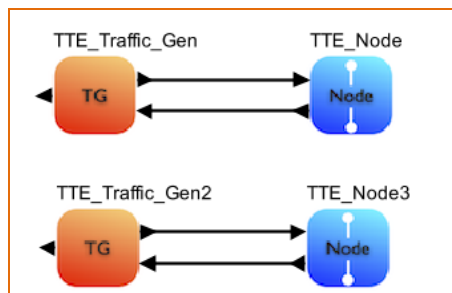


Figure 5: Assembly of Source Node

At the receiver side or client side only Node block is sufficient. To collect network activities and statistics one can connect **TTE_Stats** to the client node block. The assembly of the two blocks is as shown below

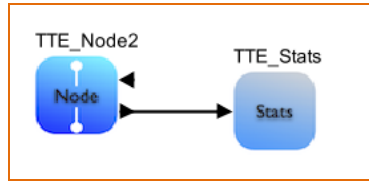


Figure 6: Assembly of Client Node/ Destination only node

9.12.2.4 Step4 – Defining Network Connectivity

The Bridge connects the Nodes to the network and also connects to other Bridges. The bridge block handles the Routing, broadcast of clock synchronization. A model can have a single bridge with all the Nodes connected to it.

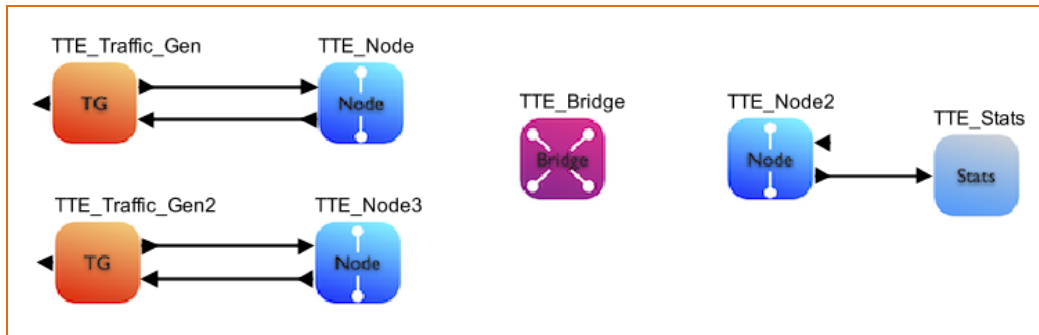


Figure 7 Single Bridge Network

If a model has multiple bridges, instantiate as **many TTE_Bridge** blocks and update the **Link_Setup**. The topology of the network is determined by the **Link_Setup** table. The Block Diagram shows the number of Bridge and Node instances.

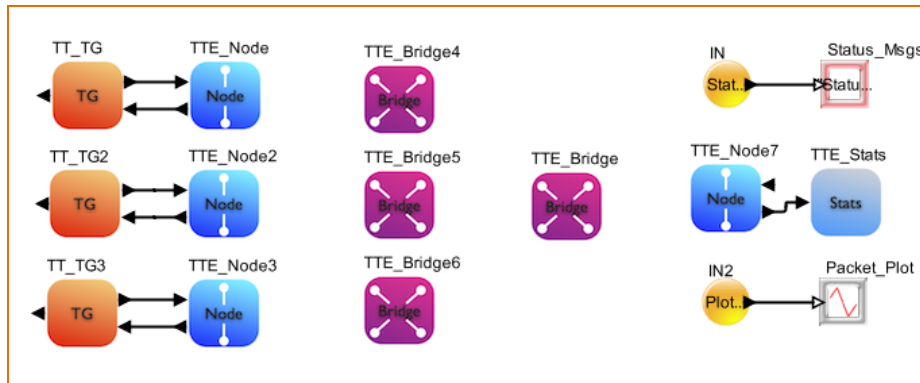


Figure 8 Multiple Bridge Network

The following is the sequence:

- Add as many **Interfaces and Buses** ->**TimeTriggeredEthernet** ->**TTE_Bridge** blocks as you have defined in the Link Configuration table
- Assign a unique name to each Bridge block. The rule here is that the name must be **Bridge_** followed by a number. No two Bridges can have the same
- For the parameter, add the **Routing_Table_Name**. For now it is just "RT".

9.12.2.5 Step5 - Reports

The **TTE_Stats** block can be connected directly to the Node out for a Destination-only or the data_out port of the **TT_Traffic_Gen** block for a Source+Destination block.

The **TTE_Stats** generates 3 different pieces of information. They are:

1. The latency statistics and through for all the streams arriving at Node to which this TTE_Stats is connected.

Time-Triggered	Rate Constraint
<p>N1_to_N5_TT1_1</p> <p>Latency</p> <p>Minimum: 2.5061367090 ms</p> <p>Mean: 2.5061367090 ms</p> <p>Std Dev: 29.1 ps</p> <p>Maximum: 2.5061367090 ms</p> <p>Throughput</p> <p>Mbps: 0.0328661952051</p>	<p>N2_to_N5_RC2_3</p> <p>Latency</p> <p>Minimum: 24.5134240 us</p> <p>Mean: 494.2624945 us</p> <p>Std Dev: 571.1953459 us</p> <p>Maximum: 2.0779767120 ms</p> <p>Throughput</p> <p>Mbps: 0.5011401721222</p>
Ethernet	
<p>N1_to_N5_Ether_4</p> <p>Latency</p> <p>Minimum: 14.2734240 us</p> <p>Mean: 519.6828018 us</p> <p>Std Dev: 617.3499543 us</p> <p>Maximum: 2.8099000580 ms</p> <p>Throughput</p> <p>Mbps: 0.3010222764019</p>	

Here

N1_to_N5- Node_1 to Node_5
 TT_1- Time Triggered stream with ID 1 in the Traffic Table
 RC2_3- Rate Constrained stream with ID 3 in the Traffic Table
 Ether_4- Ethernet stream with ID 4 in the Traffic Table
 Minimum- Lowest latency recorded during the simulation
 Maximum- Highest latency recorded during the simulation
 Mean- Average of all the latencies recorded
 Std Dev- Standard Deviation of the latencies

2. Graphical plot showing the Latency for all the streams arriving at the Node to which this TTE_Stats block is connected.

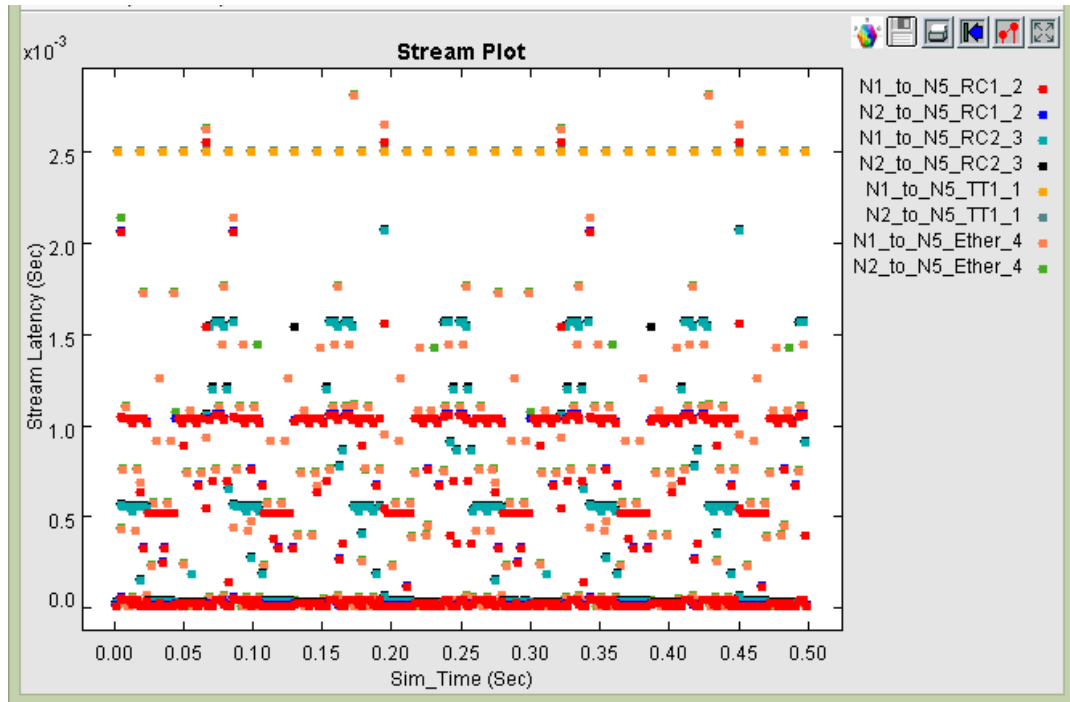


Figure 9: Stream Latency Plot

3. Histogram of the Latency for all the streams arriving at the Node to which this TTE_Stats block is connected.

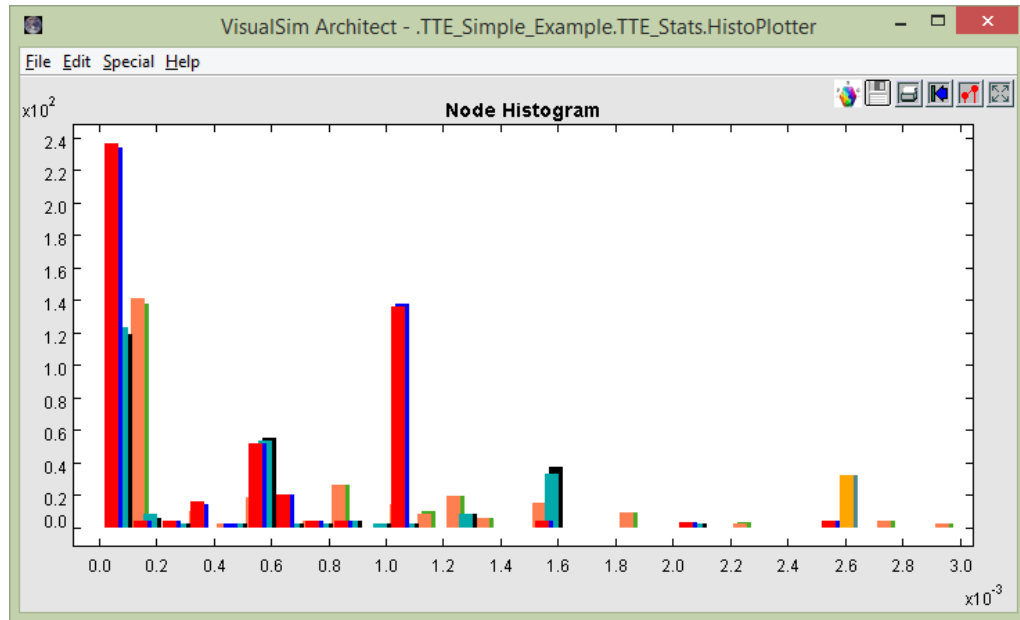


Figure 10 Histogram Latency plot

The following is the connection sequence:

- a. Connect an **Interfaces and Buses** ->**TimeTriggeredEthernet** ->**TTE_Stats** block to the output of each of the Nodes. This will capture all the statistics including the activity traces, latency and histogram

plots. You can see a unique latency graph at each Listener of Destination Node. We only need this block for Nodes that will receive data.

b. Enter the name of the Node that this block is connected.

9.12.2.6 Step6 – Analyzing System

Now the model has been constructed. The next step is to run the simulation. Each simulation run can have different setup values- such as Network Toplogy (Link_Setup file), traffic streams from each node (Traffic table), number of TT/RC/Ethernet streams (stream table) and allocation of bandwidth to each Type at each Bridge (Type_to_BW).

For each run, the statistics are provided in a combination of files and graphical plots. View Step 6 to see the list of plots and reports.

9.13 Advanced Tutorial

9.13.1 TTEthernet model with 8 Source Nodes

Let us make use of `TTEthernet_Simple_Example.xml` model to extend the system with additional 6 Nodes or total of 8 Nodes.

9.13.1.1 Extending System with Additional Nodes

1. Instantiate additional TT_Node's and provide unique name. Connect TT_Traffic_Gen blocks for each TT_Node's and provide unique name.

9.13.1.2 Step2: Configure TTE_Config_Table

1. Open TT_Config_Table block, create additional Traffic_Table's for all Source Nodes.
2. Update Routing Table (RT) with additional node details
3. Update additional node details to Type_to_BW and Class_to_Type blocks.
4. Update TT_Config table with updated Traffic_Table details (TT1, TT2, TT3...) and timing precision based on the network speed and task size.
5. VLAN Table – No modification is required
6. Stream Table – No modifications required

9.13.1.3 Step3: TTE_Setup

1. Update Traffic_Tables parameter with all additional Traffic_Table's names

9.13.1.4 Step4: Run Simulation

9.13.2 TTEthernet Model with 8 nodes and 3 Destination Nodes

In this tutorial we shall extend the model developed in advanced tutorial part 1.

9.13.2.1 Step1:

Open the model developed in Part-1 and instantiate additional destination nodes and TTE_Stats blocks. Provide unique names for TTE destination nodes.

9.13.2.2 Step2:

Open TTE_Config_Table block and update Routing Table(RT), Type_to_BW, Class_to_Type, TT_Config and Traffic Table. With Regards to Traffic table, user can decide which is the Destination node should be selected.

9.13.2.3 Step4: Run Simulation

Note: Final version of the model is available in
VS_AR\doc\Training_Material\Tutorial\WebHelp\Tutorial\Networking\TTE_8_Nodes_3dest_Nodes.xml

9.13.3 TTEthernet Model with 8 Nodes, 2 Bridges and 3 Destination Nodes

In this tutorial we shall extend the model developed in advanced tutorial Part2.

9.13.3.1 Step1:

Open the model developed in Part-1 and instantiate additional 1 TTE_Bridge block. Provide unique name.

9.13.3.2 Step2:

Open TTE_Config_Table block and update Routing Table(RT), Type_to_BW, Class_to_Type, TT_Config and Traffic Table. In the RT table one can select the flow, sample RT table is shown below

ID	Source_Node	Destination_Node	Distance	Speed_Mbps	Duplex
1	Node_1	Bridge_4	Link_Dist	100.0	true ;
2	Node_2	Bridge_5	Link_Dist	100.0	true ;
3	Node_3	Bridge_4	Link_Dist	100.0	true ;
4	Node_4	Bridge_5	Link_Dist	100.0	true ;
5	Node_5	Bridge_4	Link_Dist	100.0	true ;
6	Node_6	Bridge_5	Link_Dist	100.0	true ;
7	Node_7	Bridge_4	Link_Dist	100.0	true ;
8	Node_8	Bridge_4	Link_Dist	100.0	true ;
9	Bridge_4	Node_10	Link_Dist	100.0	true ;
10	Bridge_5	Node_10	Link_Dist	100.0	true

```
;  
11 Bridge_5      Node_11      Link_Dist  100.0  true  
;  
12 Bridge_4      Node_12      Link_Dist  100.0  true  
;
```

9.13.3.3 Step3:

Run Simulation

Note: Final version of the model is available in

VS_AR\doc\Training_Material\Tutorial\WebHelp\Tutorial\Networking\TTE_8_Nodes_3dest_Nodes_2Bridge.xml

10 IEEE1394/Firewire

10.1 Introduction

Mirabilis Design Inc provides IEEE 1394 or FireWire as a standard library in VisualSim Architect modeling environment. Using this configurable library, designers can architect next generation networking system for high performance systems by conducting early design space exploration, power and performance analysis.

FireWire Modeling library enables system designers and architects to conduct early design explorations of FireWire based System Architecture. Library is preconfigured to generate network statistics and understand network activities to analyze network behavior by varying workload and network configurations. As the FireWire library package includes Node, Link and also Integrated Traffic generator, designers can assemble simulation model of the proposed system very quickly. For example, if a designer wants to evaluate performance of FireWire network with a Camera with a resolution of 800x600 8- bits per pixel at 30 fps. Typical challenges that a designer may come across at this stage would be, how many such cameras can be connected to the network, Bandwidth allocation percentage for isochronous transfers, available bandwidth after connecting second camera with the same format, possible bottlenecks. VisualSim FireWire library enables designers to address these questions and also enables designers to evaluate system behavior with custom network configurations as well.

FireWire library is built to the specifications and is compliant with IEEE Std 1394™-2008 revision. Data transmission rates of S100, S200, S400, S800, S1600 and S3200 are supported.

10.2 About FireWire

The 1394 protocol is a peer-to-peer network. Every node of the network is a device equipped with the necessary functionality to perform configuration and control actions. Each network node may have multiple ports that act as a repeater. That means that packets received on some port of a node are forwarded to all other ports of the same node. The maximum number of devices on the bus is 63. According to the spec P1394.1 if there is a need to add more devices, then bridges can be used, raising the total number of supported connected devices on a 1394 network to 65536 (64 nodes on each bus and up to 1024 buses).

Data transfers between connected devices can be realized in two ways: *isochronous* or *asynchronous*. *Isochronous transfers* are always broadcast with either a transmitter and a single receiver or a transmitter and numerous receivers. During an isochronous transfer no error correction or retransmission is performed. Up to 80% of the available bandwidth can be used for this kind of transfers.

Asynchronous transfers, on the other hand, are not broadcasted but always target a specific network node using an explicit address. These transfers do not have a guaranteed bandwidth on the bus, but when asynchronous transfers are allowed, fair access on the bus is provided. For each asynchronous transfer, there is always confirmation for the integrity of data sent from the receiver to the transmitter.

10.3 About FireWire Library

VisualSim FireWire Library models details of Physical layer, link layer, transaction layer and Application layer. Application layer is responsible for bus management and bandwidth allocation activities. The library is classified into three parts, FireWire Node, FireWire Link and FireWire Config. FireWire Node block can behave as a Root Node, Intermediate node or a leaf node based on the position of node in a network topology. FireWire Link models the communication link between one node to another, parameters are provided to select type of communication medium and also the length. FireWire Config block holds details on the network configurations and routing.

Each FireWire node has a traffic generator and the parameters are provided to define destination node, start/inter-arrival time for generating arbitration request and data packets. Parameters are also provided to define the size of the data and transfer type (Isochronous or Asynchronous). This enables the designer to emulate network behavior and also to capture end-to-end latency, throughput and system bottlenecks.

Graphical representation of a sample FireWire network model structure in VisualSim will be as shown in figure 1.0

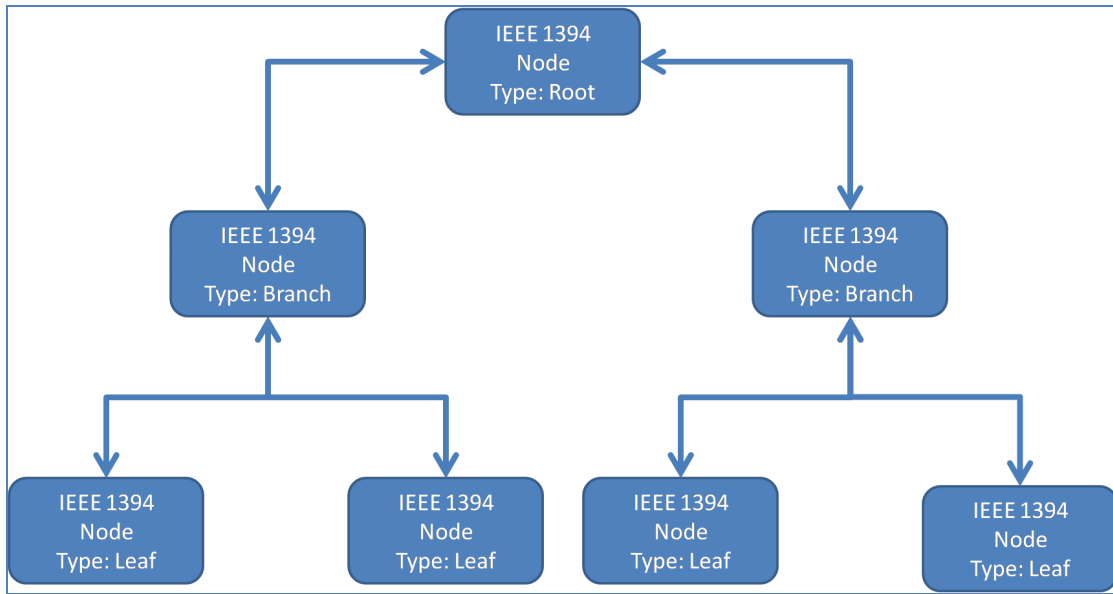


Figure 1.0

10.4 Model Parameters

During model construction, FireWire libraries require few Model Parameters to define FireWire network speed, Bandwidth Allocation Percentage and Name of the Root Node. Parameter is defined as below

Network_Speed = 400.0 /* Mbps */

ISO_BW_Percent = 80.0

Root_Node = "Root_5"

Parameter Network_Speed is used to determine the selected network data rate. VisualSim supports different transfer modes, S100, S200, S400, S800, S1600 and S3200

Parameter ISO_BW_Percent defines the Bandwidth Percentage allocated for Isochronous transfers.

10.5 Assumptions

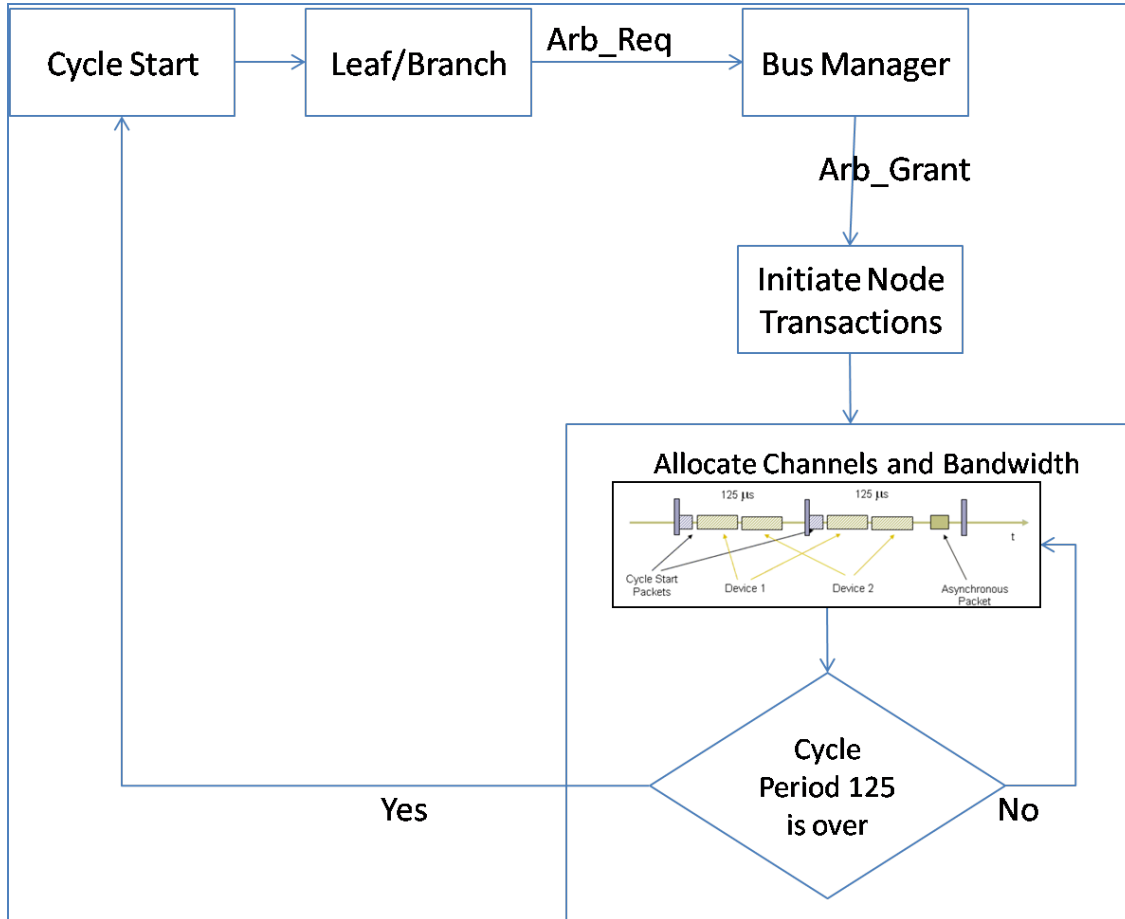
FireWire Model has few of assumptions as these factors will not affect system performance. We assume that topology is already defined and identification of Leaf Nodes, branch and Root nodes are completed and finally Self-Identification process is complete.

The self-identification phase begins with the root node sending an arbitration grant signal to its lowest numbered port. That port will assign itself ID 0 and propagate a self-ID packet up the chain to the root node. As it does, each node updates its internal ID counter by 1. After getting the ID, the root node will then continue to signal an Arbitration Grant signal to the currently lowest numbered port, and the process repeats itself. It continues until all ports on the root node have indicated a self-ID done condition. The root node then assigns itself a physical ID number 1 higher than the last number issued, always making itself the highest numbered device on the bus. During the self-ID process, parent and children nodes exchange their maximum speed capabilities.

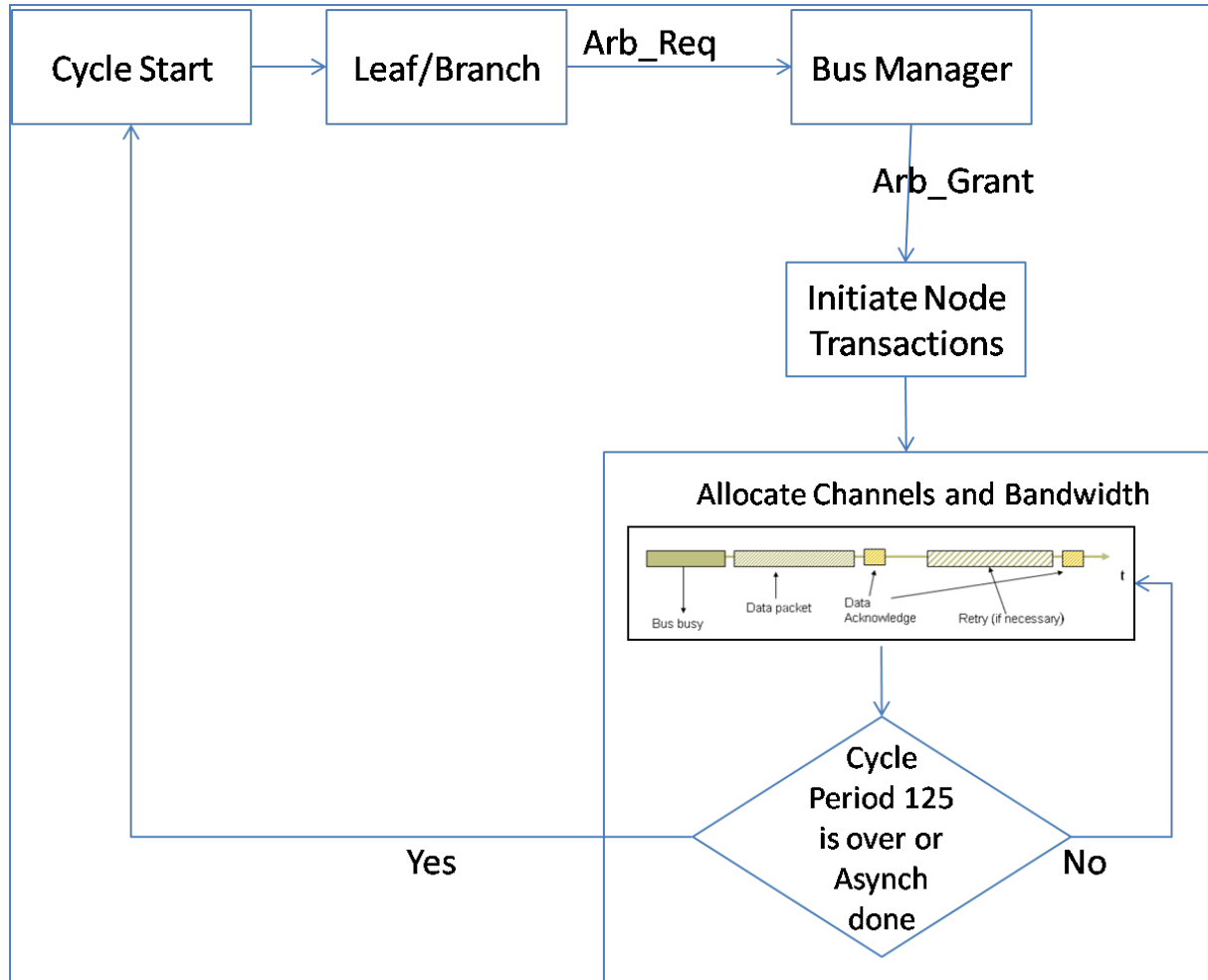
10.6 FireWire Node

FireWire Node block can behave as a root, branch or leaf node based on the node position in a network topology; Configurable parameter is provided to select the Node type. Each node will be assigned with a unique identifier and the root node must be provided with highest identifier number.

10.6.1 Flow Diagram for Isochronous Transfers



10.6.2 Flow Diagram for Asynchronous Transfers



10.6.3 Block Details

Current version of FireWire Library considers that the topology is already defined and associated identifiers are provided for all Leaf nodes, Branch nodes and Root Node. Leaf node will have only one connection to the node with the port identified as parent, whereas branch node will have more than one connections with the ports types as both parent and child connections. Root node will have only child connections and there will be no parent connections to it. FireWire Root node will be assigned with a highest identifier number. Root Node initiates Cycle Start signal and is broadcasted to all nodes down the stream and waits for acknowledgement from the leaf/branch nodes. The delay associated with Cycle Start signal will be reused for consecutive transactions. Right after the transmission of the *cycle start* packet, the devices that wish to transmit isochronous packets may arbitrate for the bus. According to the arbitration scheme used on the 1394 standard, whenever a device wishes to gain access to the bus, it sends a signal to its parent-node, which in turn forwards it to the upper layers of the bus's tree structure, till it reaches the root node. The root node decides which node will gain access to the bus on a first come first serve basis. It becomes obvious that the device located closest to the root node wins the arbitration.

Each device that wishes to transmit isochronous packets is assigned a logic channel by the Isochronous Resource Manager in Root Node. A channel may be used only once in every bus cycle. This ensures on one hand, that each device that has been granted a channel will transmit at some point during every cycle and on the other, that the devices located near the root node won't monopolize the bus. If a device has significant bandwidth requirements and there are available channels, then the IRM may grant this device more than one channels, enabling it to transmit multiple packets per clock cycle.

When the data of an isochronous channel have been streamed, the bus remains idle waiting for another isochronous channel to begin arbitration. If no other isochronous channel wishes to transmit data, the bus will remain idle for more time than the isochronous gap (the time interval between two isochronous transmissions with duration from 40 to 50 nsec). If the bus idle time exceeds the subtraction gap of 11.2micro seconds, the devices that wish to transmit asynchronous data may arbitrate for the bus.

The bandwidth in 1394 parlance is described in a time unit called the "Bandwidth Allocation Unit". One unit stands for the amount of time required to transmit one quadlet of data (4 bytes) at the S1600 data rate (1.6Gbps). This is the same time required to transmit 2 bytes of data at S800 (800Mbps) or 1 byte of data at S400 (400Mbps).

One unit is 2 bytes at S800, so 6144 units in a cycle mean 12288 bytes per cycle at S800. Multiplying this by 8000 cycles per second we get 98,304,000 bytes per second, or 786,432,000 bits per second, which is the nominal value of the S800 data rate (786.432 Mbps).

IEEE 1394 states that only 4915 out of the 6144 bandwidth units can be used for isochronous traffic. This is equivalent to 100 microseconds or 80 percent of the 125 microsecond cycle. The other 20 percent (or 1229 bandwidth units) should be left available for asynchronous traffic.

The Bus is divided into 125 micro seconds as all operations on the bus are synchronised using an 8kHz clock signal produced by the cycle master. Structure of isochronous transfer is shown in figure 2.0

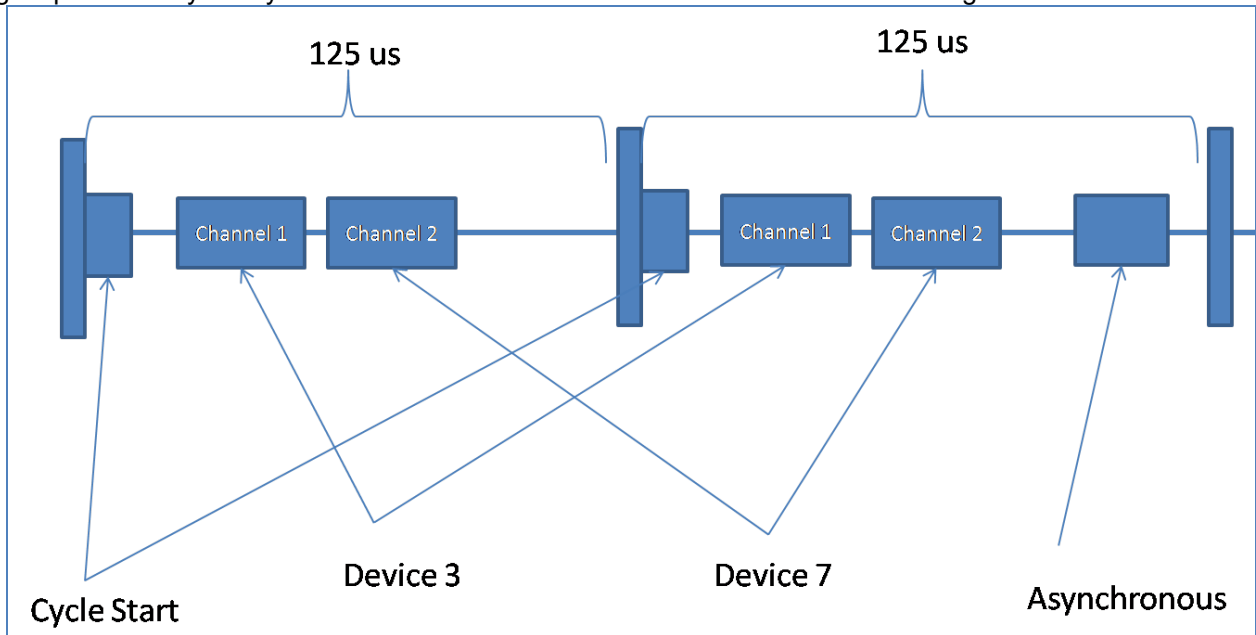


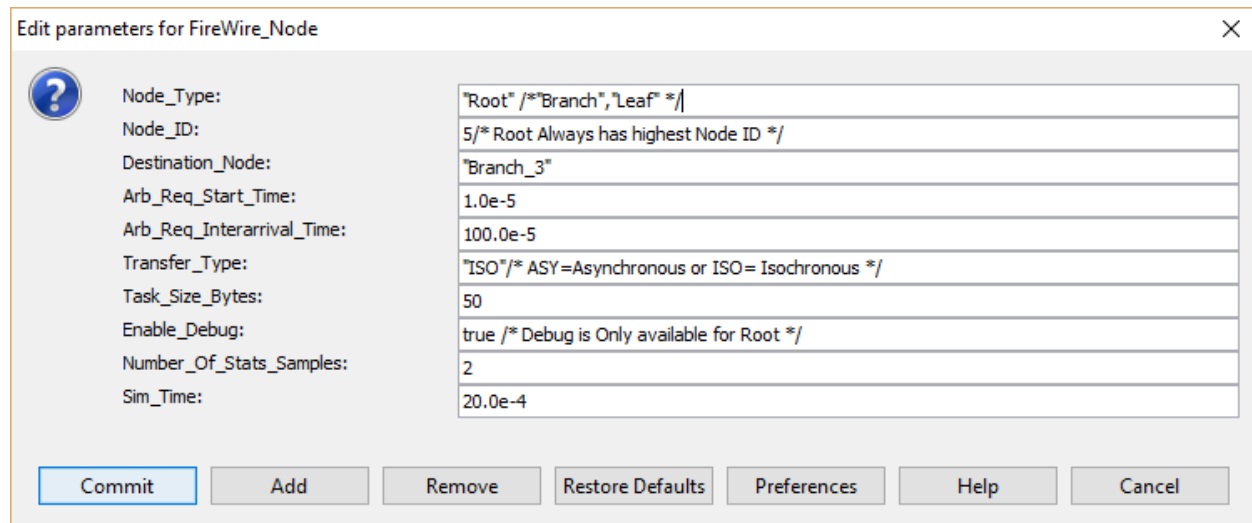
Figure 2.0

In contrast to the isochronous transfers, for the asynchronous transfers there is not a manager to assign transmission channels. As a result, the devices that wish to transmit multiple packets during the duration of a cycle, may arbitrate more than once for the bus. This may lead to unfairness for the devices that are located far from the root node, as the closest to the root devices are favored. To alleviate this problem, the standard defines the fairness interval and the arbitration rest gap. Each device that transmits asynchronous packets has a bit named arbitration enable bit. Depending on the value of this bit, the device is entitled or not to arbitrate for the bus during the current cycle. Initially, the bit is set to 1 and when the device completes an asynchronous transaction, it is cleared and the device may not participate in any arbitration process. This scheme ensures a fair access to the bus even for devices located far away from the root.

If the data being sent is time-critical and error-tolerant (like a video or audio stream), then choose isochronous mode to transfer the data. If the data being sent isn't time-critical and error-tolerant, then asynchronous mode is the right choice.

10.6.4 Block Parameters

Block parameters are provided to configure the FireWire Node library block for design requirements. FireWire Node block also has an internal traffic generator that generates Arbitration request messages and transfer data packets once arbitration is granted for the node. Block parameters available for FireWire Node block is as below



Parameter	Value
Node_Type:	"Root" /*Branch*, "Leaf" */
Node_ID:	5 /* Root Always has highest Node ID */
Destination_Node:	"Branch_3"
Arb_Req_Start_Time:	1.0e-5
Arb_Req_Interarrival_Time:	100.0e-5
Transfer_Type:	"ISO" /* ASY=Asynchronous or ISO= Isochronous */
Task_Size_Bytes:	50
Enable_Debug:	true /* Debug is Only available for Root */
Number_Of_Stats_Samples:	2
Sim_Time:	20.0e-4

Buttons: Commit, Add, Remove, Restore Defaults, Preferences, Help, Cancel

Node_Type – Type of the node must be set to either Root or Leaf or Branch based on the position of the Node in the Network.

Node_ID – Unique Identifier for the Node, Root node must be set to highest number

Destination_Node – Destination Node name

Arb_Req_Start_Time – Set the Start time for generating Arbitration Request message. Note that, this parameter will not be utilized if the Node Type is set to "Root"

Arb_Re_Interarrival_Time - Set the Inter-arrival time for generating Arbitration Request message. Note that, this parameter will not be utilized if the Node Type is set to "Root"

Transfer_Type – Transfer_Type must be set to either ISO or ASY, ISO defines Isochronous traffic while ASY defines Asynchronous Traffic

Task_Size_Bytes – Size of the Transmission Data

Enable_Debug – This parameter enables or disables capturing debug information and network Statistics

Number_Of_Stats_Samples – Parameter to select number of Statistics generated.

Sim_Time – Simulation Time. Must be mapped to top level digital simulator

10.7 FireWire Link

FireWire Link is a simple delay block that computes link delay based on the link type and distance between one node to another. FireWire is available in wireless, fiber optic, and coaxial versions using the isochronous protocols.

10.7.1 Block Parameters

Link_Type = "Optical"

Link_Distance = 3.0 /* in Meters */

10.8 FireWire Config

FireWire Config block is responsible for maintaining routing information, Maximum Bytes per cycle for a selected Network Speed.

Example Routing Table is shown below

ID	Source_Node	Destination_Node	Distance	Speed_Mbps	
1	Leaf_1	Branch_3	10.0	400.0	;
2	Branch_3	Root_5	10.0	400.0	;
3	Leaf_2	Branch_4	10.0	400.0	;
4	Branch_4	Root_5	10.0	400.0	;
5	Root_5	Branch_3	10.0	400.0	;
6	Branch_3	Leaf_1	10.0	400.0	;
7	Root_5	Branch_4	10.0	400.0	;
8	Branch_4	Leaf_2	10.0	400.0	;

10.8.1 Block Parameters

RT_Table – Routing Table defining the connectivity between nodes.

BW_Table – Do not edit this table. This table has details on Bandwidth Allocation Units for different network speeds (S100 to S3200)

ISO_BW_Percent – This parameter defines bandwidth allocated for Isochronous transfer. By default this parameter is set to 80%.

Network_Speed – Speed of the network

10.9 Reports

FireWire Library blocks are preconfigured to generate various reports and debug information. Configuring Enable_Debug parameter to “true” in Root node generates the reports and statistics. Activity graph is generated to debug network behavior and it captures details such as Arbitration Request arrival time, Arbitration Grant time and arrival data packet at Root Node for both Isochronous and Asynchronous transfers. Textual report on Network Statistics helps designer to understand the bandwidth for each link and source to destination nodes. Debug messages are generated in textual format and is saved in the same directory where the model is located.

10.10 Example

Multiple demonstration system models are provided with documentation to guide a user to adopt VisualSim FireWire Library for conducting early system exploration of FireWire based system. Purpose of the following tutorial is to introduce VisualSim FireWire libraries and understanding basic rules that needs to be followed during model construction.

Block diagram of a simple FireWire based system with single FireWire Root Node a Leaf Node is as below



Figure 1.0: System Block Diagram

Here we have a single Leaf node generating Isochronous or Asynchronous transactions and it can be considered as a camera or an audio source connected to a Computing resource or a Root node. Root Node generates a special packet called “Clock Start” every 125.0us. Leaf node can request for Arbiter control by sending a Arbiter Request message across the network to Root node. As we have only one device connected to Root Node, Leaf Node will always win the arbitration, else arbitration will be granted based on first come first serve order. Once the Arbitration is granted for a Leaf node, leaf node can send a data packet per cycle, if the

bandwidth is available multiple packets can be transmitted. Modeling a FireWire System as shown in figure 1.0 using VisualSim requires FireWire Library package. FireWire Library package includes FireWire Node, to model Root/Branch/Leaf nodes; FireWire Link, to model connectivity medium between two nodes; FireWire Config, to define the network setup and Bandwidth allocation. VisualSim model of the proposed system is shown below

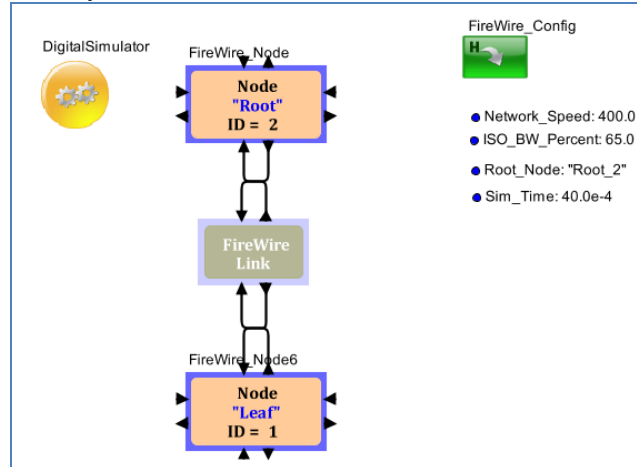


Figure 2.0 VisualSim Model

10.10.1 Basic Rules

1. While constructing the Simulation model of a FireWire based system, we assume that the topology has been identified and self-identification process is complete. As this doesn't impact on total System performance, this process is left out.
2. Always Start providing Unique identifier for nodes from Leaf Nodes and move up the ladder. Always Leaf Nodes will have the lowest identification numbers and Root nodes will have highest identification number.
3. User must determine if the transactions are Isochronous or Asynchronous based on the type of the device modeled using FireWire Node block.
4. Parameter called Root_Node with the name of Root node must be instantiated into the Block Diagram Editor.

10.10.2 Construction Steps

To Construct a FireWire network model, there are four steps

1. Instantiate FireWire Node, Link and FireWire Config blocks
2. Configuration of FireWire Node and FireWire Config blocks based on design requirements
3. Connect Nodes based on the Network Topology
4. Run Simulation and Analyze reports

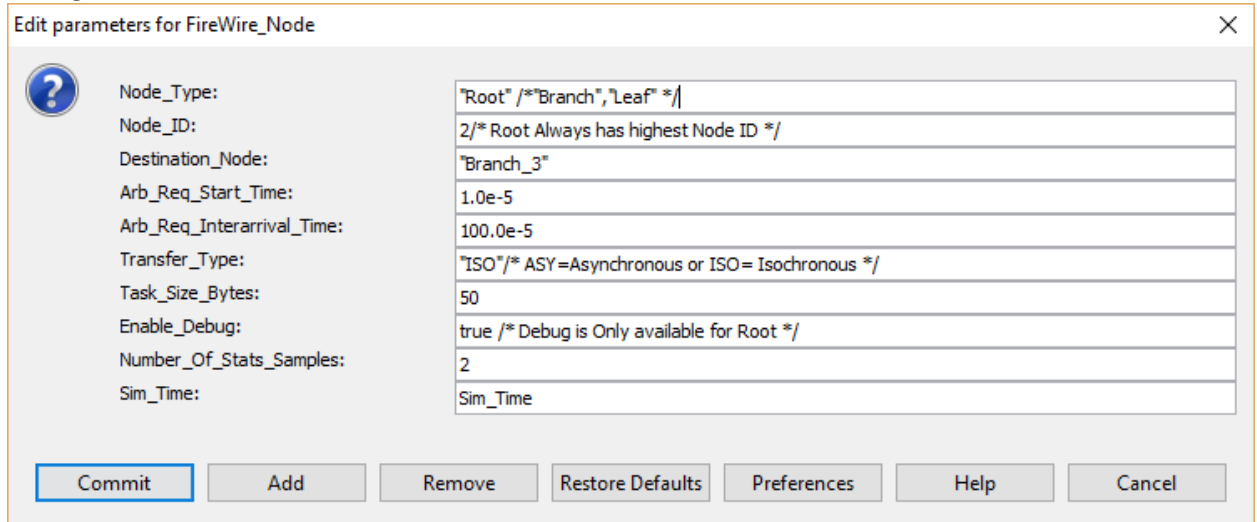
10.10.2.1 Step 1:

1. Open a New Block Diagram Editor from File → New → Block Diagram Editor
2. Instantiate Digital Simulator from the Library Menu Model Setup → Digital Simulator
3. Instantiate Parameter from the Library menu Model Setup → Parameter. Rename the Parameter as "Sim_Time" and assign the value as 40.0e-4.
4. Double Click on Digital Simulator and enter Sim_Time for stopTime Parameter

10.10.2.2 Step 2:

1. Instantiate FireWire_Node from the Library pane Interfaces and Buses → FireWire → FireWire_Node
Updating the parameter Node_Type to Root transforms the FireWire Node block to behave as a root in the network. As this is a Root node, identifier for the node must be provided with highest identifier number. For "Root" Node, user can ignore parameters; Arb_Req_Start_Time, Destination_Node, Transfer_Type, Task_Size_Bytes and Arb_Req_Interarrival_Time.

Configure the Block as below

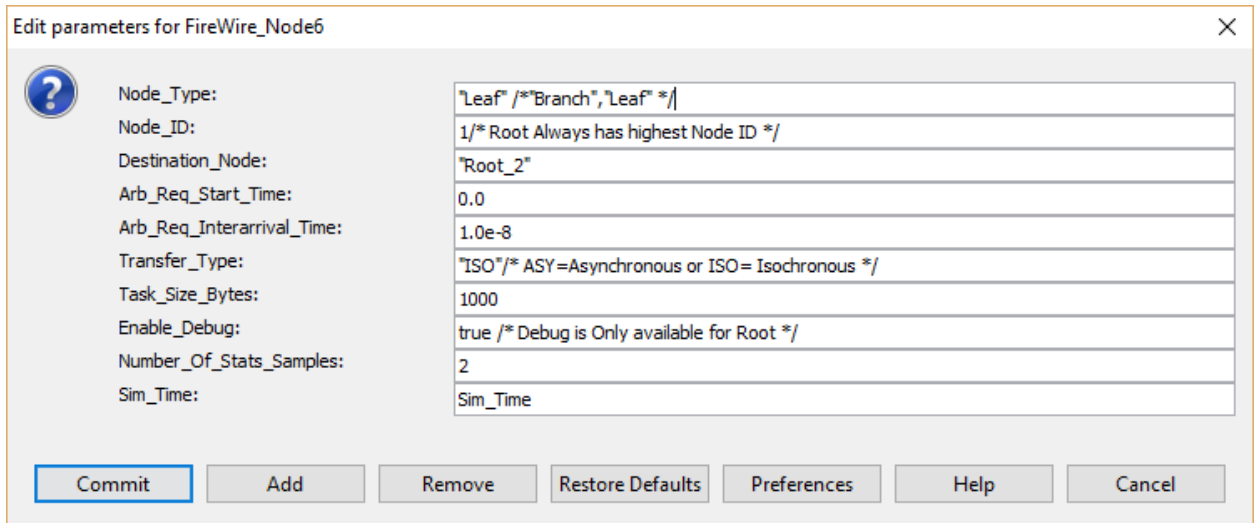


Parameter	Value
Node_Type:	"Root" /*Branch\", \"Leaf\" */
Node_ID:	2 /* Root Always has highest Node ID */
Destination_Node:	"Branch_3"
Arb_Req_Start_Time:	1.0e-5
Arb_Req_Interarrival_Time:	100.0e-5
Transfer_Type:	"ISO" /* ASY=Asynchronous or ISO= Isochronous */
Task_Size_Bytes:	50
Enable_Debug:	true /* Debug is Only available for Root */
Number_Of_Stats_Samples:	2
Sim_Time:	Sim_Time

2. Instantiate one more FireWire_Node from the Library pane Interfaces and Buses → FireWire → FireWire_Node

Updating the parameter Node_Type to Root transforms the FireWire Node block to behave as a root in the network. Node must be provided with a unique identifier and the root node must be provided with highest identifier number and Leaf Nodes will have must have least identifier number. FireWire Node is provided with a facility to generate Isochronous or Asynchronous transfers, Parameter "Transfer_Type" must be set to "ISO" for Isochronous and "ASY" for Asynchronous transfers. Parameter "Arb_Req_Start_Time" tells Leaf/Branch node to initiate Arbitration Request; Arb_Req_INterarrival_Time parameter allows user to set mean time to generate Arbitration Request. In this example as we have only 2 nodes, destination node for Leaf_1 is set to Root_2. Arbitration Request message will be generated every 100 us and the packet size of 1000 bytes is sent as an isochronous transfer.

Configure the Block as below



3. Instantiate FireWire_Link Node from the Library pane Interfaces and Buses → FireWire → FireWire_Link

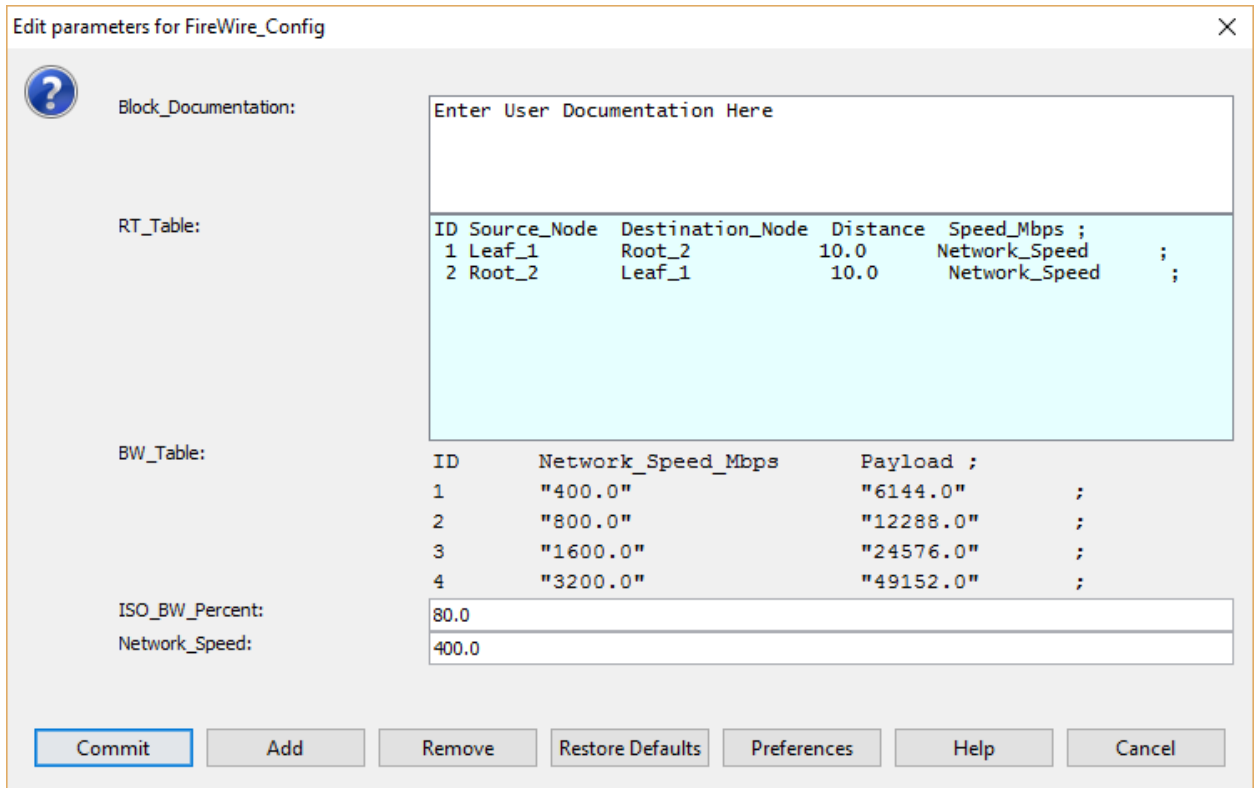
FireWire Link is a simple delay block that computes link delay based on the link type and distance between one node to another. FireWire is available in wireless, fiber optic, and coaxial versions using the isochronous protocols.

4. Instantiate FireWire_Config Node from the Library pane Interfaces and Buses → FireWire → FireWire_Config

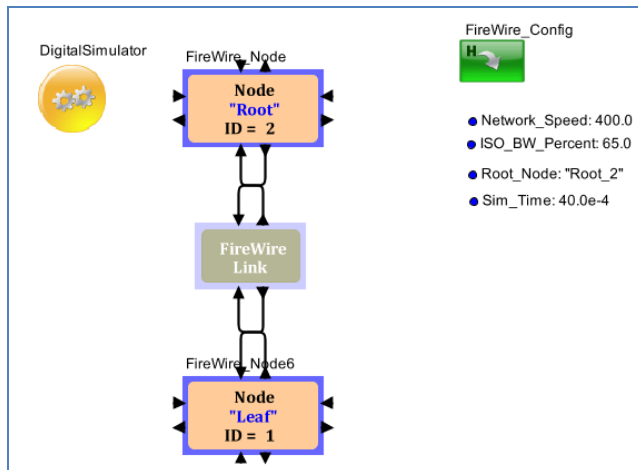
FireWire Config block is responsible for maintaining routing information, Maximum Bytes per cycle for a selected Network Speed.

As we have just two nodes, routing table will have connection defining Leaf_1 to Root_2 and Root_2 to Leaf_1. Distance is considered as 10 meters, user can modify this parameter. The Link Leaf_1 to Root_2 is running at 400 Mbps and is mapped to the block Parameter. User can modify the network speed and ISO_BW_Percent during explorations.

Configure the block parameters as below



5. Connect Nodes as shown in figure below

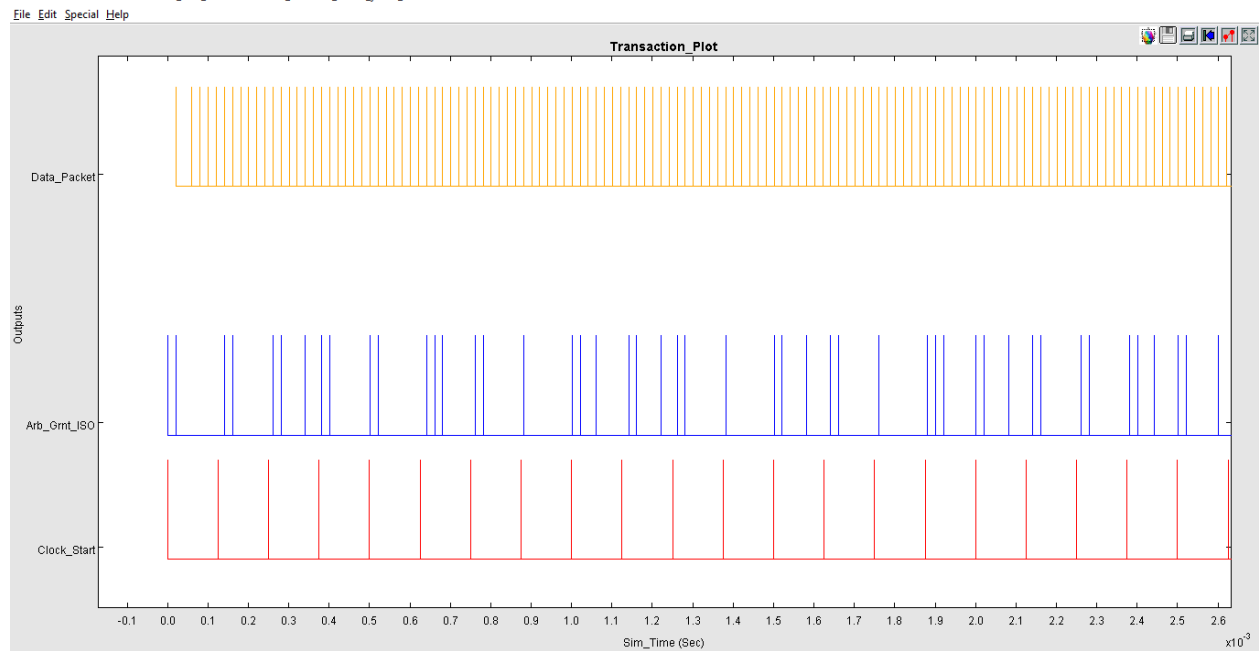


10.10.2.3 Reports

FireWire Library blocks are preconfigured to generate various reports and debug information. Configuring Enable_Debug parameter to "true" in Root node generates the reports and statistics. Activity graph is generated to debug network behavior and it captures details such as Arbitration Request arrival time, Arbitration Grant time and arrival data packet at Root Node for both Isochronous and Asynchronous transfers.

Textual report on Network Statistics helps designer to understand the bandwidth for each link and source to destination nodes. Debug messages are generated in textual format and is saved in the same directory where the model is located.

Activity Plot



Activity plot provide time stamp details on when a Arbitration is granted for Isochronous transfers or Asynchronous transfers, time stamp for clock start packet and arrival of data packet at Root Node. This plot is useful to monitor the network behavior.

Network Statistics

Network Statistics shows that each link connected to a node can capture the Mbps and min, mean, stdev, max of the link utilization. Network Statistics shown below tells us that the link Leaf_1 to Root_2 is having a bandwidth of 384.191 Mbps and the mean Bytes transferred across this link is 873 bytes while the maximum Bytes transferred across the link is 1000 Bytes. This also shows that the Link utilization is of 96.047%. This tells us the if the designer has to consider adding additional links to the network, care must be taken to make sure that the packets drop is avoided.


```

VisualSim Architect - Network Stats

DISPLAY AT TIME          ----- 1.0000000010 ms -----
{"Source", "Destination", "Mbps", "Min_Bytes", "Mean_Bytes", "Max_Bytes", "Util_Pct"}
, {"Leaf_1", "Root_2", "384.191", "0.0", "873.163", "1000.000", "96.047"}
, {"Root_2", "Leaf_1", "376.895", "4.0", "628.160", "1000.000", "94.223"}
}

DISPLAY AT TIME          ----- 2.0000000010 ms -----
{"Source", "Destination", "Mbps", "Min_Bytes", "Mean_Bytes", "Max_Bytes", "Util_Pct"}
, {"Leaf_1", "Root_2", "392.191", "0.0", "883.315", "1000.000", "98.047"}
, {"Root_2", "Leaf_1", "388.895", "4.0", "635.450", "1000.000", "97.223"}
}

DISPLAY AT TIME          ----- 3.0000000010 ms -----
{"Source", "Destination", "Mbps", "Min_Bytes", "Mean_Bytes", "Max_Bytes", "Util_Pct"}
, {"Leaf_1", "Root_2", "394.858", "0.0", "886.658", "1000.000", "98.714"}
, {"Root_2", "Leaf_1", "392.906", "4.0", "635.086", "1000.000", "98.226"}
}

```

Debug Statistics

Debug Statistics provides information on when a Clock Start signal is sent to a node, Node that received Arbitration Grant message, details on active channel, Available Bandwidth for Isochronous and Asynchronous Transfers. These debug messages helps designers to understand System behavior and make sure that the activity across the network is free from errors.

10.11 Understanding Common Errors

1. VisualSim.kernel.util.IllegalActionException:
 Problem performing RegEx Script Line (27) Result: {"none"}
Solution: This error appears when the parameter Network_Speed is set to speed apart from 100, 200, 400, 800, 1600 and 3200. Please make sure that parameter Network_Speed is provided with one of the above six values.
2. VisualSim.kernel.util.IllegalActionException:
 Problem performing RegEx Script Line (11) Result: {"Root_2"}
Solution: This error tells that the parameter Root_Node = "Root_2" (or the Name of the Root node in your case) is missing. Please add a parameter called Root_Node and provide the value as "Root_2" (or the name of the Root Node in your system model)
3. VisualSim.kernel.util.IllegalActionException: Routing Table returned null for Next Hop, check Task_Destination (Root_3) field matches Dest Node Name
Solution: Name of the Root Node given for the Parameter Root_Node is wrong or the Root Node name entered in FireWire_Config block RT_Table is wrong. Please provide the correct Root Node name



Application and Algorithm Library

1 Networking

The Networking blocks perform a variety of networking functions, including simple model routing, complex model routing, OSI layer modeling, and channel related modeling. In addition, the Networking blocks inter-operate with the Scheduler Resource blocks, using a common data structure. This means processor oriented models can be combined with network related models without data structure translators, or additional model processing.

The **NODE**, **Routing_Table** Blocks constitute the basic networking building blocks that allow connected or **virtual connection** nodes within a network. This provides the user with either a more conventional network of ports, and nets connection **NODEs** in the network, or a **virtual connection** network of mapped **NODEs**. The **Routing_Table** Block provides a routing table, plus common routing parameters for a network of **NODEs**. The user can have a variety of small networks in the same model, just requiring separate **Routing_Table** for each network topology.

Examples: The blocks in this library can be used to quickly construct a network of nodes or a channel with a preset capacity. This can be an IT network or a network on a chip/board. The network models can be used to create a verification environment around architecture or can be used to evaluate a protocol design. It can also be interfaced with external tools such as Satellite Toolkit (STK) to create complex satellite networks. Using the Channel_* blocks, a wireless network could be constructed with full effects of the error recovery and retransmission, for example.

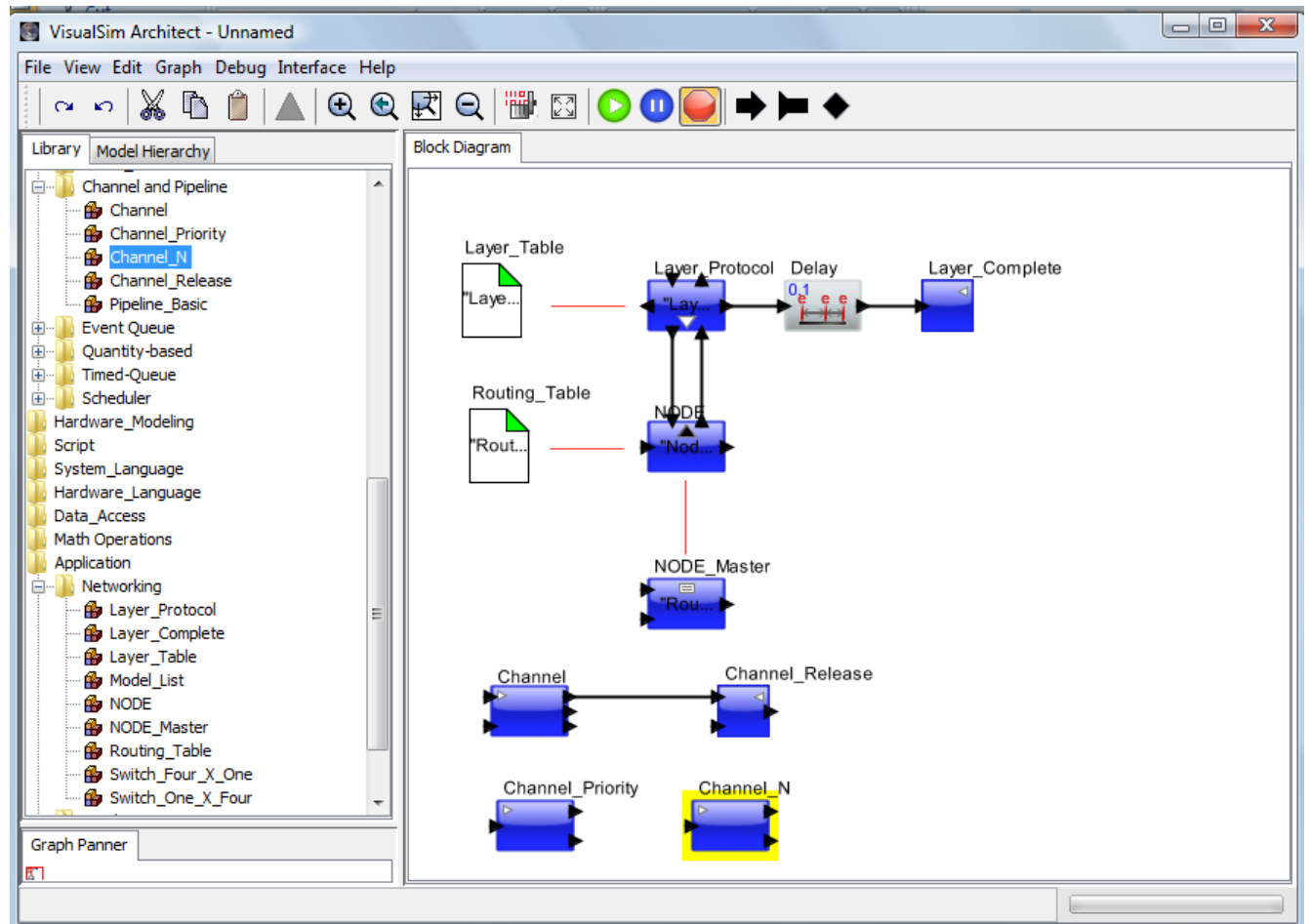


Figure 1-1 VisualSim Networking and Channel Blocks

The **Node** blocks define nodes in a user-defined topology. A number of **Layer_Protocol** blocks can be setup as parents of each node to indicate the number of layers required by a protocol stack. The basic parameters of each layer type is provided in the **Layer_Table** block. The **Layer_Table** and **Layer_Protocol** are mapped by using the Layer_Table_Name of the Layer_Protocol block. The ds_dn_output is used to connect to the lower layer and the ds_up_output is used to connected to the upper layer. The ds_dn_input is used to receive data from the upper layer and the ds_up_input is used to receive data from the lower layer. The output ports from the Layer_Protocol blocks can be connected to a FSM or other blocks to create define processing specific to the layer, such as MPLS, Ethernet etc. The routing of the data through this model is determined using the **Routing_Table** block. A single model can have multiple **Routing_Table** blocks but must be referenced in independent networks or inconsistency will occur. The **Node_Master** can be used to modify the operation of the network such as recomputing the routing table and adding/removing a link. The Layer_Complete block completes the Protocol Layer process by calling back the appropriate Layer_Protocol block signalling the completion of either an up or down protocol layer process. The use of this block assumes 'Layer_Table' Block is set to 'Layer_Configuration' menu attribute and 'External_Delay' for this block to be necessary. The Layers handle fragmentation and assembly using the standard "session" convention of TCP/IP. There is a fragment

size for the upward and downward progression of the packets. Each layer has a frame size it will propagate. When enough packets have arrived for that packet size, then the incoming packet(s) are sent on to the next layer. The model does not wait for the entire message, similar to the OSI stack operation. The block delays the arriving fragments from the same session until this fragment size is reached and then transmitted. If the incoming size is larger than the fragment number, then incoming packet are broken down into multiple packets. All the data from a session are considered in the fragmenting and assembly process. A session is identified using the Task_Number as the unique identifier.

List of Blocks

1. **Layer_Protocol** accepts data structures from a NODE block or intermediate Protocol Layer.
2. **Layer_Complete** block completes the Protocol Layer process by calling back the appropriate Layer_Protocol block signaling the completion of either an up or down protocol layer process. Assumes 'Layer_Table' Block set to 'Layer_Configuration' menu attribute and 'External_Delay' for this block to be necessary.
3. **Layer_Table** block provides for common parameters for an OSI modeling layer.
4. **Model_List** operates in the same way as the Model_List_DS. The Model_List can access the database used to store the routing information by the Routing_Table block of the Networking Library.
5. **Node** can model a network using connected or connectionless methods.
6. **Node_Master** can operate in a variety of modes: (1) Add Link, (2) Remove Link, and (3) Recompute the Routing Table.
7. **Routing_Table** operates in two modes: (1) Connected Routing Table Mode, and (2) Wireless Routing Table Mode

2 Wireless and Sensor Network System

Introduction

VisualSim Wireless is a modeling and simulation framework for wireless and sensor networks that builds on and leverages VisualSim. Modeling of wireless networks require sophisticated modeling of communication channels, sensors, ad-hoc networking protocols, localization strategies, media access control protocols, energy consumption in sensor nodes, etc. This modeling framework is designed to support a component-based construction of such models. It supports actor-oriented definition of network nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems. The software architecture consists of a set of base classes for defining channels and sensor nodes, a library of subclasses that provide certain specific channel models and node models, and an extensible visualization framework. Custom nodes can be defined by sub-classing the base classes and defining the behavior in Java or by creating composite models using any of several VisualSim modeling libraries. Custom channels can be defined by sub-classing the WirelessChannel base class and by attaching functionality defined in VisualSim models. It is intended to build models that include sophisticated elements from several aspects.

In this document, we begin by explaining the basic components in this framework: the simulator, the channel model and the sensor node model, and how to build sensor network models graphically. This document provides a tutorial that will enable the reader to construct elaborate sensor network models and to have confidence in the results of a simulation of those models.

The intended audience for this document is an engineer or researcher who is interested in wireless and sensor network systems and wishes to build models of such systems.

VisualSim Wireless is built on top of VisualSim, a framework supporting the construction of such domain-specific tools.

Installation and Quick Start

VisualSim Wireless libraries blocks are located in the Interfaces and buses->Wireless sensor Folder of the Library structure.

Modeling Wireless Networks

In this section, we explain how to read, construct and execute models of wireless sensor networks. We begin by examining a demonstration system that is accessible from the "Models in BDE" page, the wireless sound detection model. These demonstration systems are meant to illustrate capabilities, not necessarily to serve as accurate or useful models of physical systems.

Running a Pre-Built Model

The wireless sound detection model can be accessed by clicking on the link in the welcome window (figure 1), which results in the window shown in Figure 2. This is a highly simplified (even naïve) model of a sound localization system that uses a field of sensor nodes that detect a sound and report by radio to a hub that triangulates the location of the sound. Figure 2 shows the elements of the model, which include a Wireless_Simulator, which defines this as a wireless model, two channel models (a radio channel model and a sound channel model), a number of annotations (text explaining the model) and actors in the model. Each of these components plays a role in the model. The director mediates execution of the model. The channel models handle communication between the actors. The actors send and receive signals via the channel.

The model is executable. Clicking on the red triangle in the toolbar results in the SoundSource actor (represented by concentric transparent circles) beginning to move in a circular pattern, as indicated by the

blue arrow in figure 3. The SoundSource actor emits events via the SoundChannel channel model. These events propagate with a time delay dependent on distance to the blue circular nodes. When these nodes detect the sound, they emit a radio signal via the RadioChannel model and turn their icons red to indicate visually that they have done so. The radio signals include a time stamp of the detected sound event. The Triangulator actor in the center (shown with a green icon) receives these

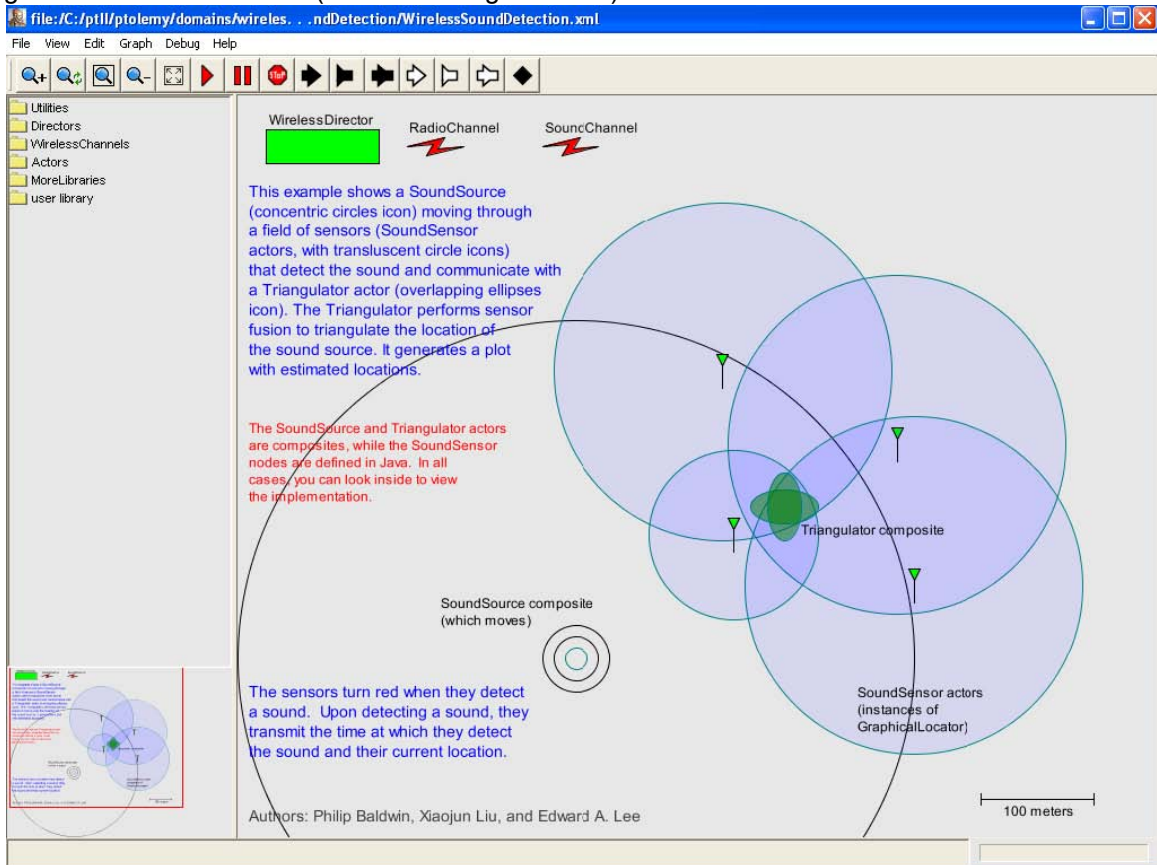


FIGURE 2. The VisualSim Wireless representation of a wireless sound detection model.

radio signals (if it is in range of the transmitter), and uses the time stamps to estimate the position of the sound source. It then plots that position, resulting in the plot shown in figure 3.

Changing Parameters

The model has parameters that you can experiment with. The parameters of two components, SoundSource and SoundChannel, are shown in figure 4. To obtain these parameter screens, you can double click on the actor, or right click and select "Configure." The SoundSource has a single parameter, called soundRange. If you change the value from 300 (meters) to, say, 500, then the circular icon for the actor increases in size, and re-running the model results in more of the trajectory of the sound source being triangulated. In the SoundChannel parameters, you could set a non-zero value for the lossProbability, in which case only some of the sound events will be detected. Setting the seed to a non-zero value results in repeatable experiments, meaning that each execution will yield the same sequence of random numbers (the type is a long, so the value should be an integer followed by the letter "L"). Leaving the seed at the default "0L" yields a new experiment on each run.

Structure of a Pre-Built Model

Let us examine how the model in figure 2 is constructed.

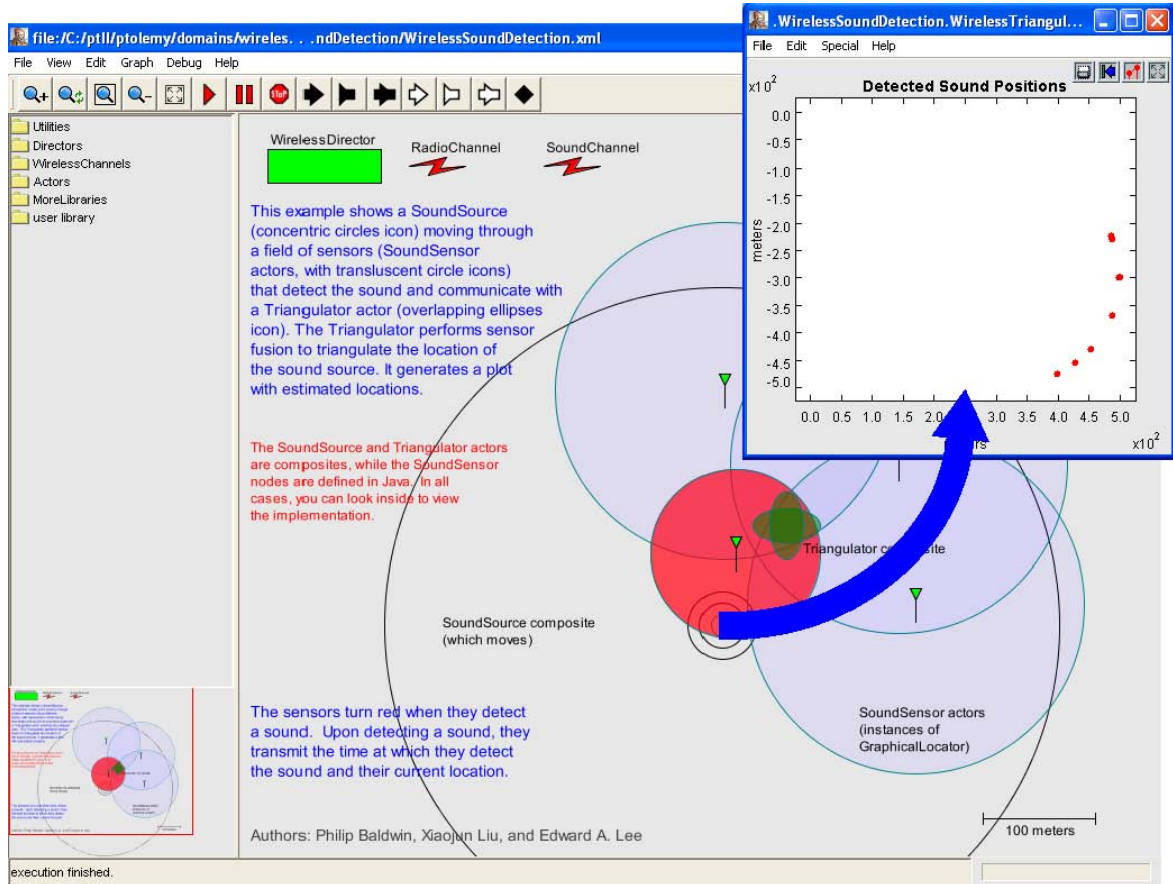


FIGURE 3. Animation as the model executes. The SoundSource actor moves in a circle through a field of Sound-Sensor actors. When these actors detect a sound, they transmit a radio signal to a Triangulator node, which estimates and plots (at the upper right) the position of the sound source.

Visual Representations (Icons)

Consider first the SoundSource actor. First, consider how its visual representation (its “icon”) changed when we changed the soundRange parameter. The definition of the icon can be viewed (and edited) by right clicking on the icon and selecting “Edit Custom Icon.” Note that to select this actor, you must place the mouse over one of the concentric circle outlines. The resulting window is shown in figure 5. Note that only the center portion of the icon is visible. Click on “Zoom Fit” in the toolbar (as shown in figure 5) to get the full image, as shown in figure 6. The navigation window at the lower left can be used to move the view around (to “pan” the view). The library at the left can be used to add items to the icon.

Consider the outer circle, which changed size when we changed the soundRange parameter. Double clicking on it (or right clicking and selecting Configure) reveals the parameter window in figure 7. Notice that the width and height parameters are given by expressions with values “soundRange*2”. The expression language that can be used here is rich, and will be described below. For now, it is sufficient to realize that arithmetic expressions that reference parameters of the actor or of the model can be used to extensively customize the visual representation of an actor, making it depend on parameter values. For more information on Expression, refer to RegEx section in Chapter 2.

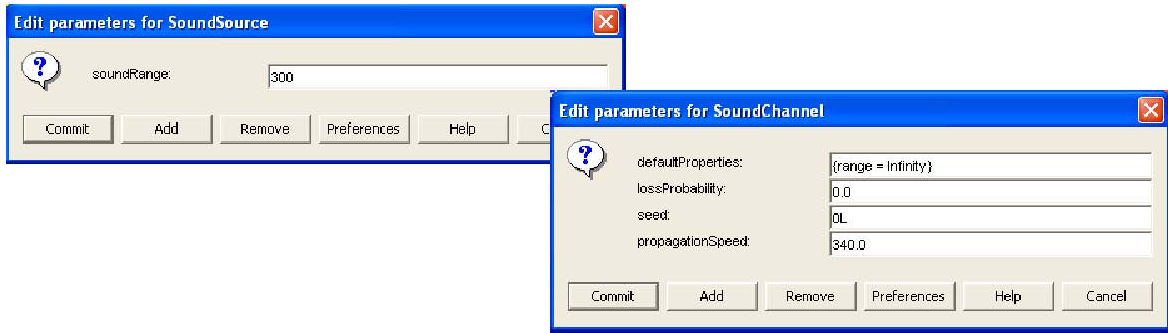


FIGURE 4. Parameters of the SoundSource actor (left) and SoundChannel channel model (right).

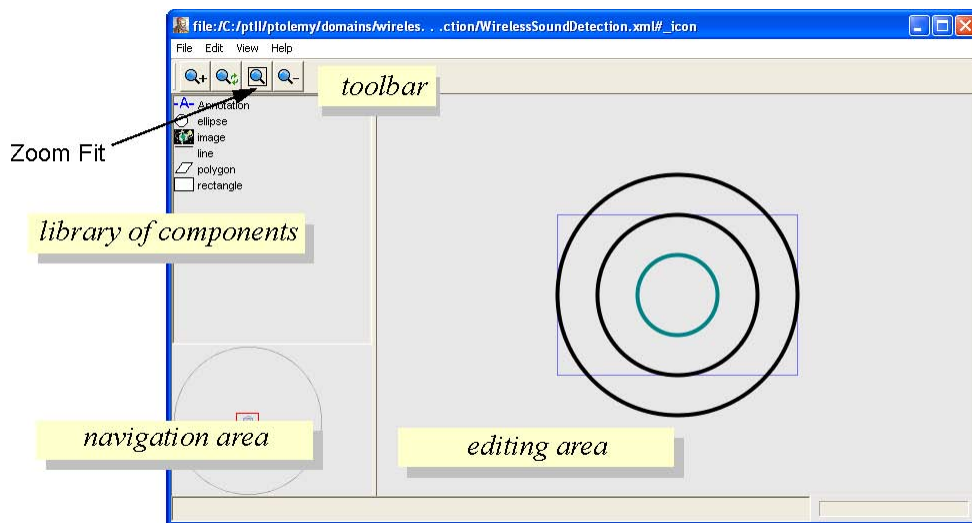


FIGURE 5. View resulting from selecting “Edit Custom Icon” after right clicking on the SoundSource in figure 2.

For example, we could fill the outer circle with a translucent color where the degree of translucency depends on the soundRange parameter, as shown in figure 8. In that figure, the color selector (shown at the right) was used to select a red color, and the alpha value of the color, which is the fourth element of the array defining the color, was manually set to “soundRange/1000.0”. The result is shown in figure 9.

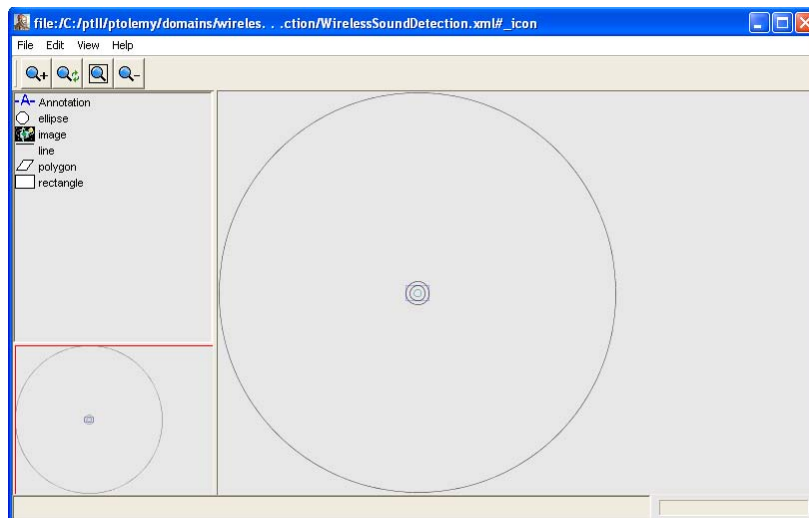


FIGURE 6. View resulting from clicking Zoom Fit in the toolbar of figure 5.

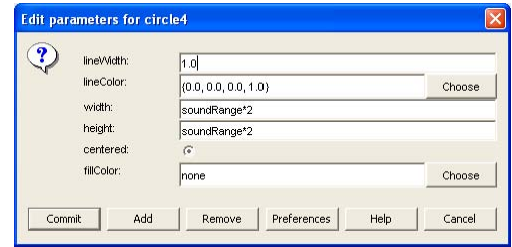


FIGURE 7. Parameters of the outer circle of the SoundSource actor icon in figure 5.

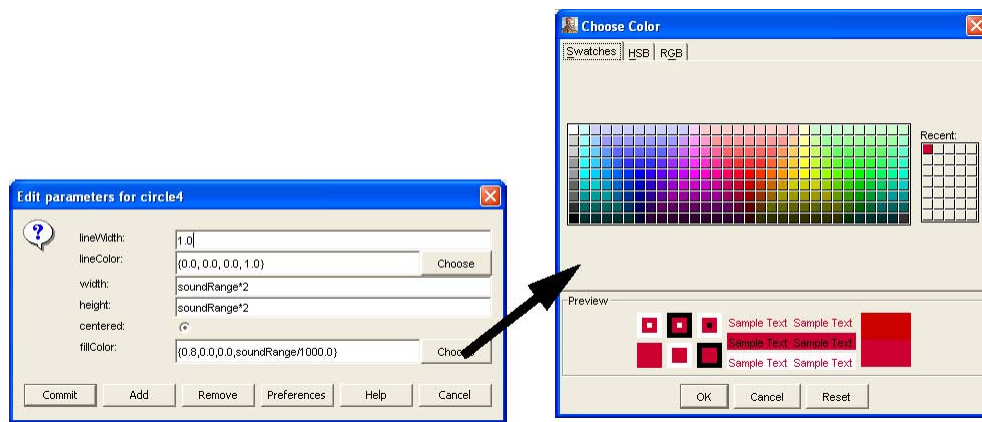


FIGURE 8. Setting the fill color of the outer circle of SoundRange to depend on its soundRange parameter.

Feel free to experiment with this icon by moving components, changing their colors, or adding new components. You can add GIF or JPEG images defined in a file using the Image component, and you can add lines, circles, polygons, or rectangles.

Note that as of this writing, the icon editor is fairly primitive. The interactors for the various shapes are not customized, so defining a shape can be a tedious matter of defining the vertex points. Also, the order in which items in the icon are drawn is the order in which they are created. Thus, the only mechanism currently to put an object in the foreground is to select it, delete it, and then re-add it. We expect this editor to improve over time.

Channels

The model shown in figure 2 has two channel models, shown in figure 10 along with their parameters. You can see that the only difference between these two channels (besides their names) is the value of the propagationSpeed parameter. For the RadioChannel, it is set to "Infinity," whereas for the SoundChannel, it is set to "340.0" (meters/second).

Note that both channels have a parameter called defaultProperties with value "{range=Infinity}." This expression defines a record with one field named "range" with value "Infinity." The fields of the defaultProperties parameter of a channel define the ways in which a particular transmission can be individually customized. In this case, a particular transmission through either channel can optionally specify a range. If it is not specified, then the default is used, which is Infinity, indicating that there is no range limitation. A transmission will succeed in reaching the receiver no matter how far away the receiver is.

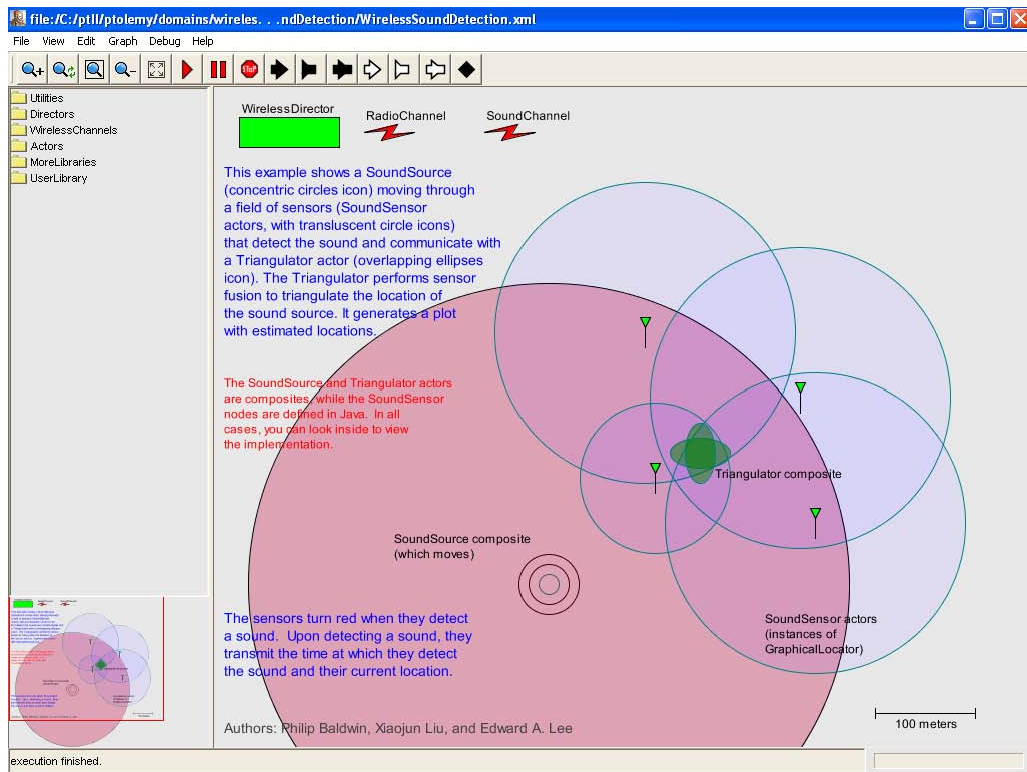


FIGURE 9. Result of changing the color of the outer circle of SoundRange as shown in figure 8.

Wireless Hierarchical Blocks

We have seen how to customize the visual representation of an actor. How can we define its behavior? The SoundSource actor in figure 2 is actually a hierarchical block whose behavior is defined by a VisualSim model. To find this definition, simply right click on the actor and select Look Inside. The inside model is shown in figure 11.

The SoundSource composite shown in figure 11 has a Digital Simulator, which defines this model as a VisualSim discrete event model. Digital models work well with wireless models, so it is common to see Digital models used to define wireless nodes. The soundRange parameter is shown next to the Digital Simulator with its default value, 300. The model itself consists of two parts, an upper part that sends a sound event, and a lower part that moves the icon.

Consider first the upper part. It has a Clock and a port named “soundPort,” as shown in figure 12. The parameters of both the Clock and the port are obtained by double clicking on them (or right clicking and selecting Configure), and are also shown in the figure. Notice that the period of the Clock is set to 2.0, and the values are set to {1}, an array with one element, the integer 1. This indicates that the clock should produce a sound every two seconds. The value produced is simply the integer 1, which has no particular meaning. Any value would have the same effect.

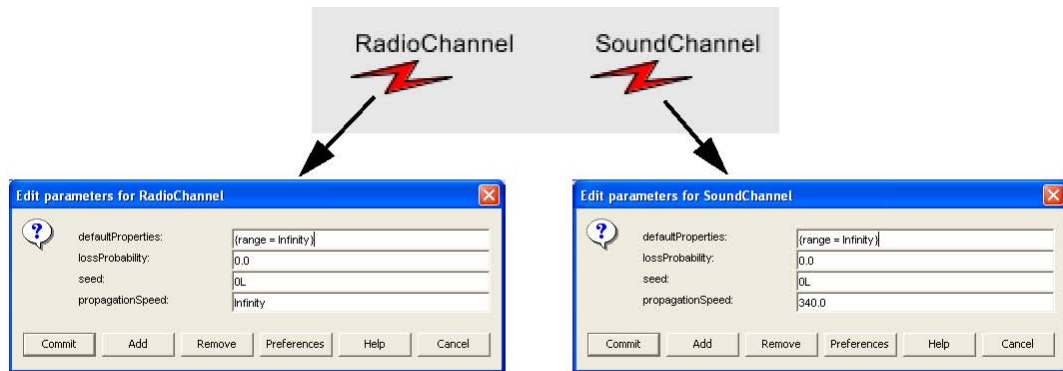
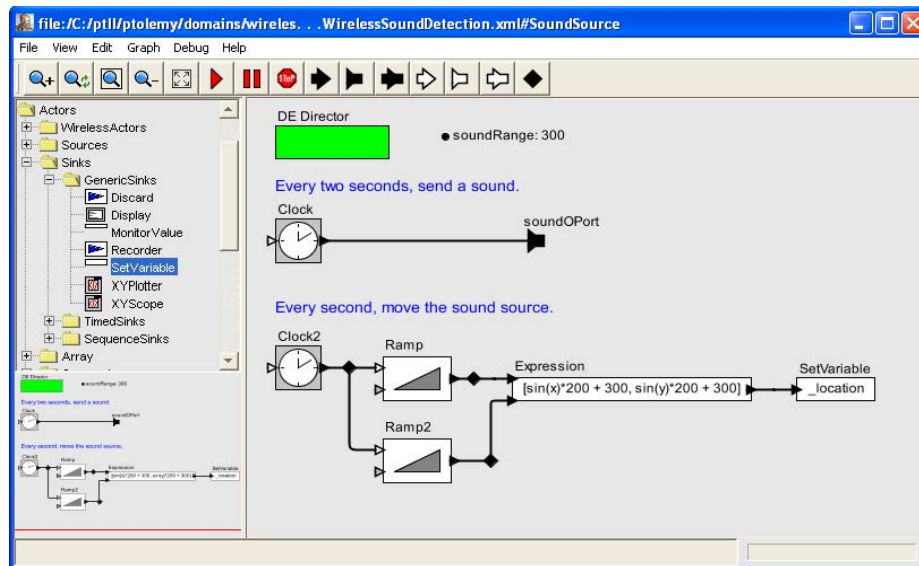


FIGURE 10. The channels of figure 2 and their parameters.



The soundPort component also has parameters, as shown in figure 12. The outsideChannel parameter is a string-valued parameter with value “SoundChannel.” This is the name of the channel that this port will use for transmission, and must correspond with the name of the channel shown in figure 10. The outsideTransmitProperties parameter has value “{range=soundRange}” which is a record with one field named “range” with value given by the expression “soundRange,” which simply obtains the value from the soundRange parameter of the hierarchical block. Notice that this will override the default value of Infinity given for this field in figure 10. Thus, the soundRange parameter controls not just the visual appearance of the icon, but also the range of transmission.

For the purposes of determining whether a receiver is in range, all of the demos included with VisualSim Wireless use the location of the icon as a (two dimensional) representation of the location of the node. The units are arbitrary, but in these models are taken to represent meters. A scale is shown at the lower right of figure 2, indicated by a line of length “100,” which represents 100 meters.

Although these demos all use two-dimensional locations, the underlying software infrastructure supports three dimensional locations. The visual editor, however, does not offer a mechanism for directly defining those locations, so for illustration purposes, the demos constrain themselves to two-dimensional locations.

Controlling the Execution

The Wireless_Simulator in figure 2 is the component that controls the execution of the model. As with most components, it too has parameters. Its parameters are shown in figure 13. Notice that the stop time is

set to “MaxDouble,” which is a very large number $1.7976931 \times 10^{308}$. This specifies that the model should run forever.

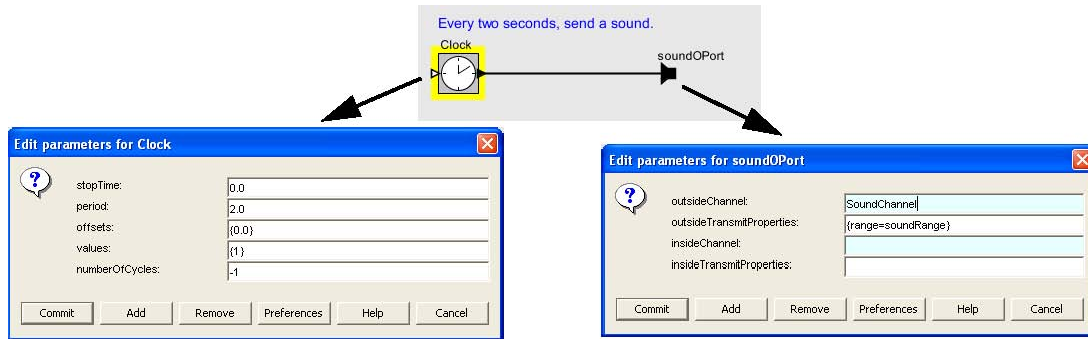


FIGURE 12. Portion of the composite in figure 11 that produces the sound event, with two parameter screens.



Notice also that the `synchronizeToRealTime` parameter of the director is checked. This means that when executing the model, the Clock actor that produces a sound every two seconds will not be allowed to produce events at a faster rate than that in real time even if the model can execute faster. This parameter is used to get realistic time scales when animating an execution. Usually, this parameter should be checked for animated models. The other director parameters have to do with tuning the performance of the discrete-event simulator. They are beyond the scope of this document.

Building a New Model

We now proceed to build a new wireless network model from scratch. In any VisualSim Wireless window, select `File→New→Block Diagram Editor`. This results in a window like that shown in figure 14. Drag in the `Wireless_Simulator` from the `Full Library →Application→Wireless` folder. Drag in a `PowerLossChannel` from the `WirelessChannels` library at the left, as shown in figure 15.

Notice the parameters of this channel, which are also shown in figure 15. Notice that the default-Properties parameter contains a record with two fields, `{range = Infinity, power = Infinity}`. This channel can be used to model variations in transmit power and also power loss as a function of distance. We will construct a simple model that achieves communication if the receiver gets enough power, and does not achieve communication otherwise.

Documentation for the `PowerLossChannel` actor (and any other actor) can be obtained by right clicking on the actor and selecting `Get Documentation`. In this example, we get the screen shown in figure 16, which shows the documentation of this channel. The top of this display shows the inheritance chain for the actor, which indicates that this actor extends `LimitedRangeChannel`, which extends `DelayChannel`, which extends `ErasurChannel`, which extends `AtomicWirelessChannel`. Each of these channels adds a small amount of functionality, and source code for each one is provided as an illustration of how to define channel models. You can view the source code by right clicking and selecting `Open Block`, which results in the screen shown in figure 17. In the case of both the source code and the documentation, you have to scroll down some to get to the interesting part. For example, this documentation explains the `powerPropagationFactor` parameter as follows:

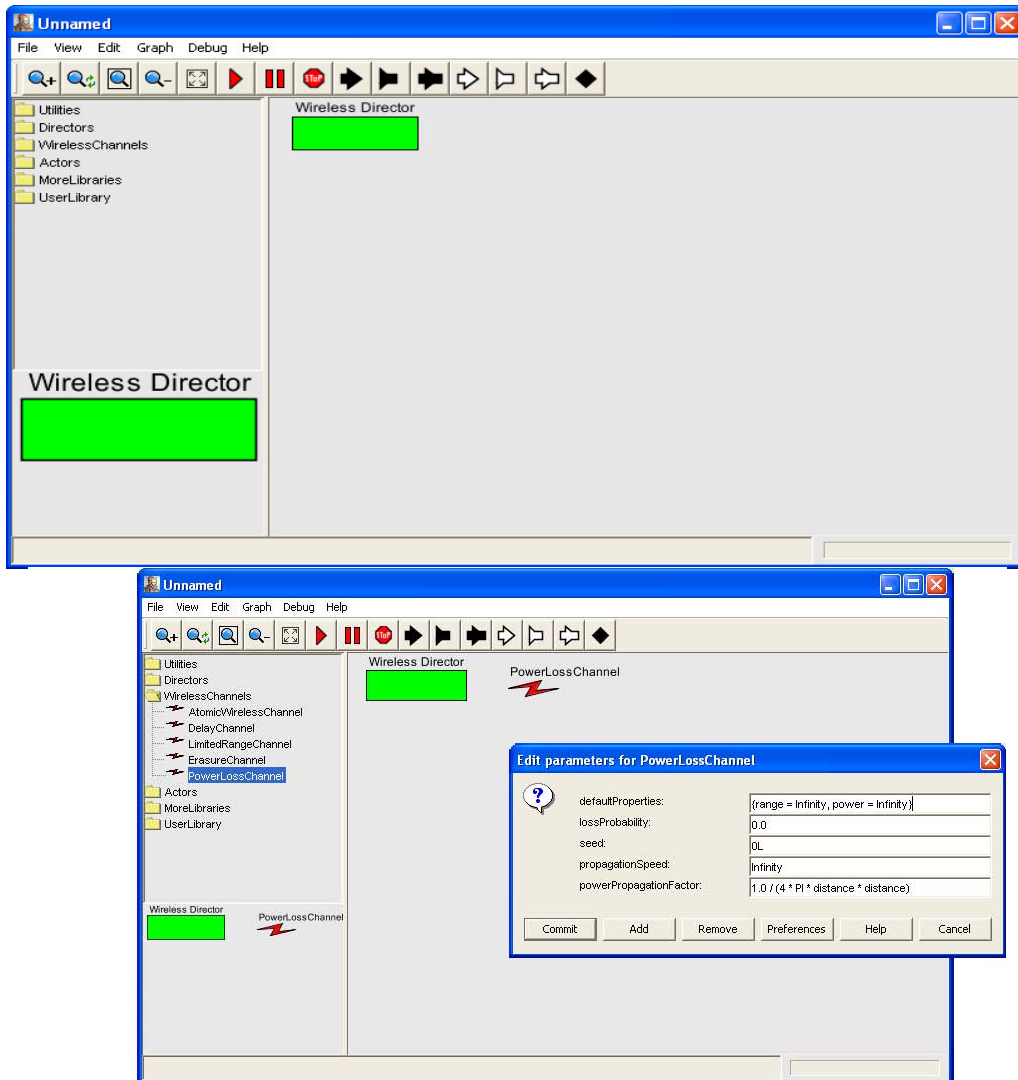
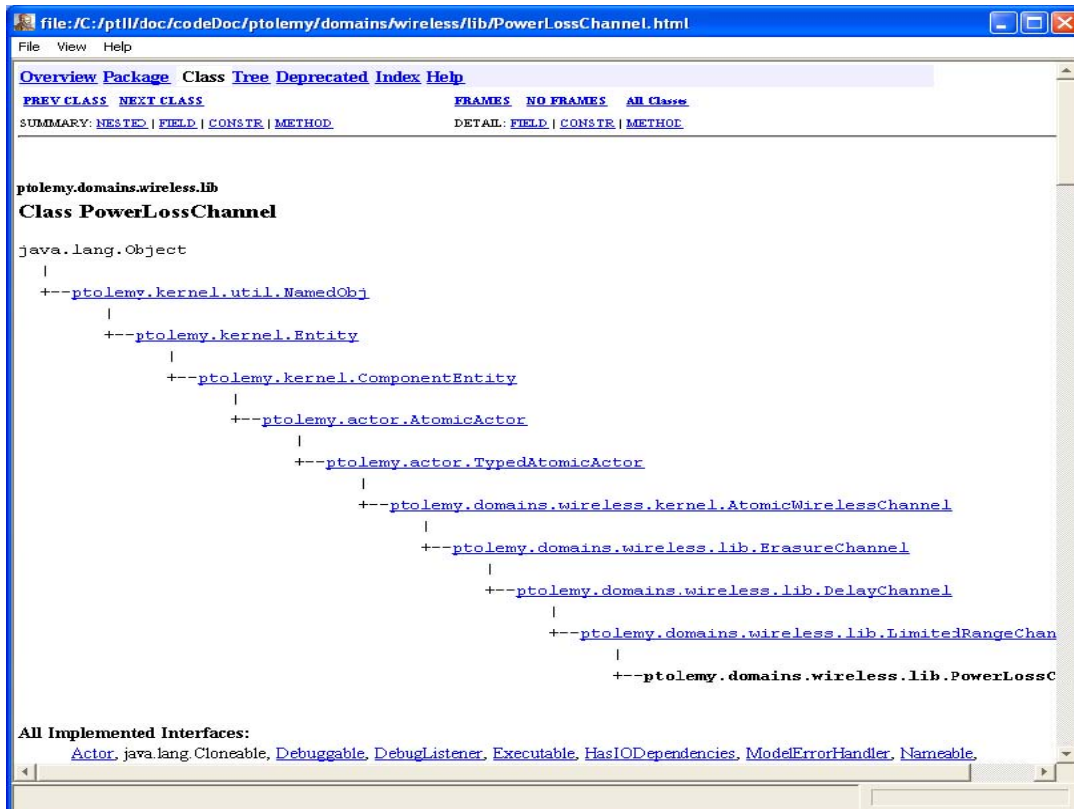


FIGURE 15. New model populated with a channel.



“The power propagation is given as an expression that is evaluated and then multiplied by the power field of the transmit properties before delivery to the receiver. For convenience, a variable named “distance” is available and equal to the distance between the transmitter and the receiver when the power propagation formula is evaluated. Thus, the expression can depend on this distance. The value of the power field should be interpreted as power at the transmitter but power density at the receiver. A receiver may multiply the power density with its efficiency and an area (typically the antenna area). A receiver can then use the resulting power to compare against a detectable threshold, or to determine signal-to-interference ratio, for example.

The default value of powerPropagationFactor is

$$1.0 / (4 * \text{PI} * \text{distance} * \text{distance}).$$

This assumes that the transmit power is uniformly distributed on a sphere of radius distance. The result of multiplying this by a transmit power is a power density (power per unit area). The receiver should multiply this power density by the area of the sensor it uses to capture the energy (such as antenna area) and also an efficiency factor which represents how effectively it capture the energy.


```

file:/C:/ptll/ptolemy/domains/Wireless/lib/PowerLossChannel.java
File Help
/* A channel with a distance-dependent power loss.

Copyright (c) 2004 The Regents of the University of California.
All rights reserved.
Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

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.

PT_COPYRIGHT_VERSION_2
COPYRIGHTENDKEY

*/

package ptolemy.domains.wireless.lib;

import ptolemy.data.DoubleToken;
import ptolemy.data.RecordToken;
import ptolemy.data.ScalarToken;
import ptolemy.data.Token;
import ptolemy.data.expr.Parameter;
import ptolemy.data.type.BaseType;
import ptolemy.data.type.RecordType;
import ptolemy.data.type.Type;
import ptolemy.domains.wireless.kernel.WirelessIOPort;
import ptolemy.kernel.CompositeEntity;

```

The power field of the transmit properties can be supplied by the transmitter as a record with a power field of type double. The default value provided by this channel is Infinity, which when multiplied by any positive constant will yield Infinity, which presumably will be above any threshold. Thus, the default behavior is to encounter no power loss and no limits to communication due to power.”

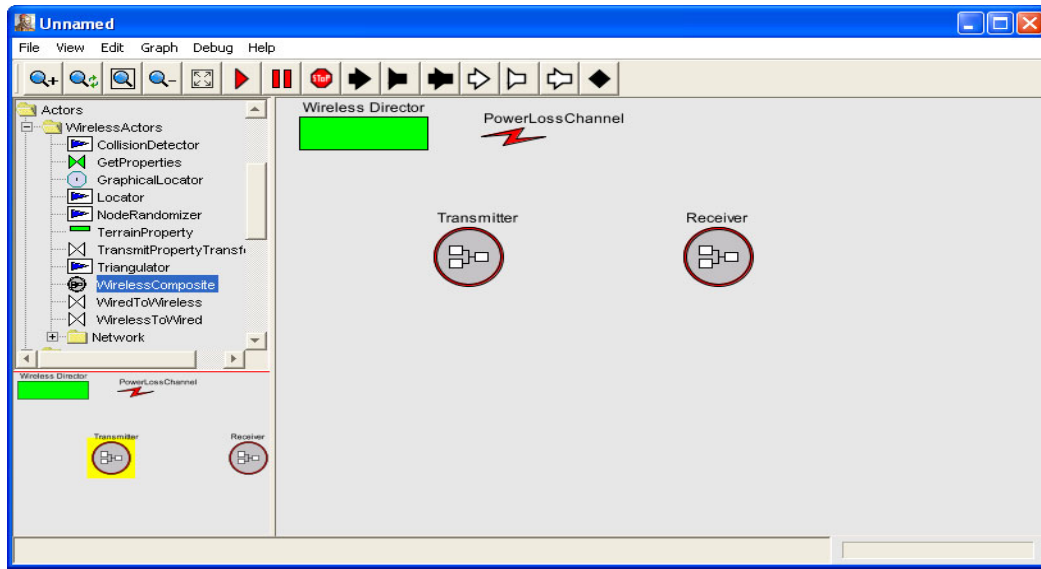
Hopefully, this makes it reasonably clear how to use these parameters. Let us build a model that uses them.

Begin by dragging in two instances of WirelessComposite from the Application→Wireless library at the left. Rename them Transmitter and Receiver by right clicking on them and selecting Customize Name, to get the result shown in figure 18. These components now need ports. To create these, right click on each icon and select Configure Ports. Click on the Add button and create an output port named output for the Transmitter, and an input port named input for the Receiver, as shown in figure

19. To specify that these ports use the PowerLossChannel, right click on each port and select Configure, and specify the outsideChannel to be “PowerLossChannel” (this must match exactly the name of the channel).

We start by populating the transmitter and receiver with simple models of the nodes. To do this, look inside the transmitter, which yields the window shown in figure 20. Note that the output port is (rather poorly) placed at the upper left. Move it to a more reasonable place, and connect to it an instance of the PoissonClock actor from the Sources→Clocks library to get the model shown in figure 21. To make a connection, either click and drag from the output port of the Poisson-Clock actor, or control-click and drag from output port of the Transmitter to the output port of the PoissonClock actor.

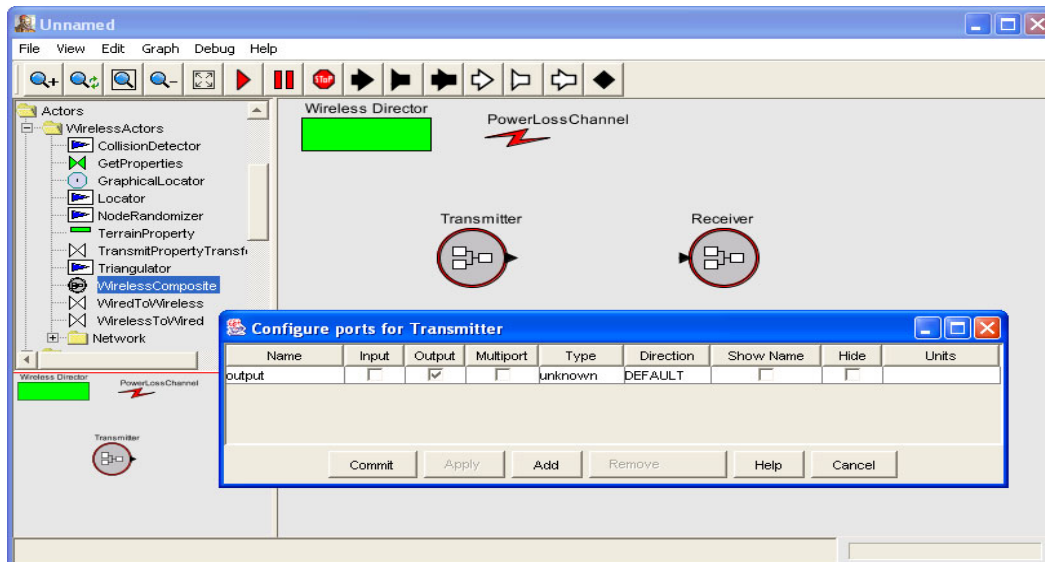
The PoissonClock actor will produce events at random times, where the time between events is obtained from an exponential random variable with mean given by the meanTime parameter of the PoissonClock. The default value is 1.0, which is fine for our purposes. If you return to the top-level window and double click on the Wireless_Simulator to set its synchronizeToRealTime parameter, then the transmitter will produce events at an average rate of one per second.

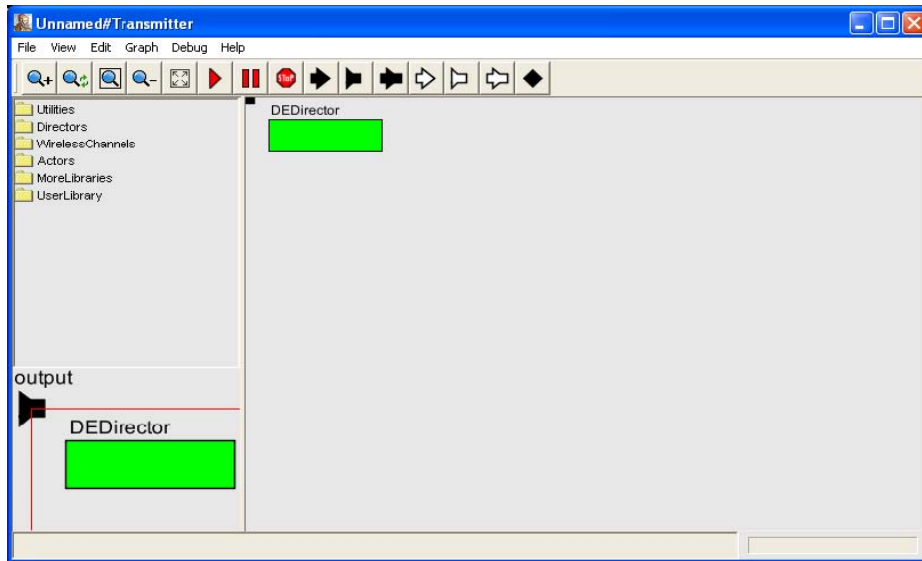


Look inside the Receiver actor and build the model shown in figure 22. The Ramp actor is found in the Actors library under Sources→Clocks, and the Display actor is found under Results→Plotter, as shown on the left in the figure. The model is now ready to execute. Clicking on the red triangle in the toolbar will result in the display shown in figure 23. The Ramp produces a count of arrivals. If you remembered to set the synchronizeToRealTime parameter of the Wireless_Simulator, then the count numbers will appear at random times with an average interval of one second.

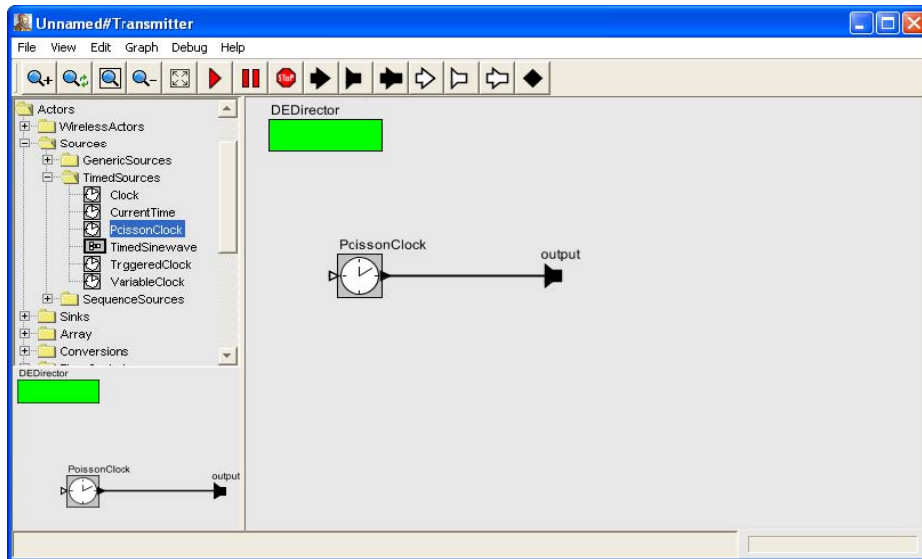
You may want to save your model using the File→Save menu command. Use the file extension .xml (or .moml) to ensure that VisualSim Wireless will recognize this as a model file. Notice that the title bar on the window now reflects the name of your model, which is the same as the name of the file.

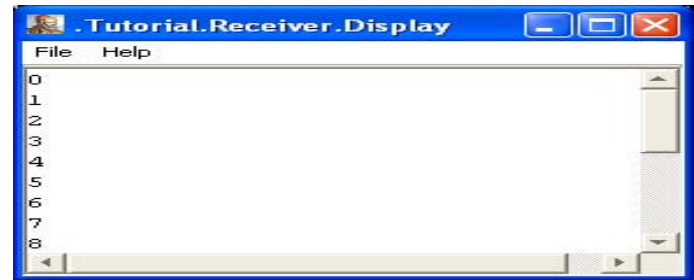
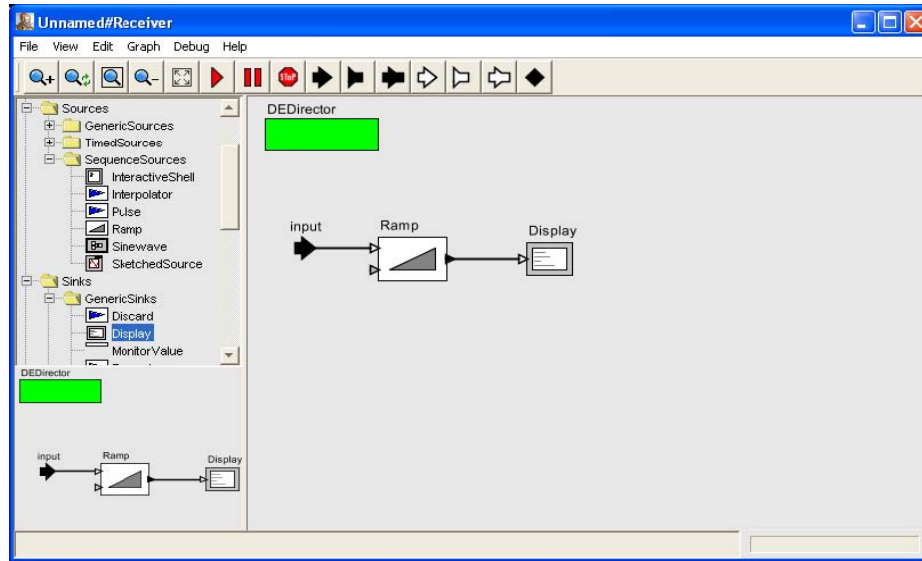
Let us modify this model so that the power loss of the channel as a function of distance is observed. To do this, find the GetProperties actor in the Full Library → Application→Wireless library, and replace





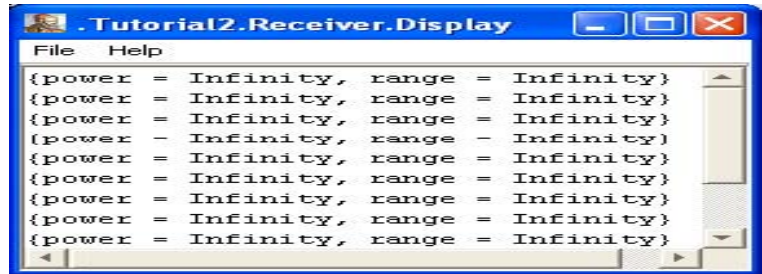
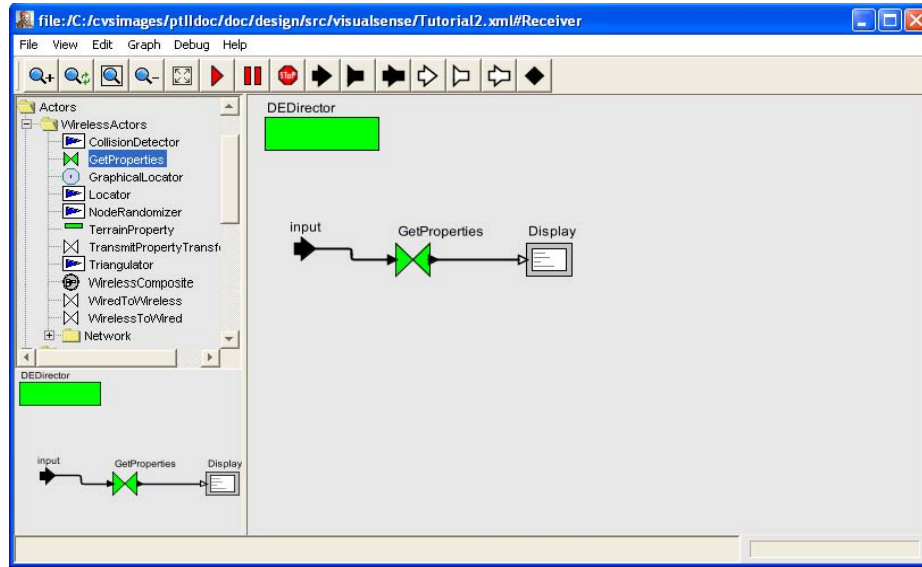
the Ramp block inside the Receiver as shown in figure 24. Running the model now results in the display shown in figure 25. Notice that the received power is always Infinity, which is not very useful.





Indeed, the Transmitter has not specified a transmit power, and the PowerLossChannel has a default power of Infinity, as shown in figure 15. The power loss introduced by the channel becomes irrelevant because in this model, the transmit power is infinite, which when multiplied by any non-zero loss, still yields infinite power.

To get a more reasonable model of power loss, set the transmit power by right clicking on the output port of the Transmitter and setting the outsideTransmitProperties parameter to "{power = 1.0}" as shown in figure 26. Re-running the model now results in a display like that shown in figure 27, where the variability in power level was obtained by moving the Receiver towards and over the Transmitter while the model was running.



Notice in figure 27 that one of the displays shows a received power of Infinity. This occurred when the Transmitter and Receiver were directly on top of one another. Recall from the documentation for PowerLossChannel that the value of the power field in the received properties is a power density (power per unit area), not an absolute power. Hence, indeed, if the receiver and transmitter occupy the same physical space, and the transmitter is a point source, then the power density at the receiver is infinite. Typically, a receiver model will multiply this power density by an effective antenna area and an antenna efficiency to get an absolute received power level.

The received power density can be used to decide at the receiver whether transmission is successful. To do this, modify the Receiver model to get the structure shown in figure 28. The blocks used here are found as follows:

- RecordDisassembler: Actors→FlowControl→Aggregators

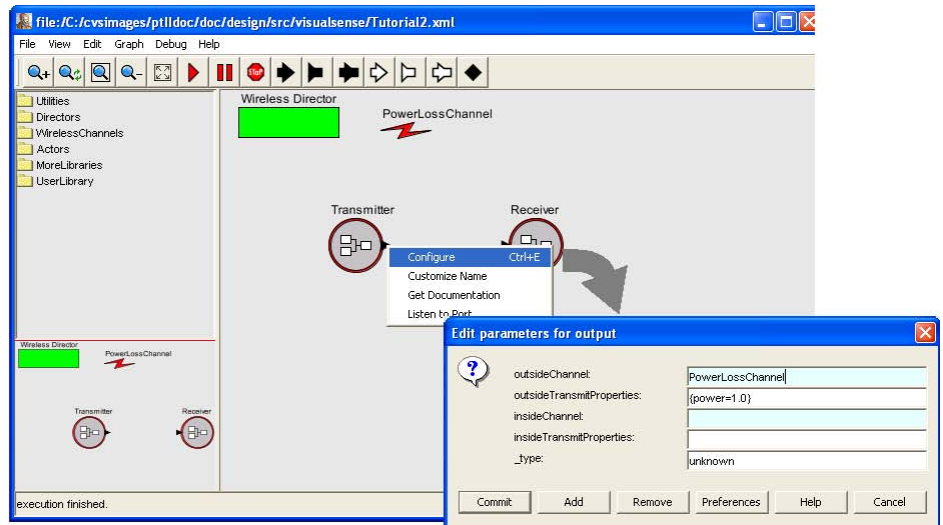
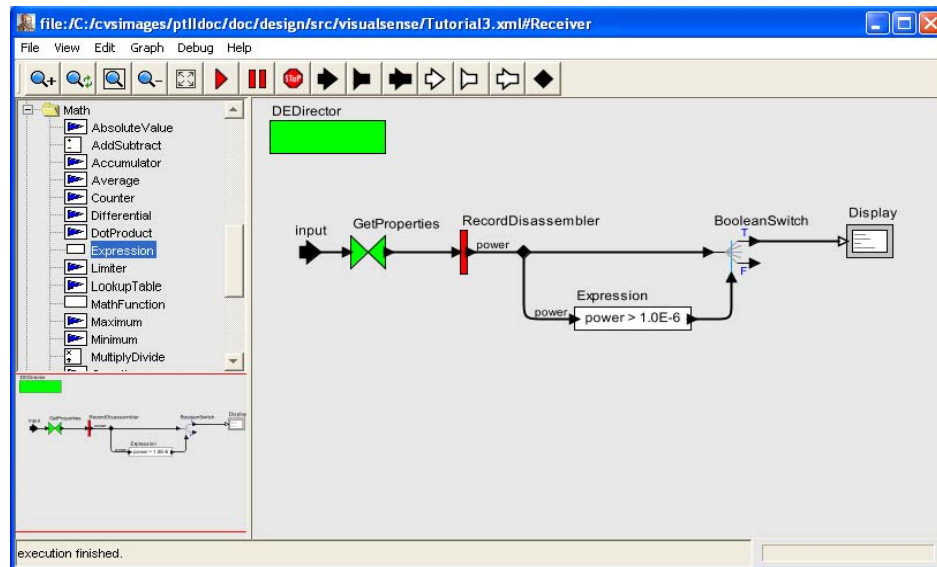


FIGURE 26. Setting the transmit power of the Transmitter.



- Expression: Math
- BooleanSwitch: Actors→FlowControl→BooleanFlowControl The RecordDisassembler actor extracts fields from a record. To use it, you must create output ports that have the same name as the field, in this case, power. To use the Expression actor, you must create

input ports, using whatever names you like (“power” in figure 28), and then give an expression that defines the output in terms of the inputs (“power > 1.0E-6” in figure 28). The output of this Expression actor will be true if the received power is greater than 1.0×10^{-6} , and false otherwise. That boolean signal drives the control port of the BooleanSwitch, which sends its input to one of two output ports depending on the value of the control input. In this case, we observe only the true output, which will be the received power values that exceed 1.0×10^{-6} .

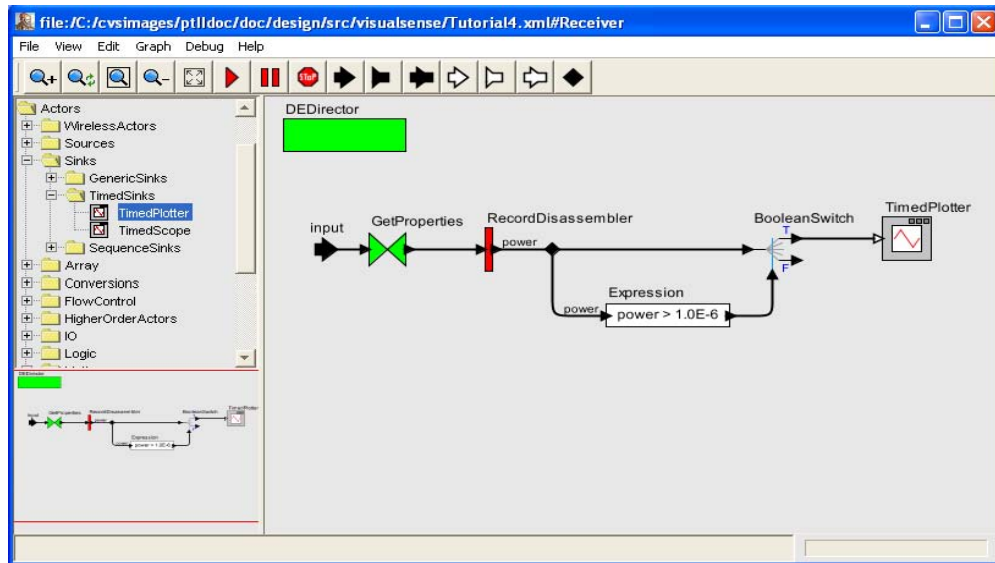
Notice that in figure 28, some connections involve a small black diamond. This is the visual mechanism for routing a signal to multiple places. To create the diamond (which is called a vertex), you can either control click on the background of the editor, or click on the black diamond in the toolbar. To link wires to the vertex, hold the control key while clicking and dragging to draw the connection.

Using the Plot Blocks

Often, it is more useful for a model to graph data rather than display it in textual form. Modify the model of figure 28 as shown in figure 29, where the Display actor has been replaced by a XTime_YData_Plotter from Results->Plotter. The result of a run is shown in figure 30, where the Receiver was moved during the execution so it passed very close to the Transmitter.

This plot display can be improved considerably. In the plot window, click on the format button at the upper right, as shown in figure 30, to get the window shown in figure 31. Setting the parameters as indicated in that window results in the plot in figure 32, which is a more appealing rendition of the data.

Notice that you can zoom into a region of the plot by simply clicking and dragging out the region of interest. You can zoom out by clicking and dragging upwards or leftwards rather than downwards or rightwards. You can zoom fit by clicking on the zoom fit button at the upper right.



Modeling Capabilities

VisualSim Wireless is an extension of the Digital modeler of VisualSim. It largely preserves the discrete-event semantics, but changes the mechanism for connecting components so that explicit wires are not required. In the models constructed in the previous section, wired and wireless models were combined hierarchically. Indeed, all of VisualSim, which includes a very rich set of modeling mechanisms, can be used to construct very elaborate models of sensor nodes and propagation effects.

In this section, we explain the channel model that is used to decide connectivity in sensor nets and the hierarchical component model for each sensor node. We then illustrate capabilities by discussing some of the examples that are provided as demos with the system.

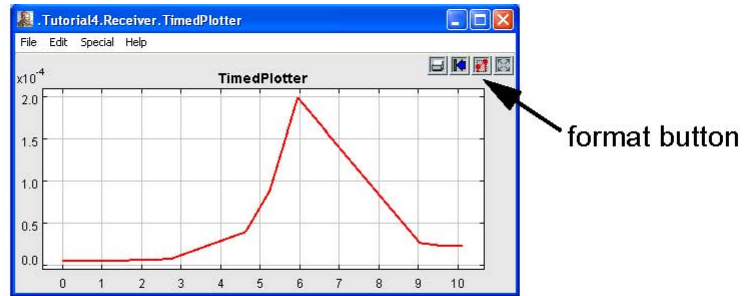
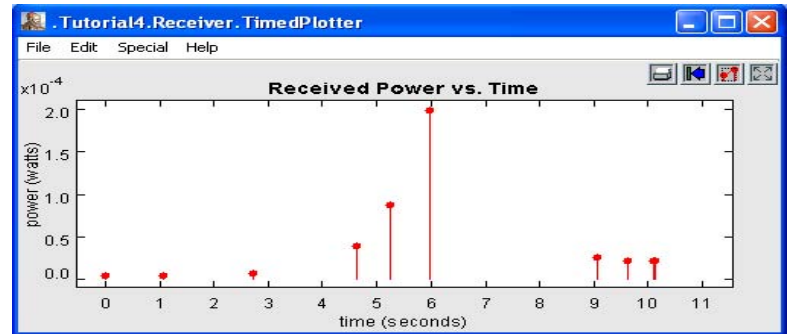
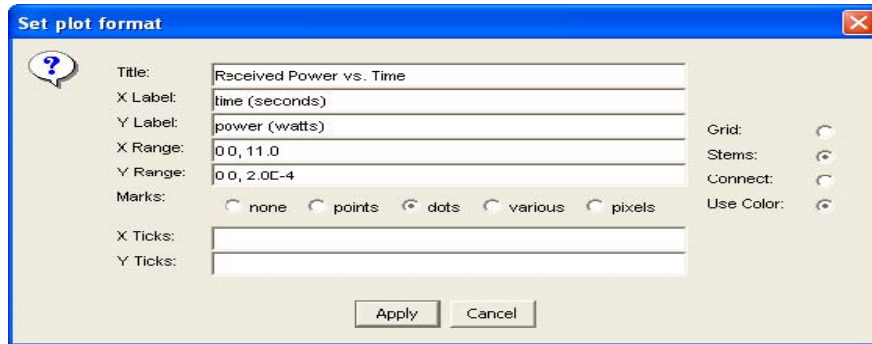


FIGURE 30. Plot showing the received power a function of time as the Receiver is moved close to the Transmitter.



Channel Models

A channel model in VisualSim Wireless is itself a block. When a transmitter produces an event on a wireless port that references the channel by name, the event is delivered to the channel for transformation. The channel may alter the properties that are supplied by the transmitter, and may delay delivery of the event to a receiver to model propagation delay. In VisualSim Wireless, the responsibility of the channel ends there. Other components are used to model terrain effects, antenna gains, etc. Some of these are described below.

Wireless Node Models

Sensor nodes themselves can be modeled in Java, or more interestingly, using more conventional Digital models (as block diagrams) or other VisualSim models (such as dataflow models, finite-state machines or continuous-time models). For example, a sensor node with modal behavior can be defined by sketching a finite-state machine and providing refinements to each of the states to define the behavior of the node in that state. This can be used, for example, to model energy consumption as a function of state.

Sophisticated models of the coupling between energy consumption and media access control protocols become possible.

Examples of Modeling Capabilities

Packet Structure

VisualSim includes a sophisticated type system that includes Data Structures. Above, we showed how Data Structures can be used for transmit properties. They can also be used to construct packets with arbitrary payloads.

Packet Losses

The ErasureChannel model, which is a base class for most of the channel models, offers a parameter `lossProbability` that can be used to model independent, identically distributed packet losses.

Battery Power

Since nodes in a wireless network can be defined by arbitrary VisualSim models, it is easy to incorporate models of energy or power consumption. A simple example is given in the quick tour under “Circular Range Channel,” shown in figure 34, where on the right you can see that the Transmitter uses a PoissonClock to decrease the range of transmission at random times to model the transmission range degradation over time as its battery is depleted. When this model executes, the size of the circular icon representing the transmitter decreases as its range decreases.

Power Loss

The quick tour includes a model called “Power Loss Channel” that illustrates power variability at the receiver as a function of distance. The top-level model, receiver implementation, and a plot resulting from its execution are shown in figure 33. The model uses the same principles as the tutorial example described above.

Collisions

In the underlying discrete-event semantics of VisualSim Wireless, events occur instantaneously at a particular time. That is, they do not have a duration. To model collisions of messages that take time and share a common channel, the model must explicitly include the message duration.

A simple example of such a model is shown in figure 35. In this model, two transmitters share the same channel and transmit messages of fixed duration at random times. As the model executes, one of the transmitters moves in a circular pattern, starting far from the receiver, coming close, then moving away again. At the start, when it is far from the receiver, its messages get through to the receiver only if the other transmitter does not transmit a message that overlaps in time. Whether the message from the other transmitter gets through in the event of a collision depends on how far away the first transmitter is. If it is sufficiently far away, then the interfering power is not sufficient to prevent communication, so the message gets through. If it is closer, then the interfering power will be sufficient that neither message gets through.

Two plots are shown in figure 35. The upper plot shows the messages that are transmitted (in red and blue), giving a visual indication of when overlap occurs. The magnitude in the plot represents the received power. For the transmitter that is stationary, the receiver power is constant. For the transmitter that moves, the received power starts low, then rises to nearly equal the power of the stationary transmitter, then drops again. The lower plot indicates whether messages are lost. In the figure, a total of seven messages are lost, all but one of them from the mobile transmitter (shown in red, if you have a color copy of this document). The duration of a message in this model is represented by an extra field added to the transmit properties by the channel. The parameters of the channel are shown at the lower right in figure 35. Notice that the

defaultProperties parameter has value “{range=Infinity, power=Infinity, duration=1.0}”. The duration field in this record represents the duration of a message. Individual transmitters can override this by setting the outsideTransmitProperties parameters of their ports to give any desired duration.

The Receiver implementation is shown in figure 36. In this model, the value of the received signal is a boolean with value false if the originator is the fixed transmitter and value true if the originator is the mobile transmitter. The GetProperties actor is used to extract the received properties, which will include the received power and the message duration. The power and duration fields of the properties record are extracted by the RecordDisassembler actor and fed into the CollisionDetector actor, which determines which of the messages are received and which are lost. The rest of the model is devoted to constructing meaningful plots so that we get a visual rendition of the behavior.

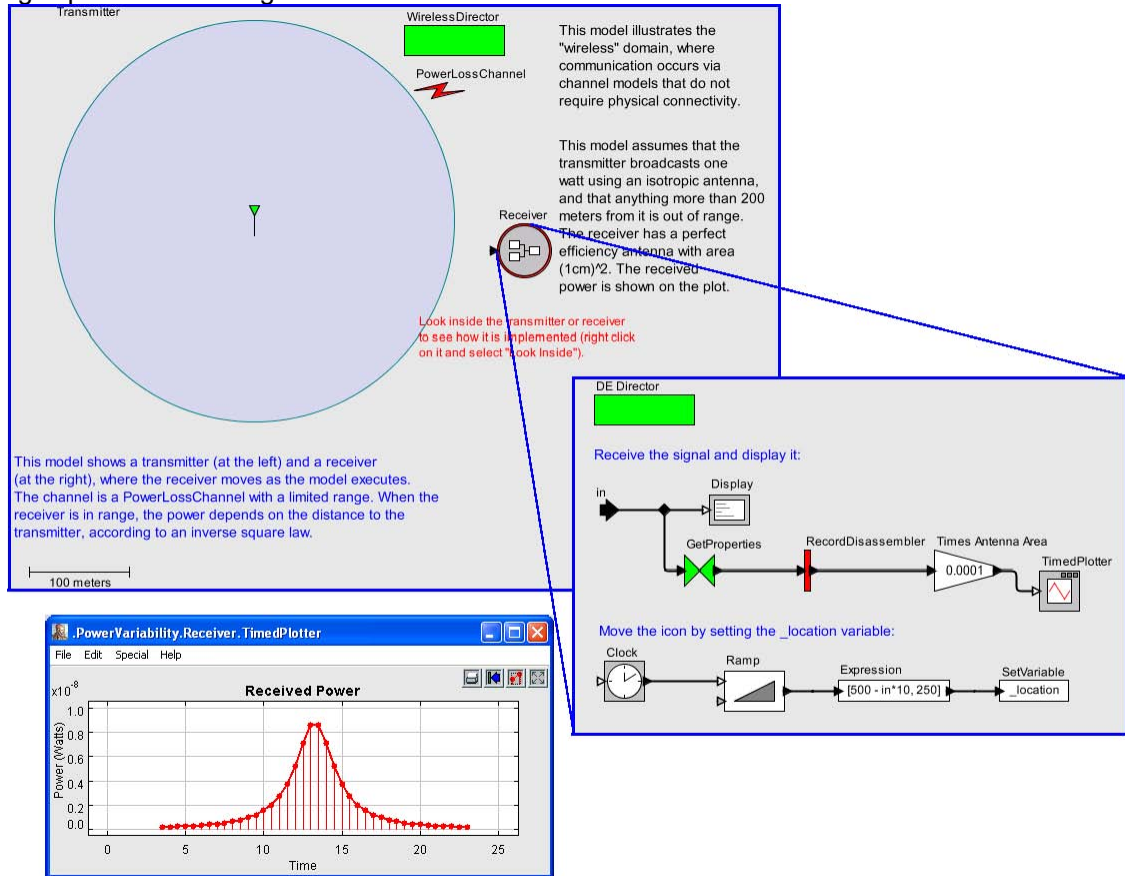


FIGURE 33. Model of power loss as a receiver moves into range and then close to a transmitter.

The CollisionDetector actor is fairly sophisticated. Its documentation is shown in figure 37. This actor assumes that the duration of messages is short relative to the rate at which the actors move. That is, the received power (and whether a receiver is in range) is determined once, at the time the message starts, and remains constant throughout the transmission.

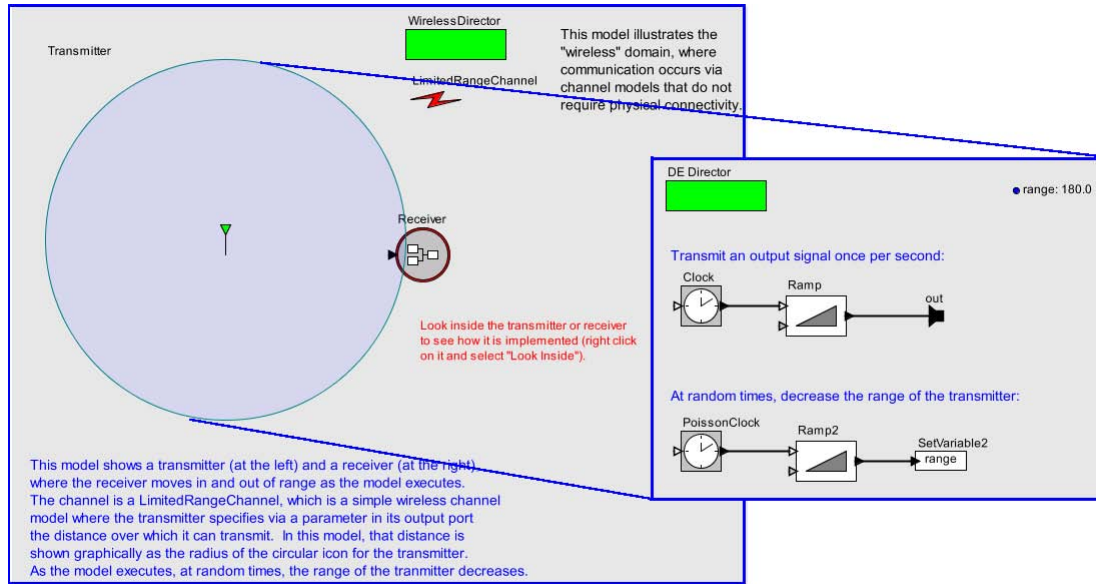


FIGURE 34. Model where transmission range degrades over time as a battery is depleted.

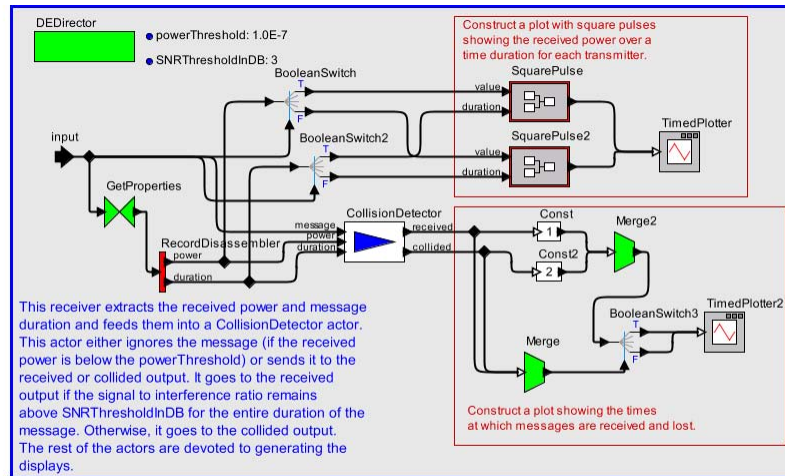


FIGURE 36. Implementation of the Receiver in figure 35, which models and tracks collisions.

Transmit Antenna Gain

A transmitter for a wireless channel may have a directional antenna. This introduces a significant complication in modeling because, although the directionality is a local property of the transmitter, its effect depends on the location of the receiver. We have seen above the use of transmit properties to model propagation losses. Transmit properties are also used to model antenna gains. The transmitter registers with the channel a property transformer, which is an actor that will modify the transmit properties for any particular transmission. Before the channel delivers an event to a receiver, it executes the property transformer, informing it of the location of the transmitter and receiver, and permitting it to modify the transmit properties.

An example of a model that includes a directional transmit antenna is shown in figure 38. This model is visible in the quick tour under "Transmit Antenna Gain." When this model executes, the receiver moves in a circular pattern around the transmitter and measures and plots the received power. The transmitter has an 8-element phased-array antenna with steering.

The design of the transmitter is quite sophisticated, as is shown in figure 39. It illustrates how the full modeling power of VisualSim can be used in VisualSim Wireless. At the top left of the figure, the Trans-

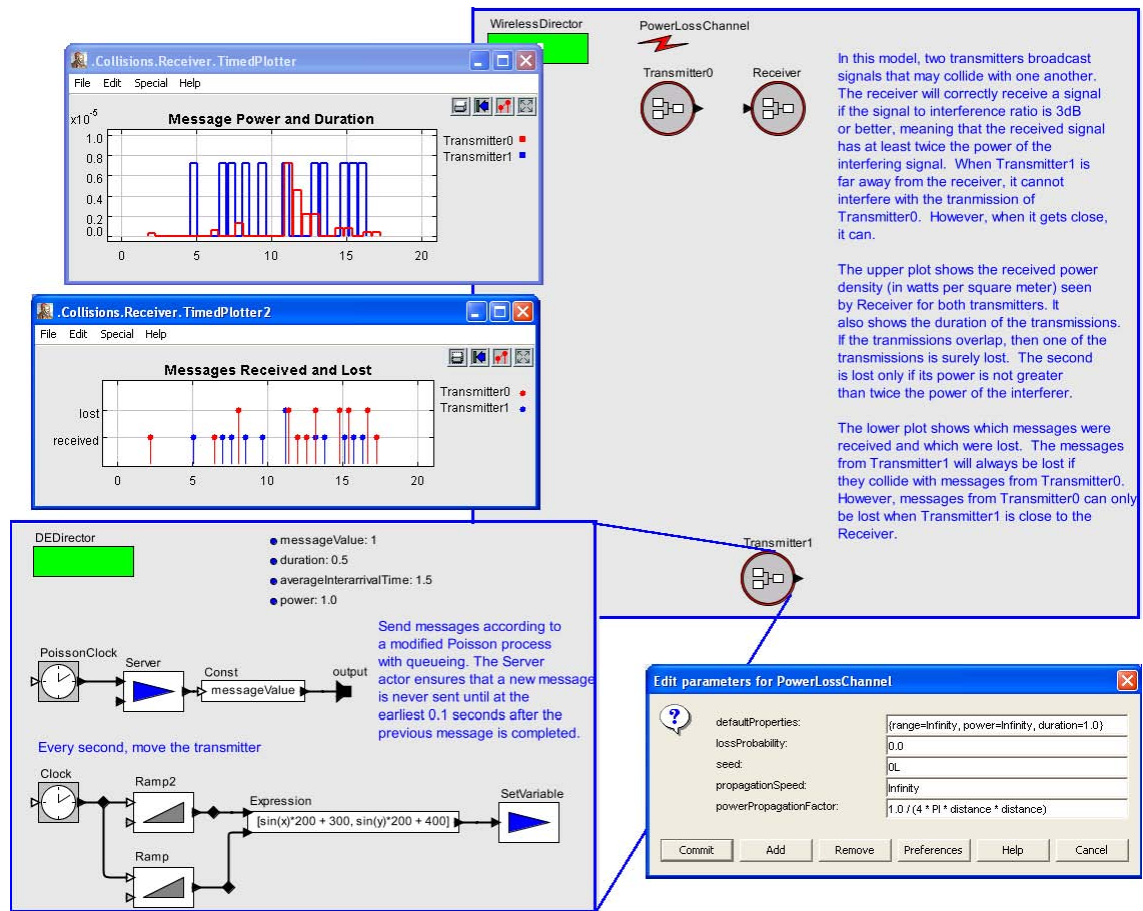


FIGURE 35. Model of collisions of messages that take time.

CollisionDetector: This actor models a typical physical layer front end of a wireless receiver. It models a receiver where messages have a non-zero duration and messages can collide with one another, causing a failure to receive. A message is provided to this actor at the time corresponding to the start of its transmission. Along with the message (an arbitrary token), the inputs must provide the duration of the message and its power. The message spans an interval of time starting when it is provided to this actor and ending at that time plus the duration. If another message overlaps with a given message and has sufficient power, then the given message will be sent to the collided output. Otherwise it is sent to the received output. In both cases, the message appears at the corresponding output at the time it is received plus the duration (i.e. the time at which the message has been completed).

The inputs are:

- message: The message carried by each transmission.
- power: The power of the received signal at the location of this receiver.
 - duration: The time duration of the transmission. The power and duration are typically delivered by the channel in the “properties” field of the transmission. The power is usually given as a power density (per unit area) so that a receiver can multiply it by its antenna area to determine the received power. It is in a linear scale (vs. DB), typically with units such as watts per square meter. The duration is a non-negative double, and the message is an arbitrary token.

The outputs are:

- received: The message received. This port produces an output only if the received power is sufficient and there are no collisions. The output is produced at a time equal to the time this actor receives the message plus the value received on the duration input.
- collided: The message discarded. This port produces an output only if the received message collides with another message of sufficient power. The output is produced at a time equal to the time this actor receives the message plus the value received on the duration input. The value of the output is the message that cannot be received.

This actor is typically used with a channel that delivers a properties record token that contains power and duration

fields. These fields can be extracted by using a GetProperties actor followed by a RecordDisassembler. The PowerLossChannel, for example, can be used. However, in order for the type constraints to be satisfied, the PowerLossChannel's defaultProperties parameter must be augmented with a default value for the duration. Each transmitter can override that default with its own message duration and transmit power.

Any message whose power (as specified at the power input) is less than the value of the powerThreshold parameter is ignored. It will not cause collisions and is not produced at the collided output. The power-Threshold parameter thus specifies the power level at which the receiver simply fails to detect the signal. It is given in a linear scale (vs. DB) with the same units as the power input. The default value is zero, i.e. by default it won't ignore any received signal.

Any message whose power exceeds powerThreshold has the potential of being successfully received, of failing to be received due to a collision, and of causing a collision. A message is successfully received if throughout its duration, its power exceeds the sum of all other message powers by at least SNRThreshold-InDB (which as the name suggests, is given in decibels, rather than in a linear scale, as is customary for power ratios). Formally, let the message power for the i -th message be $p_i(t)$ at time t . Before the message is received and after its duration expires, this power is zero. The i -th message is successfully received if

$$p_i(t) \geq P \sum_{j \neq i} p_j(t) \text{ for all } t \text{ where } p_i(t) > 0, \text{ where } P = 10^{(\text{SNRThresholdInDB}/10)}, \text{ which is the signal to interference ratio in a linear scale.}$$

FIGURE 37. Documentation for the CollisionDetector actor used in figure 36.

mitPropertyTransformer actor models the transmitter antenna. Its firing behavior is very simple: when presented with an input token, it simply produces that same input token, unchanged, on the output port. However, in addition to this firing behavior, this actor registers itself with the channel used by the port that its output is connected to as a property transformer. When wireless communication occurs through that output port to some receiver, the channel calls back the TransmitPropertyTransformer once for each receiver, provides the location of the receiver, and executes the model contained by the TransmitPropertyTransformer actor.

The model contained by the TransmitPropertyTransformer actor is shown in figure 39. At the top right is the top level of this model. It shows that when it is executed (on request by the channel, once for each transmission), it is provided with three values, senderLocation, receiverLocation, and properties. The properties value is a record that in this case includes a power field that is to be modified by the model to account for the antenna gain in the direction from the transmitter to the receiver. This model calculates the angle of the transmission, calculates the antenna gain in that direction, and then scales the power field of the properties record. Notice that this model has an Un-Timed Simulator rather than the usual Wireless_Simulator or Digital Simulator used most commonly in VisualSim Wireless. This is because the calculation of antenna gain is essentially a signal processing function, something that the Untimed Digital Simulator handles very well.

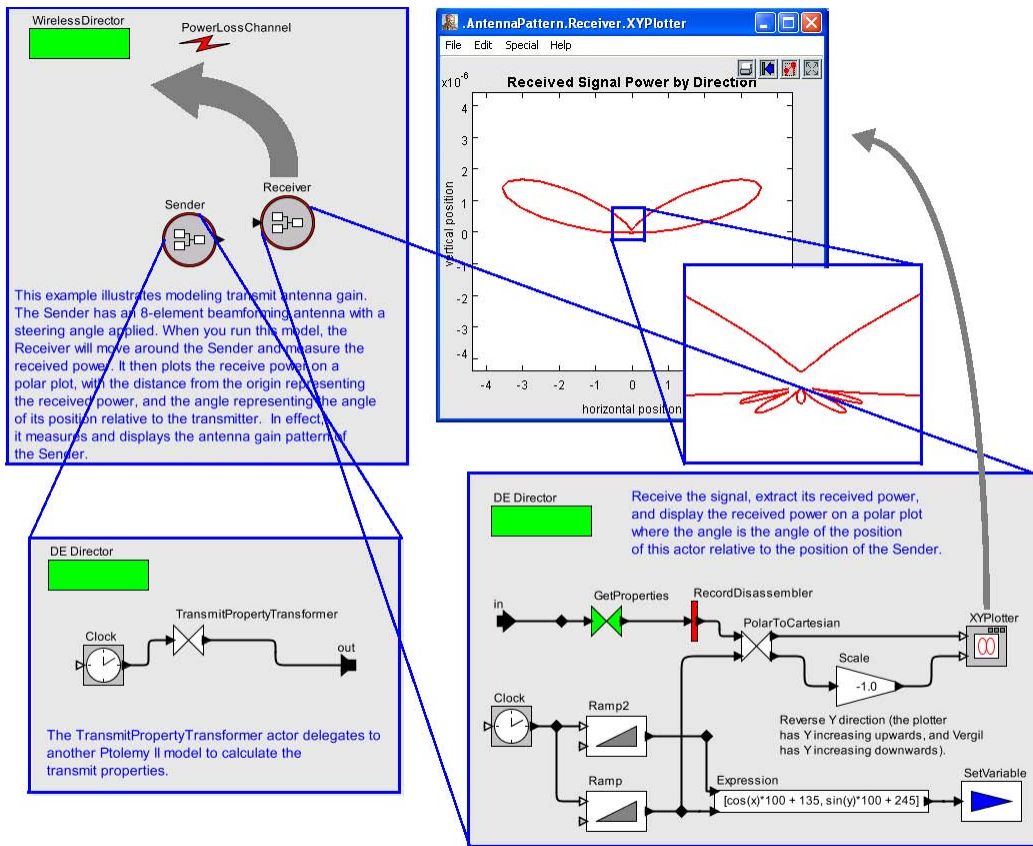


FIGURE 38. Model that includes a directional transmit antenna. As the model executes, the Receiver actor moves in a circular pattern around the transmitter and measures and plots the received power.

The antenna gain is calculated using the model shown in the middle of figure 39. This model uses two IterateOverArray actors (named “ArrayElements” and “Steering”) to model the antenna array elements and application of the steering vector. These actors are hierarchical blocks that execute their contained models once for each element of an input array. These actors are examples of higher-order components, and in this case enable the definition of a model where the number of antenna elements is given by a parameter rather than hardwired into the diagram. The same mechanism can be used to model the antenna gain pattern of the receiver.

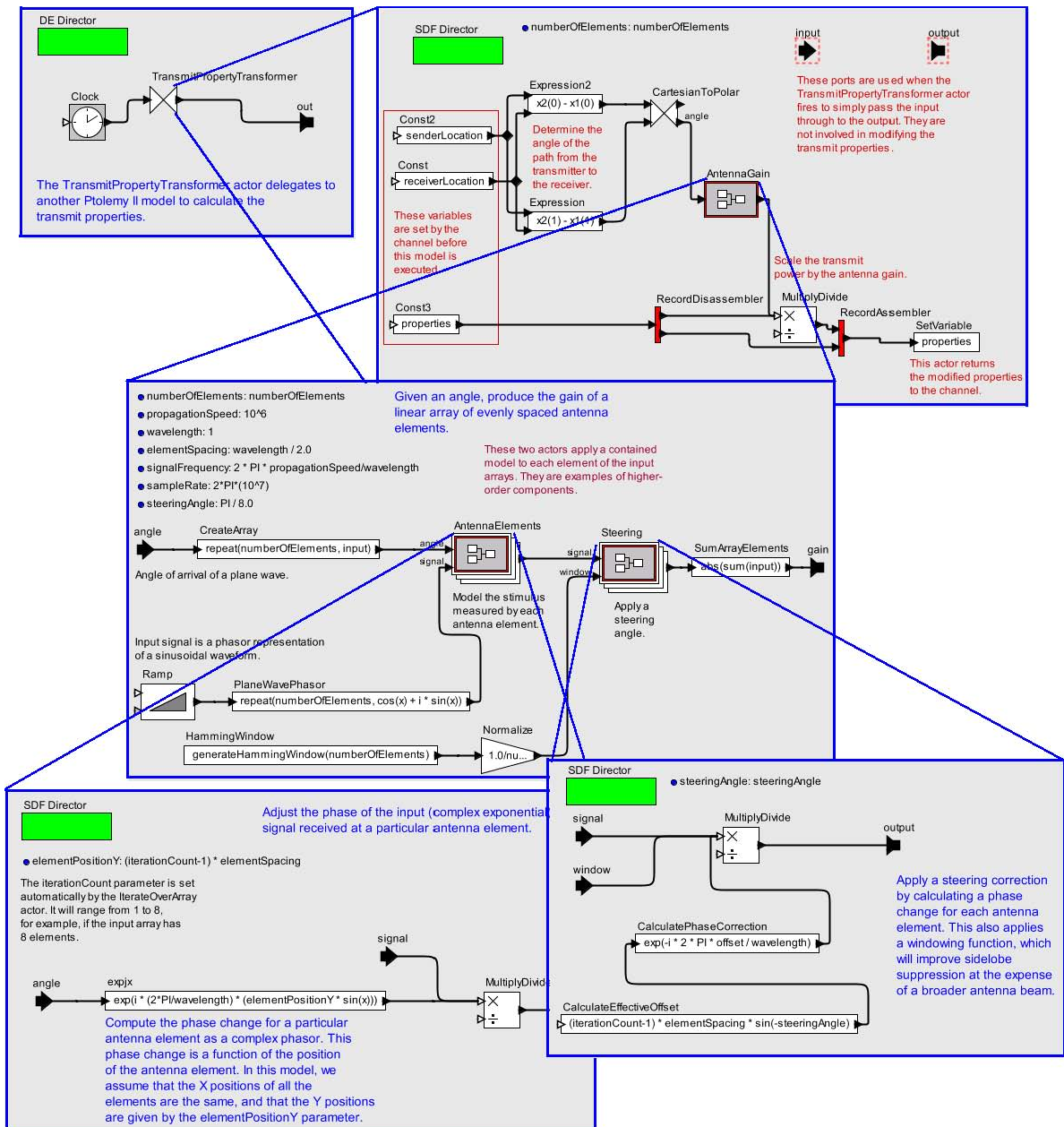


FIGURE 39. Transmitter design for the model in figure 38, showing how a Ptolemy II model (in this case a *synchronous dataflow* model) can be used to model transmission effects.

If there are multiple property transformers that are applicable to a particular transmission, then they are executed in an arbitrary order, so the operations they perform on the properties must be commutative. Typically, they select a field and multiply it by a constant.

Algorithmic

I Analog

This library contains a set of continuous-time blocks designed specifically for use in the CT domain. The continuous time directory of the Model Builder block library contains subdirectories named “event generators and “waveform generators”.

1. **ContinuousClock**: Generate a piecewise-constant signal with instantaneous transitions between levels.
2. **TriggeredContinuousClock**: Generate a piecewise-constant signal with instantaneous transitions between levels, where two input ports are provided to start and stop the clock.
3. **ContinuousSinewave**: Generate a continuous-time sinusoidal signal.

Event Generator

The actors in this sub-library produce discrete event signals, which are signals that only have values at discrete points in time.

1. **EventSource**: Output a set of events at discrete set of time points.
2. **LevelCrossingDetector**: An event detector that converts continuous signals to discrete events when the continuous signal crosses a level threshold.
3. **PeriodicSampler**: Sample the input signal with the specified rate, producing discrete output events.
4. **TriggeredSampler**: Sample the input signal at times where the trigger input has discrete input events.
5. **ThresholdMonitor**: Output true if the input value is in the interval $[a, b]$, which is centered at thresholdCenter and has width thresholdWidth . This block controls the integration step size so that the input does not cross the threshold without producing at least one true output.
6. **ZeroCrossingDetector**: When the trigger is zero (within the specified errorTolerance), then output the value from the input port as a discrete event. This block controls the integration step size to accurately resolve the time at which the zero crossing occurs.

Waveform generators

The blocks in this sub-library convert discrete event signals into continuous-time signals.

1. **ZeroOrderHold**: Convert discrete events at the input to a continuous-time signal at the output by holding the value of the discrete event until the next discrete event arrives.
2. **FirstOrderHold**: Convert discrete events at the input to a continuous-time signal at the output by projecting the value with the derivative.

Control-Analog Functions

The blocks in this sub-library have continuous-time dynamics (i.e., they involve integrators, and hence must coordinate with the differential equation solver).

1. **CTCompositeActor**: Composite block to use when a continuous-time model is created within a continuous-time model.
2. **Integrator**: Integrate the input signal over time to produce the output signal. That is, the input is the derivative of the output with respect to time. This block can be used to close feedback loops in CT to define interesting differential equation systems.
3. **LaplaceTransferFunction**: Filter the input with the specified rational Laplace transform transfer function. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.
4. **LinearStateSpace**: Filter the input with a linear system. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.



5. **DifferentialSystem**: Filter the input with the specified system, which can nonlinear, and is specified using the expression language. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.
6. **RateLimiter**: Limit the first derivative of the input signal, and produce the result as an output sequence.

II Control Systems

This library contains a set of continuous-time blocks designed specifically for use in the CT domain. The continuous time directory of the Model Builder block library contains subdirectories named “event generators and “waveform generators”.

1. **ContinuousClock**: Generate a piecewise-constant signal with instantaneous transitions between levels.
2. **TriggeredContinuousClock**: Generate a piecewise-constant signal with instantaneous transitions between levels, where two input ports are provided to start and stop the clock.
3. **ContinuousSinewave**: Generate a continuous-time sinusoidal signal.

Event Generator

The actors in this sub-library produce discrete event signals, which are signals that only have values at discrete points in time.

1. **EventSource**: Output a set of events at discrete set of time points.
2. **LevelCrossingDetector**: An event detector that converts continuous signals to discrete events when the continuous signal crosses a level threshold.
3. **PeriodicSampler**: Sample the input signal with the specified rate, producing discrete output events.
4. **TriggeredSampler**: Sample the input signal at times where the trigger input has discrete input events.
5. **ThresholdMonitor**: Output true if the input value is in the interval $[a, b]$, which is centered at `thresholdCenter` and has width `thresholdWidth`. This block controls the integration step size so that the input does not cross the threshold without producing at least one true output.
6. **ZeroCrossingDetector**: When the trigger is zero (within the specified `errorTolerance`), then output the value from the input port as a discrete event. This block controls the integration step size to accurately resolve the time at which the zero crossing occurs.

Waveform generators

The blocks in this sub-library convert discrete event signals into continuous-time signals.

1. **ZeroOrderHold**: Convert discrete events at the input to a continuous-time signal at the output by holding the value of the discrete event until the next discrete event arrives.
2. **FirstOrderHold**: Convert discrete events at the input to a continuous-time signal at the output by projecting the value with the derivative.

Control-Analog Functions

The blocks in this sub-library have continuous-time dynamics (i.e., they involve integrators, and hence must coordinate with the differential equation solver).

1. **CTCompositeActor**: Composite block to use when a continuous-time model is created within a continuous-time model.
2. **Integrator**: Integrate the input signal over time to produce the output signal. That is, the input is the derivative of the output with respect to time. This block can be used to close feedback loops in CT to define interesting differential equation systems.
3. **LaplaceTransferFunction**: Filter the input with the specified rational Laplace transform transfer function. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.
4. **LinearStateSpace**: Filter the input with a linear system. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.
5. **DifferentialSystem**: Filter the input with the specified system, which can nonlinear, and is specified using the expression language. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.
6. **RateLimiter**: Limit the first derivative of the input signal, and produce the result as an output sequence.

II Petri Net

1. [PetriNetActor](#): As defined in the PetriNet Simulator, a PetriNetBlock is a directed and weighted graph $G = (V, E)$ containing three kinds of nodes: Places p_i , Transitions t_i , and PetriNetBlockss PA_i , i.e., $V = \{p_i\} \cup \{t_i\} \cup \{PA_i\}$, where each PA_i itself is again defined as a PetriNetBlock. Each node of V is called a component of the PetriNetBlock G . A PetriNetBlock is implemented as an extension of TypedCompositeActor. The current file contains two main methods: `fire()` and `prefire()`. More details of PetriNetBlock can be found in PetriNet Simulator.
2. [PetriNetDirector](#): This Simulator implements the Petri Net model where Places and Transitions form a bipartite graph and enabled Transitions can fire randomly. It also allows Transitions to be replaced by any other block in VisualSim. It implements two forms of Hierarchical and compositional Petri nets. The first form of hierarchical and compositional Petri net semantics comes from the fact that a Transition can contain a sub-Petri-net which is invisible to the director of the container of the Transition. The second form of hierarchical and compositional Petri net semantics comes from a new block called PetriNetBlock which is a collection of Places and Transitions, and those Places and Transitions are visible to the director of the container of the PetriNetBlock. The users can choose which form of models to use, and/or mix them together.
3. [Place](#): A Petri net place is a basic component of the Petri Net model. Another basic component is the Transition. A place is connected to transitions. It contains an integer as the marking of the place, which represents the number of tokens in the place. The operation of the Petri net is controlled by the marking and the weights of arcs connecting places and transitions. The methods here are used to manipulate the integer marking. The TemporaryMarking is used for checking whether a transition is ready or not.

III Image Processing

Basic

1. **Image Reader**: This block reads an Image from a File location, and outputs it as an AWTImageToken.
2. **Image Display (Object)**: Display an image on the screen. For a sequence of images that are all the same size, this block will continually update the picture with new data. If the size of the input image changes, then a new Picture object is created. This block will only accept an ImageToken on its input.
3. **Image Display (Matrix)**: Display an image on the screen. For a sequence of images that are all the same size, this block will continually update the picture with new data. If the size of the input image changes, then a new Picture object is created. This block will only accept a IntMatrixToken on its input, and assumes that the input image contains grayscale pixel intensities between 0 and 255 (inclusive).
4. **URL To Image**: This block reads a String input token naming a URL and outputs an Object Token that contains a `java.awt.Image`.
5. **Image To String**: This block reads an ObjectToken that is a `java.awt.Image` from the input and writes information about the image to the output as a StringToken.
6. **HTVQ Encode**: This block encodes a matrix using Hierarchical Table-Lookup Vector Quantization. The matrix must be of dimensions that are amenable to this method. (i.e. 2x1, 2x2, 4x2, 4x4, etc.) Instead of performing a full-search vector quantization during execution, all the optimal encoding vectors are calculated before hand and stored in a lookup table. (This is known as Table-lookup Vector Quantization).
7. **VQ Decode**: This block decompresses a vector quantized signal. This operation is simply a table lookup into the codebook.
8. **Image Contrast**: This block changes the contrast of an image i.e. if the input image has a lot of pixels with the same or similar color. This block uses gray scale equalization to redistribute the value of each pixel between 0 and 255.
9. **Image Rotate**: The amount of rotation in degrees. This parameter contains an IntegerToken, initially with value 90.

10. **Image Sequence:** Load a sequence of binary images from files, and create a sequence of IntMatrixTokens from them.
11. **Image Partition:** Partition an image into smaller sub-images.
12. **Image Unpartition:** Combine sub-images into a larger image.
13. **PSNR:** This block consumes an IntMatrixToken from each input port, and calculates the Power Signal to Noise Ratio (PSNR) between them. The PSNR is output on the output port as a DoubleToken.

Advanced (Using Java Advanced Imaging)

1. Adaptive Median
2. Double Matrix To JAI
3. Image To JAI
4. JAI Band Combine
5. JAI Band Select
6. JAI Border
7. JAI Box Filter
8. JAI BMP Writer
9. JAI Constant
10. JAI Convolve
11. JAI Crop
12. JAI Data Convert
13. JAI DCT
14. JAI DFT
15. JAI Edge Detection
16. JAI IDCT
17. JAI IDFT
18. **JAI Image Reader:** Supports BMP, FPX, GIF, JPEG, PNG, PBM, PGM, PPM, and TIFF file formats.
19. JAI Invert
20. JAI JPEG Writer
21. JAI Log
22. JAI Magnitude
23. JAI Median Filter
24. JAI Periodic Shift
25. JAI Phase
26. JAI Polar To Complex
27. JAI PNM Writer
28. JAI PNG Writer
29. JAI Rotate
30. JAI Scale
31. JAI TIFF Writer
32. JAI To Double Matrix
33. JAI Translate
34. JAI Transpose
35. Salt And Pepper

Media Interfaces (Using Java Media Framework)

The list of media supported include: aiff, avi, gsm, hotmedia, midi, MPEG 1, MPEG II, Sun Audio and wav formats.

1. **Audio Player:** This actor accepts an ObjectToken that contains a DataSource. This is typically obtained from the output of the StreamLoader actor. This actor will play Datasources containing a MP3, MIDI, and CD Audio file. After the model is run, a window will pop up allowing control of playing, rate of playback, and volume control.
2. **Color Finder:** A block that searches for a color in a Buffer.

- 3 **Image To JMF:**
- 4 **Movie Reader:** Media formats supported are listed on the [Sun Web Site](#).
- 5 **Movie Writer:** Media formats supported are listed on the [Sun Web Site](#).
- 6 **Play Sound:**
- 7 **Stream Loader:**
- 8 **Video Camera:**
- 9 **Video Player:**

IV Signal Processing

Sources

1. **Ramp** (extends SequenceSource): Produce a sequence that begins with the value given by init and is incremented by step after each iteration. The types of init and step are required to support addition.
2. **Sinewave** (composite actor): Output successive samples of a sinusoidal waveform. This is a sequence actor.
3. **TimedSinewave:** Timed version of the Sinewave from above.
4. **InteractiveShell** (extends TypedAtomicActor): This actor creates a command shell on the screen, sending commands that are typed by the user to its output port, and reporting strings received at its input by displaying them. Each time it fires, it reads the input, displays it, then displays a command prompt (which by default is ">>"), and waits for a command to be typed. The command is terminated by an enter or return character, which then results in the command being produced on the output
5. **Interpolator** (extends SequenceSource): Produce an output sequence by interpolating a specified set of values. This can be used to generate complex, smooth waveforms.
6. **Pulse** (extends SequenceSource): Produce a sequence of values at specified iteration indexes. The sequence repeats itself when the repeat parameter is set to true. This is similar to the Clock actor, but it is not timed. Whenever it is fired, it progresses to the next value in the values array, irrespective of the current time.
7. **SampleDelay**- Produce a set of initial tokens
8. **SketchedSource** (implements SequenceActor): Output a signal that has been sketched by the user on the screen.

Audio

The audio library provides actors that can read and write audio files, can capture data from an audio input such as a CD or microphone, and can play audio data through the speakers of the computers.

1. **AudioCapture** (extends Source): Capture audio from the audio input port of the computer, or from its microphone, and produce the samples at the output.
2. **AudioReader** (extends Source): Read audio from a URL, and produce the samples at the output.
3. **AudioPlayer** (extends Sink): Play audio samples on the audio output port of the computer, or from its speakers.
4. **AudioWriter** (extends Sink): Write audio data to a file.

Communications

The communications library collects actors that support modeling and design of digital communication systems.

1. **ConvolutionalCoder** (extends Transformer): Encode an input sequence of bits using a convolutional code.

2. **DeScrambler** (extends Transformer): Descramble the input bit sequence using a feedback shift register.
3. **HadamardCode** (extends Source): Produce a Hadamard codeword by selecting a row from a Hadamard matrix.
4. **Interleaver**: This block interleaves a sequence of binary bits.
5. **LineCoder** (extends SDFTransformer): Read a sequence of booleans (of length wordLength) and interpret them as a binary index into the table, from which a token is extracted and sent to the output.
6. **LMSAdaptive** (extends FIR): Filter the input with an adaptive filter, and update the coefficients of the filter using the input error signal according to the LMS (least mean-square) algorithm.
7. **RaisedCosine** (extends FIR): An FIR filter with a raised cosine frequency response. This is typically used in communication systems as a pulse shaper or a matched filter.
8. **Scrambler** (extends Transformer): Scramble the input bit sequence using a feedback shift register.
9. **ViterbiDecoder** (extends Transformer): Decode inputs using (hard or soft) Viterbi decoding.

Statistical

A small number of statistical analysis blocks are provided.

1. **Autocorrelation** (extends SDFTransformer): Estimate the autocorrelation by averaging products of the input samples.
2. **PowerEstimate** (extends Transformer): Estimate the power of the input signal.

Filtering

1. **DelayLine** (extends SDFTransformer): In each firing, output the n most recent input tokens collected into an array, where n is the length of initialValues. In the beginning, before there are n most recent tokens, use the tokens from initialValues.
2. **DownSample** (extends SDFTransformer): Read factor inputs and produce only one of them on the output.
3. **FIR** (extends SDFTransformer): Produce an output token with a value that is the input filtered by an FIR filter with coefficients given by taps.
4. **IIR** (extends Transformer): Produce an output token with a value that is the input filtered by an IIR filter using a direct form II implementation.
5. **Lattice** (extends Transformer): Produce an output token with a value that is the input filtered by an FIR lattice filter with coefficients given by reflectionCoefficients.
6. **LinearDifferenceEquationSystem** (extends Transformer): Linear system given by an [A, b, c, d] statespace model.
7. **LMSAdaptive** (extends FIR): Filter the input with an adaptive filter, and update the coefficients of the filter using the input error signal according to the LMS (least mean-square) algorithm.
8. **RecursiveLattice** (extends Transformer): Produce an output token with a value that is the input filtered by a recursive lattice filter with coefficients given by reflectionCoefficients.
9. **UpSample** (extends SDFTransformer): Read one input token and produce factor outputs, with all but one of the outputs being a zero of the same type as the input.
10. **VariableFIR** (extends FIR): Filter the input sequence with an FIR filter with coefficients given on the newTaps input port. The blockSize parameter specifies the number of successive inputs that are processed for each set of taps provided on newTaps.
11. **VariableLattice** (extends Lattice): Filter the input sequence with an FIR lattice filter with coefficients given on the newCoefficients input port. The blockSize parameter specifies the number of successive inputs that are processed for each set of taps provided on newCoefficients.

12. **VariableRecursiveLattice** (extends Lattice): Filter the input sequence with a recursive lattice filter with coefficients given on the newCoefficients input port. The blockSize parameter specifies the number of successive inputs that are processed for each set of taps provided on newCoefficients.

Spectrum

1. **DB** (extends Transformer): Produce a token that is the value in decibels ($k \cdot \log_{10}(z)$) of the token received, where k is 10 if inputIsPower is true, and 20 otherwise. The output is never less than min (it is clipped if necessary).
2. **FFT** (extends SDFTransformer): A fast Fourier transform of size 2order.
3. **IFFT** (extends SDFTransformer): An inverse fast Fourier transform of size 2order.
4. **LevinsonDurbin** (extends SDFTransformer): Calculate the linear predictor coefficients (for both an FIR and Lattice filter) for the specified autocorrelation input.
5. **MaximumEntropySpectrum** (composite actor): A fancy spectrum estimator that uses the Levinson-Durbin algorithm to calculate linear predictor coefficients, and then uses those as a parametric model for the random process.
6. **Periodogram** (composite actor): A spectrum estimator calculates a periodogram.
7. **PhaseUnwrap** (extends Transformer): A simple phase unwrapper.
8. **Quantizer**: Produce an output token with the value in levels that is closest to the input value.
9. **SmoothedPeriodogram** (composite actor): A spectrum estimator called the Blackman-Tukey algorithm, which estimates an autocorrelation function by averaging products of the input samples, and then calculates the FFT of that estimate.
10. **Spectrum** (composite actor): A simple spectrum estimator that calculates the FFT of the input. For a random process, this is called the periodogram spectral estimate.

VisualSim Custom Development

1 Custom-Coded Blocks using Java

1.1 Overview

VisualSim is a component-based design. The simulators define the semantics of the interaction between components. This chapter explains the common, simulator-independent principles in the design of components that are blocks. Blocks are components with input and output that at least conceptually operate concurrently with other blocks.

The functionality of blocks in VisualSim can be defined in a number of ways. The most basic mechanism is hierarchy, where a block is defined as a composite of other blocks. But composites are not always the most convenient. Defining the blocks in an advanced language like the SmartMachine can reduce the overhead burden of definition, threads and dealing with complex debuggers. Using Expression block, for instance, is often more convenient for involved mathematical relations. The functionality is defined using the expression language explained in an earlier chapter. Alternatively, you can use the MatlabExpression block and give the behavior as a MATLAB script (assuming you have MATLAB installed). You can also define the behavior of a block in Python, using the PythonActor or PythonScript actor. But the most flexible method is to define the actor in Java.

Some blocks are designed to be simulator polymorphic, meaning that they can operate in various simulators. Others are simulator specific. This chapter explains how to design blocks so that they are maximally simulator polymorphic. As also explained in the previous chapter, many blocks are also data polymorphic. This means that they can operate on a wide variety of token types. Simulator and data polymorphism help to minimize the amount of duplicated code when writing blocks.

Code duplication can also be avoided using object-oriented inheritance. Inheritance can also be used to enforce consistency across a set of classes. Three base classes, Source, Sink, and Transformer, exist to ensure consistent naming of ports and to avoid duplicating code associated with those ports. Since most blocks in the library extend these base classes, users of the library can guess that an input port is named "input" and an output port is named "output," and they will probably be right. Using base classes avoids input ports named "in" or "inputSignal" or something else. This sort of consistency helps to promote re-use of blocks because it makes them easier to use. Thus, we recommend using a reasonably deep class hierarchy to promote consistency.

1.2 Anatomy of a Block

Each block consists of a source code file (or, rarely, a class file) written in Java. Sources are compiled to Java byte code as directed by the Ant- Build.XML script in the user directory. When creating a new block, the Ant function can be used to compile the Java

code. ModelBuilder, described fully in its own chapter, is the graphical design tool commonly used to compose blocks and other components into a complete program, a “VisualSim model.” To facilitate use of a block in ModelBuilder, it must appear in one of the block libraries. This permits it to be dragged from the library palette onto the design canvas. The libraries are XML files. The user block libraries are located in \$VS/User_Library/lib directory. The basic structure of a block is shown in figure 1. In that figure, keywords in bold are features of VisualSim that are briefly described here. Italic text would be substituted with something else in an actual block definition. We will go over this structure in detail in this chapter. The source code for existing VisualSim blocks, located in \$VS/VisualSim/simulators/(de, ct, sdf, fsm)/lib, should also be viewed as a key resource.

1.2.1 Ports

By convention, ports are public members of blocks. They represent a set of input and output *channels* through which tokens may pass to other ports. Figure 1 shows a single port *portName* that is an instance of TypedIOPort, declared in the line

```
public TypedIOPort portName;
```

Most ports in blocks are instances of TypedIOPort, unless they require simulator-specific services, in which case they may be instances of a simulator-specific subclass, such as DEIOPort. The port is actually created in the constructor by the line

```
portName = new TypedIOPort(this, "portName", true, false);
```

The first argument to the constructor is the container of the port, this block. The second is the name of the port, which can be any string, but by convention, is the same as the name of the public member. The third argument specifies whether the port is an input (it is in this example), and the fourth argument specifies whether it is an output (it is not in this example). There is no difficulty with having a port that is both input and output, but it is rarely useful to have one that is neither.

Multiports and Single Port. A port can be a single port or a multiport. By default, it is a single port. It can be declared to be a multiport with a statement like

```
portName.setMultiport(true);
```

All ports have a *width*, which corresponds to the number of channels the port represents. If a port is not connected, the width is zero. If a port is a single port, the width can be zero or one. If a port is a multiport, the width can be larger than one.

```
/** Javadoc comment for the class. */
public class ClassName extends BaseClass implements MarkerInterface {
    /** Javadoc comment for constructor. */
    public ClassName(CompositeEntity container, String name)
        throws NameDuplicationException, IllegalActionException {
        super(container, name);
        // Create and configure ports, e.g. ...
        portName = new TypedIOPort(this, "portName", true, false);
        // Create and configure parameters, e.g. . . .
        parameterName = new Parameter(this, "parameterName");
        parameterName.setTypeEquals(BaseType.DOUBLE);
    }
    ////////////////////////////////////////////////////////////////////
    /// ports and parameters ///
    /** Javadoc comment for port. */
```



```

public TypedIOPort portName;
/** Javadoc comment for parameter. */
public Parameter parameterName;
////////////////////////////////////
//// public methods ////
/** Javadoc comment for fire method. */
public void fire() {
    super.fire();
    ... read inputs and produce outputs ...
}
/** Javadoc comment for initialize method. */
public void initialize() {
    super.initialize();
    ... initialize local variables ...
}
/** Javadoc comment for prefire method. */
public boolean prefire() {
    ... determine whether firing should proceed and return false if not ...
    return super.prefire();
}
/** Javadoc comment for postfire method. */
public boolean postfire() {
    ... update persistent state ...
    ... determine whether firing should continue to next iteration and return false if not ...
    return super.postfire();
}
/** Javadoc comment for wrapup method. */
public void wrapup() {
    super.wrapup();
    ... display final results ...
}
}

```

Figure 1-1 Anatomy of a Block/ Library Block

Reading and Writing. Data (encapsulated in a *token*) can be sent to a particular channel of an output multiport with the syntax

```
portName.send(channelNumber, token);
```

where *channelNumber* is the number of the channel (beginning with 0 for the first channel). The width of the port, the number of channels, can be obtained with the syntax

```
int width = portName.getWidth();
```

If the port is unconnected, then the token is not sent anywhere. The send() method does not complain.

Note that in general, if the channel number refers to a channel that does not exist, the send() method does not complain.

A token can be sent to all output channels of a port (or none if there are none) with the syntax

```
portName.broadcast(token);
```

You can generate a token from a value and then send this token by

```
portName.send(channelNumber, new IntToken(integerValue));
```

A token can be read from a channel with the syntax

```
Token token = portName.get(channelNumber);
```


You can read from channel 0 of a port and extract the contained value (if you know its type) with the syntax

```
double variableName = ((DoubleToken)portName.get(0)).doubleValue();
```

You can query an input port to see whether such a `get()` will succeed (whether a token is available or can be made available) with the syntax

```
boolean tokenAvailable = portName.hasToken(channelNumber);
```

`hasToken()` has been updated to check for zero width, hence `getWidth()` is no longer necessary. You can also query an output port to see whether a `send()` will succeed using

```
boolean spaceAvailable = portName.hasRoom(channelNumber);
```

although with most current simulators, the answer is always true. Note that the `get()`, `hasRoom()` and `hasToken()` methods throw `IllegalActionException` if the channel is out of range, but `send()` just silently returns.

VisualSim includes a sophisticated type system, described fully in the Type System chapter. This type system supports specification of type constraints in the form of inequalities between types. These inequalities can be easily understood as representing the possibility of lossless conversion. Type *a* is less than type *b* if an instance of *a* can be losslessly converted to an instance of *b*. For example, `IntToken` is less than `DoubleToken`, which is less than `ComplexToken`. However, `LongToken` is not less than `DoubleToken`, and `DoubleToken` is not less than `LongToken`, so these two types are said to be *incomparable*.

Suppose that you wish to ensure that the type of an output is greater than or equal to the type of a parameter. You can do so by putting the following statement in the constructor:

```
portName.setTypeAtLeast(parameterName);
```

This is called a *relative type constraint* because it constrains the type of one object relative to the type of another. Another form of relative type constraint forces two objects to have the same type, but without specifying what that type should be:

```
portName.setTypeSameAs(parameterName);
```

These constraints could be specified in the other order,

```
parameterName.setTypeSameAs(portName);
```

which obviously means the same thing, or

```
parameterName.setTypeAtLeast(portName);
```

which is not quite the same.

Another common type constraint is an *absolute type constraint*, which fixes the type of the port (i.e. making it monomorphic rather than polymorphic),

```
portName.setTypeEquals(BaseType.DOUBLE);
```

The above line declares that the port can only handle doubles. Another form of absolute type constraint imposes an upper bound on the type,

```
portName.setTypeAtMost(BaseType.COMPLEX);
```

which declares that any type that can be losslessly converted to ComplexToken is acceptable. By default, for any input port that has no declared type constraints, type constraints are automatically created that declares its type to be less than that of any output ports that have no declared type constraints.

If there are input ports with no constraints, but no output ports lacking constraints, then those input ports will be unconstrained. Conversely, if there are output ports with no constraints, but no input ports lacking constraints, then those output ports will be unconstrained. Of course, you can declare a port to be unconstrained by saying

```
Portname.setTypeAtMost(BaseType.GENERAL);
```

Examples. To be concrete, consider first the code segment shown in figure 2, from the Transformer class in the VisualSim.actor.lib package. This block is a base class for blocks with one input and one output.

The code shows two ports, one that is an input and one that is an output. By convention, the Javadoc1 comments indicate type constraints on the ports, if any. If the ports are multiports, then the Javadoc comment will indicate that. Otherwise, they are assumed to be single ports. Derived classes may change this, making the ports into multiports, in which case they should document this fact in the class comment. Derived classes may also set the type constraints on the ports.

An extension of Transformer is shown in figure 3, the SimplerScale block, which is a simplified version of the Scale block which is defined in \$VS/VisualSim/actor/lib/Scale.java. This block produces an output token on each firing with a value that is equal to a scaled version of the input. The block is polymorphic in that it can support any token type that supports multiplication by the *factor* parameter. In the constructor, the output type is constrained to be at least as general as both the input and the *factor* parameter.

Notice in figure 3 the fire() method uses hasToken() to ensure that no output is produced if there is no input. Also, only one token is consumed from each input channel, even if there is

```
public class Transformer extends TypedAtomicActor {
/** Construct a block with the given container and name.
 * @param container The container.
 * @param name The name of this block.
 * @exception IllegalArgumentException If the block cannot be contained
 * by the proposed container.
 * @exception NameDuplicationException If the container already has an
 * block with this name.
 */
public Transformer(CompositeEntity container, String name)
throws NameDuplicationException, IllegalArgumentException {
super(container, name);
input = new TypedIOPort(this, "input", true, false);
output = new TypedIOPort(this, "output", false, true);
}
////////////////////////////////////
//// ports and parameters ////
/** The input port. This base class imposes no type constraints except
 * that the type of the input cannot be greater than the type of the
 * output.
 */
public TypedIOPort input;
/** The output port. By default, the type of this output is constrained
 * to be at least that of the input.
 */
public TypedIOPort output;
```

Figure 1-2 Code segment showing the port definitions in the Transformer class.

```

import VisualSim.actor.lib.Transformer;
import VisualSim.data.IntToken;
import VisualSim.data.expr.Parameter;
import VisualSim.data.Token;
import VisualSim.kernel.util.*;
import VisualSim.kernel.CompositeEntity;
public class SimplerScale extends Transformer {
...
public SimplerScale(CompositeEntity container, String name)
throws NameDuplicationException, IllegalArgumentException {
super(container, name);
factor = new Parameter(this, "factor", new IntToken(1));
// set the type constraints.
output.setTypeAtLeast(input);
output.setTypeAtLeast(factor);
}
////////////////////////////////////
//// ports and parameters ////
/** The factor.
 * This parameter can contain any token that supports multiplication.
 * The default value of this parameter is the IntToken 1.
 */
public Parameter factor;
////////////////////////////////////
//// public methods ////
/** Clone the block into the specified workspace. This calls the
 * base class and then sets the type constraints.
 * @param workspace The workspace for the new object.
 * @return A new block.
 * @exception CloneNotSupportedException If a derived class has
 * an attribute that cannot be cloned.
 */
public Object clone(Workspace workspace)
throws CloneNotSupportedException {
SimplerScale newObject = (SimplerScale)super.clone(workspace);
newObject.output.setTypeAtLeast(newObject.input);
newObject.output.setTypeAtLeast(newObject.factor);
return newObject;
}
/** Compute the product of the input and the <i>factor</i>.
 * If there is no input, then produce no output.
 * @exception IllegalArgumentException If there is no simulator.
 */
public void fire() throws IllegalArgumentException {
if (input.hasToken(0)) {
Token in = input.getToken();
Token factorToken = factor.getToken();
Token result = factorToken.multiply(in);
output.send(0, result);
}
}
}

```

Figure 1-3 Code segment from the SimplerScale block, showing the handling of ports and parameters.

more than one token available. This is generally the behavior of simulator-polymorphic blocks. Notice also how it uses the multiply() method of the Token class. This method is polymorphic. Thus, this scale block can operate on any token type that supports multiplication, including all the numeric types and matrices.

Note: To create ports and their specified types in such a way as the type of port can be changed in the VisualSim GUI, the "_type" of the port must be defined as a "VisualSim.actor.TypeAttribute". If you use the type parameter from the default class "VisualSim.data.expr.Parameter", the type cannot be changed from the VisualSim GUI.

1.2.2 Port Rates and Dependencies between Ports

Many VisualSim simulators perform analysis of the topology of a model for the purposes of scheduling. SDF, for example, constructs a static schedule that sequences the invocations of actors. DE and CT all examine data dependencies between blocks to prioritize reactions to events that are simultaneous. In all these cases, the director of the simulator requires some additional information about the behavior of blocks in order to perform the analysis. In this section, we explain what additional information you can provide in a block that will ensure that it can be used in all these domains.

Suppose you are designing a block that does not require a token at its input port in order to produce one on its output port. It is useful for the director to have access to this information. For example, the TimedDelay actor of the DE simulator declares that its *output* port is independent of its *input* port by defining this method:

```
public void pruneDependencies() {
    super.pruneDependencies() ;
    removeDependency(input, output);
}
```

An output port has a function dependency on an input port if in its *fire()* method, it sends tokens on the output port that depend on tokens gotten from the input port. By default, blocks declare that each output port depends on all input ports. If the block writer does nothing, this is what a scheduler will assume. By overriding the *pruneDependencies()* method as above, the block writer is asserting that for this particular block, the output port named *output* does not depend on the input named *input* in any given firing.

The scheduler can use this information to sequence the execution of the blocks and to resolve causality loops. For domains that do not use dependency information (such as SDF), it is harmless to include the above the method. Thus, by making such declarations, you maximize the reuse potential of your blocks. Some domains (notably SDF) make use of information about production and consumption rates at the ports of actors. If the block writer does nothing, the SDF will assume that a block requires and consumes exactly one token on each input port when it fires and produces exactly one token on each output port. To override this assumption, the block writer only needs to include a parameter (an instance of `VisualSim.data.expr.Parameter`) in the port that is named either “tokenConsumptionRate” (for input ports) or “tokenProductionRate” (for output ports). The value of these parameters is an integer that specifies the number of tokens consumed or produced in a firing. As always, the value of these parameters can be given by an expression that depends on the parameters of the actor. Including these parameters in the ports is harmless for domains that do not make use of this information, but including them makes such blocks useful in SDF, and hence improves their reusability.

In addition to production and consumption rates, feedback loops in SDF require that at least one actor in the loop produce tokens in its *initialize()* method. To make the SDF scheduler aware that an block does this, include a parameter in the output port that produces these tokens named “tokenInitProduction” with a value that is an integer specifying the number of tokens initially produced. The SDF scheduler will use this information to determine that a model with cycles does not deadlock.

1.2.3 Parameters

Like ports, by convention, parameters are public members of blocks. Figure 3 shows a parameter *factor* that is an instance of `Parameter`, declared in the line

```
public Parameter factor;
```

and created in the line

```
factor = new Parameter(this, "factor", new IntToken(1));
```

The third argument to the constructor, which is optional, is a default value for the parameter. In this example, the *factor* parameter defaults to the integer one. Alternatively, the default value of the parameter can be set via an expression, as in

```
factor = new Parameter(this, "factor");  
factor.setExpression("2*PI");
```

As with ports, you can specify type constraints on parameters. The most common type constraint is to fix the type, using

```
parameterName.setTypeEquals(BaseType.DOUBLE);
```

In fact, exactly the same relative or absolute type constraints that one can specify for ports can be specified for parameters as well. But in addition, arbitrary constraints on parameter values are possible, not just type constraints.

An block is notified when a parameter value changes by having its `attributeChanged()` method called. Consider the example shown in figure 4, taken from the `PoissonClock` block. This block generates timed events according to a Poisson process. One of its parameters is *meanTime*, which specifies the mean time between events. This must be a double, as asserted in the constructor.

The `attributeChanged()` method is passed the parameter that changed. (Typically the user changes it via the `Configure` dialog.) If this is *meanTime*, then this method checks to make sure that the specified value is positive, and if not, it throws an exception. This exception is presented to the user in a new dialog box. It shows up when the user attempts to commit a non-positive value.

The new dialog requests that the users choose a new value or cancel the change. A change in a parameter value sometimes has broader repercussions than just in the local block. It may, for example, impact the schedule of execution of blocks. An block can call the `invalidateSchedule()` method of the simulator, which informs the simulator that any statically computed schedule (if there is one) is no longer valid. This would be used, for example, if the parameter affects the number of tokens produced or consumed when a block fires.

When the type of a parameter changes, the `attributeTypeChanged()` method in the block containing that parameter will be called. The default implementation of this method in `TypedAtomicActor` is to invalidate type resolution. So parameter type change will cause type resolution to be performed in the model. This default implementation is suitable for most blocks. In fact, most of the blocks in the block library do not override this method.

However, if for some reason, a block does not wish to redo type resolution upon parameter type change, the `attributeTypeChanged()` method can be overridden. But this is rarely necessary.

1.2.4 Constructors

We have seen already that the major task of the constructor is to create and configure ports and parameters. In addition, you may have noticed that it calls

```
super(container, name);
```

and that it declares that it throws `NameDuplicationException` and `IllegalActionException`. The latter is the most widely used exception, and many methods in blocks declare that they can throw it. The former is thrown if the specified container already contains an block with the specified name.

1.2.5 Cloning

All blocks are cloneable. An block clone needs to be a new instance of the same class, with the same parameter values, but without any connections to other blocks.

```
public class PoissonClock extends TimedSource {
    public Parameter meanTime;
    public Parameter values;
    public PoissonClock(CompositeEntity container, String name)
        throws NameDuplicationException, IllegalActionException {
        super(container, name);
        meanTime = new Parameter(this, "meanTime", new DoubleToken(1.0));
        meanTime.setTypeEquals(BaseType.DOUBLE);
        ...
    }
    /** If the argument is the meanTime parameter, check that it is
     * positive.
     * @exception IllegalActionException If the meanTime value is
     * not positive.
     */
    public void attributeChanged(Attribute attribute) throws IllegalActionException {
        if (attribute == meanTime) {
            double mean = ((DoubleToken)meanTime.getToken()).doubleValue();
            if (mean <= 0.0) {
                throw new IllegalActionException(this,
                    "meanTime is required to be positive. meanTime given: " + mean);
            }
        } else if (attribute == values) {
            ArrayToken val = (ArrayToken)(values.getToken());
            _length = val.length();
        } else {
            super.attributeChanged(attribute);
        }
    }
}
```

Figure 1-4 Code segment from the `PoissonClock` block, showing the `attributeChanged()` method.

Consider the clone() method in figure 5, taken from the SimplerScale block. This method begins with:

```
SimplerScale newObject = (SimplerScale)super.clone(workspace);
```

The convention in VisualSim is that each clone method begins the same way, so that cloning works its way up the inheritance tree until it ultimately uses the clone() method of the Java Object class. That method performs what is called a “shallow copy,” which is not sufficient for our purposes. In particular, members of the class that are references to other objects, including public members such as ports and parameters, are copied by copying the references. The NamedObj and TypedAtomicActor base classes for most blocks implement a “deep copy” so that all the contained objects are cloned, and public members reference the proper cloned objects2.

Although the base classes neatly handle most aspects of the clone operation, there are subtleties involved with cloning type constraints. Absolute type constraints on ports and parameters are carried automatically into the clone, so clone() methods should never call setTypeEquals().

```
public class SimplerScale extends Transformer {
...
public SimplerScale(CompositeEntity container, String name)
throws NameDuplicationException, IllegalArgumentException {
super(container, name);
output.setTypeAtLeast(input);
output.setTypeAtLeast(factor);
}
////////////////////////////////////
/// ports and parameters ///
/** The factor. The default value of this parameter is the integer 1. */
public Parameter factor;
////////////////////////////////////
/// public methods ///
/** Clone the actor into the specified workspace. This calls the
* base class and then sets the type constraints.
* @param workspace The workspace for the new object.
* @return A new block.
* @exception CloneNotSupportedException If a derived class has
* has an attribute that cannot be cloned. */
public Object clone(Workspace workspace) throws CloneNotSupportedException {
SimplerScale newObject = (SimplerScale)super.clone(workspace);
newObject.output.setTypeAtLeast(newObject.input);
newObject.output.setTypeAtLeast(newObject.factor);
return newObject;
}
}
```

Figure 1-5 Code segment from the SimplerScale block, showing the clone() method.

However, relative type constraints are not cloned automatically because of the difficulty of ensuring that the other object being referred to in a relative constraint is the intended one. Thus, in figure 5, the clone() method repeats the relative type constraints that were specified in the constructor:

```
newObject.output.setTypeAtLeast(newObject.input);
newObject.output.setTypeAtLeast(newObject.factor);
```

Note that at no time during cloning is any constructor invoked, so it is necessary to repeat in the clone() method any initialization in the constructor. For example, the clone() method in the Expression block sets the values of a few private Variables:


```
newObject._iterationCount = 1;  
newObject._time = (Variable)newObject.getAttribute("time");  
newObject._iteration =  
(Variable)newObject.getAttribute("iteration");
```

1.3 Action Methods

Figure 1 shows a set of public methods called the *action methods* because they specify the action performed by the block. By convention, these are given in alphabetical order in VisualSim Java files, but we will discuss them here in the order that they are invoked. The first to be invoked is the `preinitialize()` method, which is invoked exactly once before any other action method is invoked. The `preinitialize()` method is often used to set type constraints. After the `preinitialize()` method is called, type resolution happens and all the type constraints are resolved. The `initialize()` method is invoked next, and is typically used to initialize state variables in the block, which generally depends on type resolution. After the `initialize()` method, the block experiences some number of *iterations*, where an iteration is defined to be exactly one invocation of `prefire()`, some number of invocations of `fire()`, and at most one invocation of `postfire()`.

1.3.1 Initialization

The `initialize()` method of the Average block is shown in figure 6. This data- and simulator-polymorphic block computes the average of tokens that have arrived. To do so, it keeps a running sum in a private variable `_sum`, and a running count of the number of tokens it has seen in a private variable `_count`. Both of these variables are initialized in the `initialize()` method. Notice that the block also calls `super.initialize()`, allowing the base class to perform any initialization it expects to perform. This is essential because one of the base classes initializes the ports. An block will almost certainly fail to run properly if `super.initialize()` is not called.

Note that the initialization of the Average block does not affect, or depend on, type resolution. This means that the code to initialize this block can be placed either in the `preinitialize()` method, or in the `initialize()` method. However, in some cases an block may require part of its initialization to happen before type resolution, in the `preinitialize()` method, or part after type resolution, in the `initialize()` method. For example, a block may need to dynamically create type constraints before each execution. Such a block must create its type constraints in `preinitialize()`. On the other hand, a block may wish to produce (send or broadcast) an initial output token once at the beginning of an execution of a model. This production can only happen during `initialize()`, because data transport through ports depends on type resolution.

```

public class Average extends Transformer {
...
public void initialize() throws IllegalActionException {
    super.initialize();
    _count = 0;
    _sum = null;
}

////////////////////////////////////
//// private members ////
private Token _sum;
private int _count = 0;

```

Figure 1-6 Code segment from the Average block, showing the initialize() method.

1.3.2 Prefire

The `prefire()` method is the only method that is invoked exactly once per iteration. It returns a boolean that indicates to the simulator whether the block wishes for firing to proceed. The `fire()` method of an block should never be called until after its `prefire` method has returned true. The most common use of this method is to test a condition to see whether the block is ready to fire.

Consider for example an block that reads from *trueInput* if a private boolean variable `_state` is *true*, and otherwise reads from *falseInput*. The `prefire()` method might look like this:

```

public boolean prefire() throws IllegalActionException {
    if(_state) {
        return trueInput.hasToken(0);
    } else {
        return falseInput.hasToken(0);
    }
}

```

It is good practice to check the superclass in case it has some reason to decline to be fired. The above example becomes:

```

public boolean prefire() throws IllegalActionException {
    if(_state) {
        return trueInput.hasToken(0) && super.prefire();
    } else {
        return falseInput.hasToken(0) && super.prefire();
    }
}

```

The `prefire()` method can also be used to perform an operation that will happen exactly once per iteration. Consider the `prefire` method of the Bernoulli block in figure 7:

```

public boolean prefire() throws IllegalActionException {
    if (_random.nextDouble() <
        ((DoubleToken)(trueProbability.getToken()).doubleValue())) {
        _current = true;
    } else {
        _current = false;
    }
    return super.prefire();
}

```

This method selects a new Boolean value that will correspond to the token creating during each firing of that iteration.

```

public class Bernoulli extends RandomSource {
public Bernoulli(CompositeEntity container, String name)
throws NameDuplicationException, IllegalArgumentException {
super(container, name);
output.setTypeEquals(BaseType.BOOLEAN);
trueProbability = new Parameter(this, "trueProbability", new DoubleToken(0.5));
trueProbability.setTypeEquals(BaseType.DOUBLE);
}
public Parameter trueProbability;
public void fire() {
try {
super.fire();
output.send(0, new BooleanToken(_current));
} catch (IllegalArgumentException ex) {
// Should not be thrown because this is an output port.
throw new InternalErrorException(ex.getMessage());
}
}
public boolean prefire() throws IllegalArgumentException {
if (_random.nextDouble() <
((DoubleToken)(trueProbability.getToken()).doubleValue()) {
_current = true;
} else {
_current = false;
}
return super.prefire();
}
}
private boolean _current;

```

Figure 1-7 Code for the Bernoulli block, which is not data polymorphic.

1.3.3 Fire

The fire() method is the main point of execution and is generally responsible for reading inputs and producing outputs. It may also read the current parameter values, and the output may depend on them.

Things to remember when writing fire() methods are:

- To get data polymorphism, use the methods of the Token class for arithmetic whenever possible. Consider for example the Average block, shown in figure 8. Notice the use of the add() and divide() methods of the Token class to achieve data polymorphism.
- When data polymorphism is not practical or not desired, then it is usually easiest to use the setTypeEquals() to define the type of input ports. The type system will assure that you can safely cast the tokens that you read to the type of the port. Consider again the Average block shown in figure 9. This block declares the type of its reset input port to be BaseType.BOOLEAN. In the fire() method, the input token is read and cast to a BooleanToken. The type system ensures that no cast error will occur. The same can be done with a parameter, as with the Bernoulli block shown in figure 9.
- A simulator-polymorphic block cannot assume that there is data at all the input ports. Most simulator polymorphic blocks will read at most one input token from each port, and if there are sufficient inputs, produce exactly one token on each output port.
- Some simulators invoke the fire() method multiple times, working towards a converged solution.

Thus, each invocation of `fire()` can be thought of as doing a tentative computation with tentative inputs and producing tentative outputs. Thus, the `fire()` method should not update persistent state. Instead, that should be done in the `postfire()` method, as discussed in the next section.

1.3.4 Postfire

The `postfire()` method has two tasks:

- Updating persistent state, and
- Determining whether the execution of an block is complete.

Consider the `fire()` and `postfire()` methods of the Average block in figure 8. Notice that the persistent state variables `_sum` and `_count` are not updated in `fire()`. Instead, they are shadowed by `_latestSum` and `_latestCount`, and updated in `postfire()`. The return value of `postfire()` is a Boolean that indicates to the simulator whether execution of the block is complete. By convention, the simulator should avoid iterating further an block that returns false. In other words, the simulator won't call `prefire()`, `fire()`, or `postfire()` again during this execution of the model.

Consider the two examples shown in figure 9. These are base classes for source blocks (those with no input ports). `SequenceSource` is a base class for blocks that output sequences. Its key feature is a parameter *firingCountLimit*, which specifies a limit on the number of iterations of the block. When this limit is reached, the `postfire()` method returns false. Thus, this parameter can be used to define sources of finite sequences. `TimedSource` is similar, except that instead of specifying a limit on the number of iterations, it specifies a limit on the current model time. When that limit is reached, the `postfire()` method returns false.

```

public class Average extends Transformer {
... constructor ...
////////////////////////////////////
//// ports and parameters ////
public TypedIOPort reset;
////////////////////////////////////
//// public methods ////
... clone method ...
public void fire() throws IllegalActionException {
    _latestSum = _sum;
    _latestCount = _count + 1;
    // Check whether to reset.
    for (int i = 0; i < reset.getWidth(); i++) {
        if (reset.hasToken(i)) {
            BooleanToken r = (BooleanToken)reset.get(0);
            if(r.booleanValue()) {
                // Being reset at this firing.
                _latestSum = null;
                _latestCount = 1;
            }
        }
    }
    if (input.hasToken(0)) {
        Token in = input.get(0);
        if (_latestSum == null) {
            _latestSum = in;
        } else {
            _latestSum = _latestSum.add(in);
        }
        Token out = _latestSum.divide(new IntToken(_latestCount));
        output.send(0, out);
    }
}

public void initialize() throws IllegalActionException {
    super.initialize();
    _count = 0;
    _sum = null;
}

public boolean postfire() throws IllegalActionException {
    _sum = _latestSum;
    _count = _latestCount;
    return super.postfire();
}
////////////////////////////////////
//// private members ////
private Token _sum;
private Token _latestSum;
private int _count = 0;
private int _latestCount;

```

Figure 1-8 Code segment from the Average block, showing the action methods.

```

public class SequenceSource extends Source implements SequenceActor {
public SequenceSource(CompositeEntity container, String name)
throws NameDuplicationException, IllegalActionException {
super(container, name);
firingCountLimit = new Parameter(this, "firingCountLimit", new IntToken(0));
firingCountLimit.setTypeEquals(BaseType.INT);
}
public Parameter firingCountLimit;
...
public boolean postfire() throws IllegalActionException {
if (_firingCountLimit != 0) {
_iterationCount++;
if (_iterationCount ==
((IntToken)firingCountLimit.getToken()).intValue()) {
return false;
}
}
return true;
}
protected int _firingCountLimit;
protected int _iterationCount = 0;
}
public class TimedSource extends Source implements TimedActor {
public TimedSource(CompositeEntity container, String name)
throws NameDuplicationException, IllegalActionException {
super(container, name);
stopTime = new Parameter(this, "stopTime", new DoubleToken(0.0));
stopTime.setTypeEquals(BaseType.DOUBLE);
...
}
}
public Parameter stopTime;
...
public boolean postfire() throws IllegalActionException {
double time = ((DoubleToken)stopTime.getToken()).doubleValue();
if (time > 0.0 && getDirector().getCurrentTime() >= time) {
return false;
}
return true;
}
}

```

Figure 1-9 Code segments from the SequenceSource and TimedSource base classes.

1.3.5 Wrapup

The wrapup() method is used typically for displaying final results. It is invoked exactly once at the end of an execution, including when an exception occurs that stops execution (as opposed to an exception occurring in, say, attributeChanged(), which does not stop execution). However, when a block is removed from a model during execution, the wrapup() method is not called.

A block may lock a resource (which it intends to release in wrapup() for example). Or its designer may have another reason to ensure that wrapup() always is called, even when the block is removed from an executing model. This can be achieved by overriding the setContainer() method.

In this case, the block would have a `setContainer()` method which might look like this:

```
public void setContainer(CompositeEntity container)
throws IllegalArgumentException, NameDuplicationException {
    if (container != getContainer()) {
        wrapup();
    }
    super.setContainer(container);
}
```

When overriding the `setContainer()` method in this way, it is best to make `wrapup()` idempotent (implying that it can be invoked many times without causing harm), because future implementations of the simulator might automatically unlock resources of removed blocks, or call `wrapup()` on removed blocks.

1.4 Coupled Port and Parameter

Often, in the design of a block, it is hard to decide whether a quantity should be given by a port or by a parameter. Fortunately, you can design a block to offer both options. An example of such a block is shown in figure 9a. This block starts with an initial value, given by the *init* parameter, and increments it each time by the value of *step*. The value of *step* is given by either a parameter named *step* or a port named *step*. To use the parameter exclusively, the model builder simply leaves the port unconnected.

If the port is connected, then the parameter provides the default value, used before anything arrives on the port. But after something arrives on the port, that is used. When the model containing a Ramp block is saved, only the parameter value is stored. No data that arrives on the port is stored. Thus, the default value given by the parameter is persistent, while the values that arrive on the port are transient.

```
public class Ramp extends SequenceSource {
    public Ramp(CompositeEntity container, String name)
    throws NameDuplicationException, IllegalArgumentException {
        super(container, name);
        init = new Parameter(this, "init");
        init.setExpression("0");
        step = new PortParameter(this, "step");
        step.setExpression("1");
        // set the type constraints.
        output.setTypeAtLeast(init);
        output.setTypeAtLeast(step);
    }
    public Parameter init;
    public PortParameter step;
    public void attributeChanged(Attribute attribute) throws IllegalArgumentException {
        if (attribute == init) {
            _stateToken = init.getToken();
        } else {
            super.attributeChanged(attribute);
        }
    }
    public Object clone(Workspace workspace) throws CloneNotSupportedException {
        Ramp newObject = (Ramp)super.clone(workspace);
        newObject.output.setTypeAtLeast(newObject.init);
        newObject.output.setTypeAtLeast(newObject.step);
        ...
        return newObject;
    }
}
```

Figure 1-10. Code segments from the Ramp block

To set up this arrangement, the Ramp block creates an instance of the class `PortParameter` in its constructor, as shown in figure 9a. This is a parameter that, when created, creates a coupled port. There is no need to explicitly create the port. The Ramp block creates an instance of `Parameter` to specify the *init* value, since it makes less sense for the value of *init* to be provided via a port. Referring to figure 9a, the constructor, after creating *init* and *step*, sets up type constraints. These specify that the type of the output is at least as general as the types of *init* and *step*. The `PortParameter` class takes care of an additional constraint, which is that the type of the *step* parameter must match the type of the *step* port. The `clone()` method duplicates the type constraints that are given explicitly.

In the `attributeChanged()` method, the block resets its state if *init* is the parameter that changed. This will work with an instance of `Parameter`, but it is not recommended for an instance of `PortParameter`. The reason is that each time you call `getToken()` on an instance of `PortParameter`, the method checks to see whether there is an input on the port, and consumes it if there is. Blocks are expected to consume inputs in their action methods, `fire()` and `postfire()`, not in `attributeChanged()`. Some domains, like SDF, will be confused by the consumption of the token too early. In the Ramp block in figure 9a, the `fire()` method simply outputs the current state, whereas the `postfire()` method calls `getToken()` on *step* and adds its value to the state. This follows the VisualSim convention that the state of the block is not modified in `fire()`, but only in `postfire()`.

When using a `PortParameter` in an block, care must be exercised to call `update()` exactly once per firing prior to calling `getToken()`. Each time `update()` is called, a new token will be consumed from the associated port (if the port is connected and has a token). If this is called multiple times in a iteration, it may result in consuming tokens that were intended for subsequent iterations. Thus, for example, `update()` should not be called in `fire()` and then again in `postfire()`. Moreover, in some simulators (such as DE), it is essential that if a token is provided on a port, that it is consumed. In DE, the block will be repeatedly fired until the token is consumed. Thus, it is an error to not call `update()` once per iteration.

```

Public class Ramp extends SequenceSource {
    public Ramp(CompositeEntity container, String name)
        throws NameDuplicationException, IllegalArgumentException {
        super(container, name);
        ...
        _resultArray = new Token[1];
    }
    ...
    public Object clone(Workspace workspace) throws CloneNotSupportedException {
        ...
        _resultArray = new Token[1];
        return newObject;
    }
    public int iterate(int count) throws IllegalArgumentException {
        // Check whether we need to reallocate the output token array.
        if (count > _resultArray.length) {
            _resultArray = new Token[count];
        }
        for (int i = 0; i < count; i++) {
            _resultArray[i] = _stateToken;
        }
    }
}

```



```

        step.update();
        _stateToken = _stateToken.add(step.getToken());
    }

}
output.send(0, _resultArray, count);
if (_firingCountLimit != 0) {
    _iterationCount += count;
    if (_iterationCount >= _firingCountLimit) {
        return STOP_ITERATING;
    }
}
return COMPLETED;
}
...
private Token[] _resultArray;
}

```

Figure 1-11. More code segments from the Ramp block of figure 5.10, showing the `iterate()` method.

It is important that the block call `getToken()` exactly once in either the `fire()` method or in the `postfire()` method. In particular, it should not call it in both, because this could result in consumption of two tokens from the input port, inappropriately. Moreover, it should always call it, even if it has no use for the value. Otherwise, in the DE domain, the block will be repeatedly fired if an input event is provided on the port but not consumed. Time cannot advance until that event is processed. The way that the `PortParameter` class works is as follows. On each call to `getToken()`, the `step` instance of `PortParameter` first checks to see whether an input has arrived at the associated `step` port since the last `setExpression()` or `setToken()`, and if so, returns a token read from that port. Also, any call to `get()` on the associated port will result in the value of this parameter being updated, although normally an block writer will not call `get()` on the port.

1.5 Iterate Method

Some simulators (such as SDF) will always invoke `prefire()`, `fire()`, and `postfire()` in sequence, one after another, so there is no benefit from having their functionality separated into three methods. Moreover, in SDF this sequence of method invocations may be repeated a large number of times. An block designer can improve execution efficiency by providing an `iterate()` method. Figure 9b shows the `iterate()` method of the Ramp block. Its behavior is equivalent to invoking `prefire()`, and that returns true, then invoking `fire()` and `postfire()` in sequence. Moreover the `iterate()` method takes an integer argument, which specifies how many times this sequence of operations should be repeated. The return values are `NOT_READY`, `STOP_ITERATING`, or `COMPLETED`, which are constants defined in the `Executable` interface of the block package. Returning `NOT_READY` is appropriate when `prefire()` would have returned false. Returning `STOP_ITERATING` is appropriate when `postfire()` would have returned false. Otherwise, the proper return value is `COMPLETED`.

1.6 Time

A block whose behavior depends on current model time should implement the `TimedActor` interface. This is a marker interface (with no methods). Implementing this

interface alerts the simulator that the block depends on time. Simulators that have no meaningful notion of time can reject such blocks.

A block can access current model time with the syntax:

```
double currentTime = getDirector().getModelTime();
```

Notice that although the director has a public method `setModelTime()`, an actor should never use it. Typically, only another enclosing director will call this method.

A block can request an invocation at a future time using the `fireAt()`, `fireAtCurrentTime()`, or `fireAtRelativeTime()` method of the simulator. These methods return immediately. The `fireAt()` and `fireAtRelativeTime()` methods each take two arguments, block object and time. There is also `fireAt()` method with block object, integer ID, and double time arguments. This method can be used by blocks to generate events in the future with a unique ID. The complement to this method in `DEDirector`:

```
public void fireAt(Actor actor, int id, double time) { ... }
```

is a method to cancel an event:

```
public boolean cancelAt(Object actor, int id, double time) { ... }
```

The `cancelAt()` method will return true if event in immediate hierarchical block (if any), and top `DEDirector` are removed, else false. Typical use in a user java block would look like:

Create event

```
_director.fireAt(this, _evt_id, _TNOW.doubleValue())
```

Cancel event

```
_director.cancelAt(this, _evt_id, _event_time)
```

Where `_director` is an instance variable of the block representing immediate `DEDirector`. The `fireAtCurrentTime()` method takes only one argument, an block. The simulator is responsible for performing a iteration of the specified block at the specified time. This method can be used to get a source block started, and to keep it operating. In its `initialize()` method, it can call `fireAt()` with a zero time. Then in each invocation of `postfire()`, it calls `fireAt()` again. Notice that the call should be in `postfire()` not in `fire()` because a request for a future firing is persistent state.

Note that while `fireAt()` can safely be called by any of the blocks action methods, code which executes asynchronously from the simulator should avoid calling `fireAt()`. Examples of such code include the private thread within the `DatagramReader` block and the `serialEvent()` callback method of the `SerialComm` block. Because these process hardware events, which can occur at any time, they instead use the `fireAtCurrentTime()` method. When `fireAt()` was used (before `fireAtCurrentTime()` was written) exceptions were occasionally thrown as model time advanced just as a firing was being requested at the previous (formerly current) model time.

1.7 Icons

1.7.1 New method

An actor designer can specify a custom icon when defining the actor. A (very primitive) icon editor is supplied with VisualSim. To create an icon, in the File menu, select New and then Icon Editor. An editor opens that contains a gray box for reference that is the size of the default icon that will be supplied if you do not create a custom icon. To create a custom icon, drag in the visual elements from the library, set their parameters, and then save the icon in an XML file in the same directory with the actor definition. If the name of the actor class is Foo, then the name of the file should be Foolcon.xml. That is, simply append "Icon" to the class name and complete the file name with the extension ".xml". One useful feature that is not immediately evident in the user interface is that when you specify a color to fill a geometric shape or to serve as the outline for the shape, you can make the color translucent. To do that, first choose the color using the color chooser that is made available by the parameter editing dialog for the geometric shape, then note that the color gets specified as a four-tuple of real numbers that range from 0.0 to 1.0. The fourth of these numbers is the *alpha channel* for the color, which specifies transparency. A value of 1.0 makes the color opaque. A value of 0.0 makes the color completely transparent (no color will be visible, and the background will show through). For convenience, you can specify the color to be "none" in which case a fully transparent color is supplied.

1.7.2 Old Method

An block designer can specify a custom icon when defining the block. The Ramp block, for instance, specifies the icon shown in 10

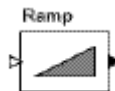


Figure 1-12 the Ramp icon.

with the following text:

```
<svg>
<rect x="-30" y="-20" width="60" height="40" style="fill:white"/>
<polygon points="-20,10 20,-10 20,10" style="fill:grey"/>
</svg>
```

This is XML, using the schema SVG (scalable vector graphics). The SVG elements that are supported are shown in figure 11. The positions in SVG are given by real numbers, where the values are increasing to the right and down from the origin, which is the nominal center of the figure. The Ramp icon contains a white rectangle and a polygon that forms a triangle.

Most of the elements in figure 11 support style attributes, as summarized in the table. A style attribute has value *keyword:value*. It can also have multiple *keyword:value* pairs, separated by semicolons.

For example, the keywords currently supported by the *rect* element are "fill", "stroke" and "stroke-width". The "fill" gives the color of the body of the figure (for figures for which this makes sense), while the "stroke" gives the color of the outline. The supported colors are black, blue, cyan, darkgray, gray, green, lightgray, magenta, orange, pink, red, white, and yellow, plus any color supported by the Java Color class `getColor()` method. The

“stroke-width” is a real number giving the thickness of the outline line, where the default is 1.0. Images are very slow to load. It is not recommended.

```
public class Ramp extends SequenceSource {  
    public Ramp(CompositeEntity container, String name)  
        throws NameDuplicationException, IllegalArgumentException {  
        super(container, name);  
        ...  
        _attachText("_iconDescription", "<svg>\n"  
            + "<rect x=\"-30\" y=\"-20\" "  
            + "width=\"60\" height=\"40\" "  
            + "style=\"fill:white\"/>\n"  
            + "<polygon points=\"-20,10 20,-10 20,10\" "  
            + "style=\"fill:grey\"/>\n"  
            + "</svg>\n");  
    }  
    ...  
}
```

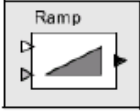


Figure 1-13 The Ramp actor defines a custom icon as shown.

1.8 Code Format

VisualSim software follows fairly rigorous conventions for code formatting. Although many of these conventions are arbitrary, the resulting consistency makes reading the code much easier, once you get used to the conventions.

A template that corresponds to these rules can be found in \$(VS)/doc/templates. There are also templates for other common files.

SVG element	Attributes
<i>rect</i>	<i>x: horizontal position of the upper left corner y: vertical position of the upper left corner width: the width of the rectangle height: the height of the rectangle style: fill, stroke, stroke-width</i>
<i>circle</i>	<i>cx: horizontal position of the center of the circle cy: vertical position of the center of the circle r: radius of the circle style: fill, stroke, stroke-width</i>
<i>ellipse</i>	<i>cx: horizontal position of the center of the ellipse cy: vertical position of the center of the ellipse rx: horizontal radius of the ellipse ry: vertical radius of the ellipse style: fill, stroke, stroke-width</i>
<i>line</i>	<i>x1: horizontal position of the start of the line y1: vertical position of the start of the line x2: horizontal position of the end of the line y2: vertical position of the end of the line style: stroke, stroke-width</i>
<i>polyline</i>	<i>points: List of x,y pairs of points, vertices of line segments, delimited by commas or spaces style: stroke, stroke-width</i>
<i>polygon</i>	<i>points: List of x,y pairs of points, vertices of the polygon, delimited by commas or spaces style: fill, stroke, stroke-width</i>
<i>text</i>	<i>x: horizontal position of the text y: vertical position of the text style: font-family, font-size, fill</i>
<i>image</i>	<i>x: horizontal position of the image y: vertical position of the image width: the width of the image height: the height of the image xlink:href: A URL for the image</i>

Figure 1-14 SVG subset currently supported by Diva, useful for creating custom icons.

1.8.1 Indentation

Nested statements should be indented 4 characters, as in:

```

if (container != null) {
    Manager manager = container.getManager();
    if (manager != null) {
        manager.requestChange(change);
    }
}

```

Closing brackets should be on a line by themselves, aligned with the beginning of the line that contains the open bracket. Tabs are 8 space characters, not a Tab character. The reason for this is that code becomes unreadable when the Tab character is

interpreted differently by different programs. Do not override this in your text editor. Long lines should be broken up into many small lines. The easiest places to break long lines are usually just before operators, with the operator appearing on the next line. Long strings can be broken up using the + operator in Java, with the + starting the next line. Continuation lines are indented by 8 characters, as in the throws clause of the constructor in figure 1.

1.8.2 Spaces

Use a space after each comma:

Right: `foo(a, b);`

Wrong: `foo(a,b);`

Use spaces around operators such as plus, minus, multiply, divide or equals signs, and after semicolons:

Right: `a = b + 1;`

Wrong: `a=b+1;`

Right: `for(i = 0; i < 10; i += 2)`

Wrong: `for(i=0 ;i<10;i+=2)`

1.8.3 Comments

Comments should be complete sentences and complete thoughts, capitalized at the beginning and with a period at the end. Spelling and grammar should be correct. Comments should include honest information about the limitations of the object definition.

Comments for base class methods that are intended to be overridden should include information about what the method does, along with a description of how the base class implements it.

Comments in derived classes for methods that override the base class should copy the general description from the base class, and then document the particular implementation. In general comments with FIXME's and implementation details should be used liberally in the code, but never in the interface description. (The interface description is the sum of all the Javadoc comments. These are the comments that will be visible in ModelBuilder via the Get Documentation right-click menu choice.) If something is important to know when using the block, put it in one of the Javadoc comments. Otherwise, put the comment elsewhere.

1.8.4 Names

In general, the names of classes, methods and members should consist of complete words separated using internal capitalization. Class names and only class names have their first letter capitalized, as in AtomicActor. Method and member names are not capitalized, except at internal word boundaries, as in getContainer(). Protected or private members and methods are preceded by a leading underscore “_” as in _protectedMethod().

Static final constants should be in uppercase, with words separated by underscores, as in INFINITE_CAPACITY. A leading underscore should be used if the constant is protected or private.

Package names should be short and not capitalized, as in “de” for the discrete-event simulator.

In Java, there is no limit to name sizes. Do not hesitate to use long names.

1.8.5 Exceptions

A number of exceptions are provided in the `VisualSim.kernel.util` package. Use these exceptions when possible because they provide convenient arguments of type `Nameable` that identify the source of the exception by name in a consistent way.

A key decision you need to make is whether to use a compile-time exception or a run-time exception.

A run-time exception is one that implements the `RuntimeException` interface. Run-time exceptions are more convenient in that they do not need to be explicitly declared by methods that throw them. However, this can have the effect of masking problems in the code.

The convention we follow is that a run-time exception is acceptable only if the cause of the exception can be tested for prior to calling the method. This is called a *testable precondition*. For example, if a particular method will fail if the argument is negative, and this fact is documented, then the method will throw a run-time exception if the argument is negative. On the other hand, consider a method that takes a string argument and evaluates it as an expression. The expression may be malformed, in which case an exception will be thrown. Can this be a run-time exception? No, because to determine whether the expression is malformed, you really need to invoke the evaluator. Making this a compile-time exception forces the caller to explicitly deal with the exception, or to declare that it too throws the same exception. In general, we prefer to use compile-time exceptions wherever possible.

When throwing an exception, the detail message should be a complete sentence that includes a string that fully describes what caused the exception. For example,

```
throw IllegalArgumentException(this,
    "Cannot append an object of type: "
    + obj.getClass().getName() + "because "
    + "it does not implement Cloneable.");
```

Note that the exception not only gives a way to identify the objects that caused the exception, but also why the exception occurred. There is no need to include in the message an identification of the “this” object passed as the first argument to the exception constructor. That object will be identified when the exception is reported to the user.

1.8.6 Javadoc

Javadoc is a program distributed with Java that generates HTML documentation files from Java source code files. Javadoc comments begin with “`/**`” and end with “`*/`”. The comment immediately preceding a method, member, or class documents that member, method, or class. `VisualSim` classes include Javadoc documentation for all classes and all public and protected members and methods. Pay special attention to the first sentence of each method comment. This first sentence is all that will describe the method in the Javadocs. Private members and methods need not be documented. Documentation can include embedded HTML formatting. For example, by convention, in block documentation, we set in italics the names of the ports and parameters using the syntax

```
/** In this block, inputs are read from the <i>input</i> port ... */
```

By convention, method names are set in the default font, but followed by empty parentheses, as in

```
/** The fire() method is called when ... */
```

The parentheses are empty even if the method takes arguments. The arguments are not shown. If the method is overloaded (has several versions with different argument sets), then the text of the documentation needs to distinguish which version is being used. It is common in the Java community to use the following style for documenting methods:

```
/** Sets the expression of this variable.  
 * @param expression The expression for this variable.  
 */  
public void setExpression(String expression) {  
    ...  
}
```

We use instead the imperative tense, as in

```
/** Set the expression of this variable.  
 * @param expression The expression for this variable.  
 */  
public void setExpression(String expression) {  
    ...  
}
```

The reason we do this is that our sentence is a well-formed, grammatical English sentence, while the usual convention is not (it is missing the subject). Moreover, calling a method is a command “do this,” so it seems reasonable that the documentation say, “Do this.” The use of imperative tense has a large impact on how interfaces are documented, especially when using the Listener design pattern. For instance, the `java.awt.event.ItemListener` interface has the method:

```
/**  
 * Invoked when an item has been selected or deselected.  
 * The code written for this method performs the operations  
 * that need to occur when an item is selected (or deselected).  
 */  
void itemStateChanged(ItemEvent e);
```

A naive attempt to rewrite this in imperative tense might result in:

```
/**  
 * Notify this object that an item has been selected or deselected.  
 */  
void itemStateChanged(ItemEvent e);
```

However, this sentence does not capture what the method does. The method may be called *in order to* notify the listener, but the *listener* does not “notify this object”. The correct way to concisely document this method in imperative tense (and with meaningful names) is:

```
/**  
 * React to the selection or deselection of an item.  
 */  
void itemStateChanged(ItemEvent event);
```


The annotation for the arguments (the @param statement) is not a complete sentence, since it is usually presented in tabular format. However, we do capitalize it and end it with a period.

Exceptions that are thrown by a method need to be identified in the Javadoc comment. An @exception tag should read like this:

```
@exception MyException If such and such occurs.
```

Notice that the body always starts with "If", not "Thrown if", or anything else. Just look at the Javadoc output to see why this occurs. In the case of an interface or base class that does not throw the exception, use the following:

```
* @exception MyException Not thrown in this base class. Derived  
* classes may throw it if such and such happens.
```

The exception still has to be declared so that derived classes can throw it, so it needs to be documented as well.

The Javadoc program gives extensive diagnostics when run on a source file. Our policy is to format the comments until there are no Javadoc warnings.

1.8.7 Code Organization

The basic file structure that we use follows the outline in figure 1, preceded by a one-line description of the file and a copyright notice. The key points to note about this organization are:

- The file is divided into sections with highly visible delimiters. The sections contain constructors, ports and parameters (and other public members, if there are any), public methods, protected methods, protected members, private methods, and private members, in that order. Note in particular that although it is customary in the Java community to list private members at the beginning of a class definition, we put them at the end. They are not part of the public interface, and thus should not be the first thing you see.
- Within each section, methods appear in alphabetical order, in order to easily search for a particular method. If you wish to group methods together, try to name them so that they have a common prefix.
- Static methods are generally mixed with non-static methods.

1.9 Java Block Template

The Java block template is located in <VisualSim Install directory>/doc/templates and is called JavaBlockTemplate.java. The Java block will contain the following:

1. Comment- One line definition. All comments are used by javadoc generator
2. Copyright Notice
3. Rating on the quality based on code completion and code tested
4. Package name
5. Import
6. Documentation
7. Class Definition
8. Tooltip
9. Parameters and Ports

10. Icon in svg format. If you would like to use the IconEditor to create a separate xml file, then make sure to name the xml file to BlockName+Icon.xml.
11. Public Methods
 - a. Attribute Changed
 - b. Preinitialize
 - c. Initialize
 - d. Prefire
 - e. Fire
 - f. Post File
 - g. Wrapup
12. Common functions such as adding an event, and rune dependencies
13. Variables definition

1.10 Debugging

The following assumes you know how to develop simple standard Java programs and will focus on how to debug Java applications in Eclipse.

1.9.1 Eclipse Debugger Setup

Create a new Java Project, to create a new java project click on

File → New → Java Project

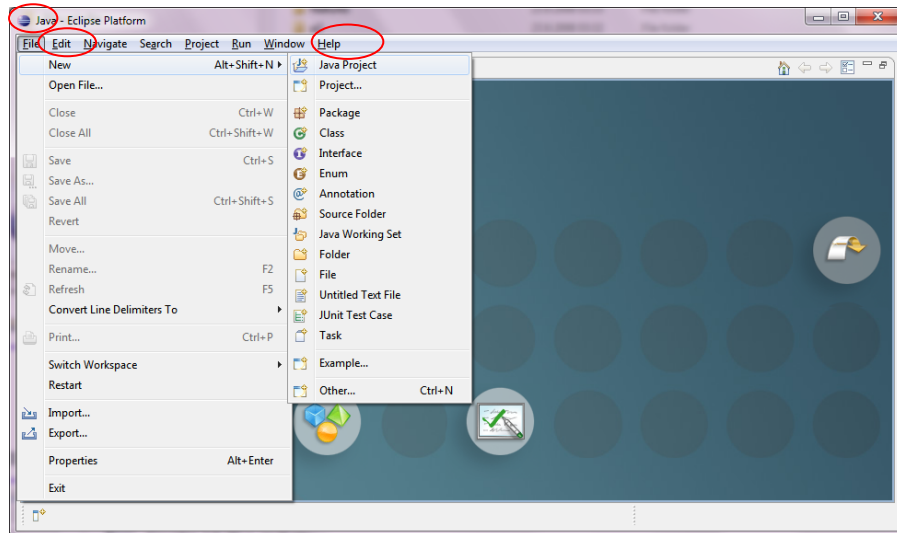


Figure 1.9.1

Set the project name and JDK as shown in the figure 1.9.2

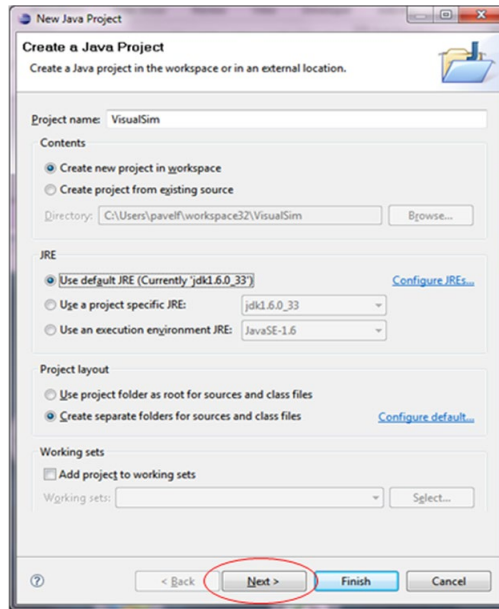


Figure 1.9.2

Add all the required external libraries, including any Apache and Java libraries on “Libraries” tab as shown in the figure 1.9.3

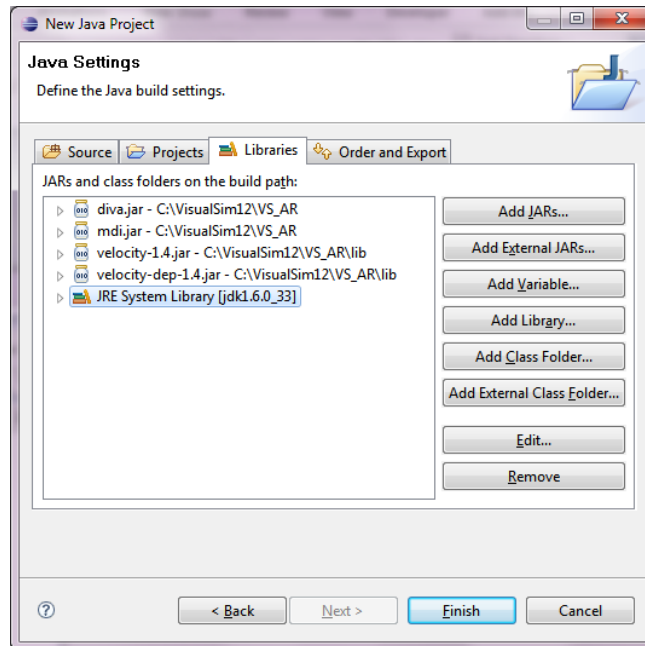


Figure 1.9.3

Using “Order and Export” tab on Eclipse debugger set the proper order of libraries. Source files should be placed on top.

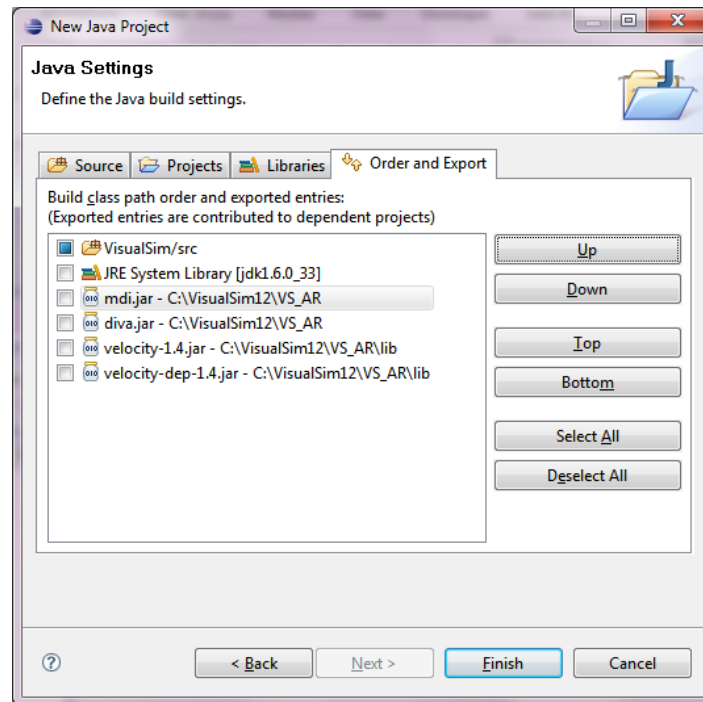


Figure 1.9.4

Update the VisualSim Start Script with below commands.

1. Add compiled classes to the class path
`set CLASSPATH = <path to compiled classes>%CLASSPATH%`
2. Prepare Java debug settings
`set dbg = -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdp:transport=dt_socket,server=y,suspend=n, address=<debug port>`
3. Modify java command
`java %dbg% ... VisualSim.ModelBuilder.ModelBuilderApplication`

On Eclipse Java Debugger, Provide full path to compiled classes in “Default Output Folder” as shown in figure 1.9.5.

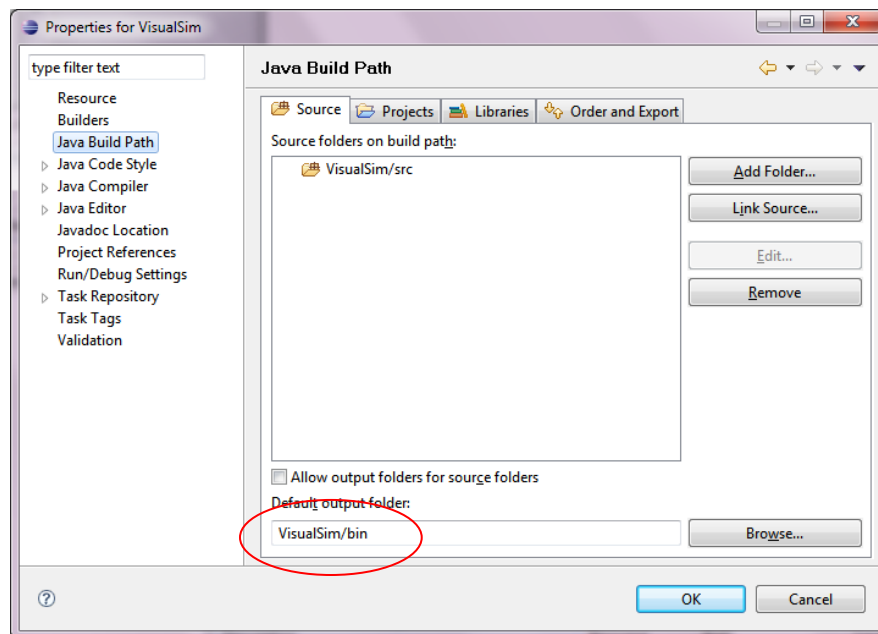


Figure 1.9.5

Click on “Ok” button and run VisualSim using VisualSim start script.

1.9.2 Setup Debug Configuration

On Eclipse Debug platform, select “Debug Configurations” (Run → Debug Configurations)

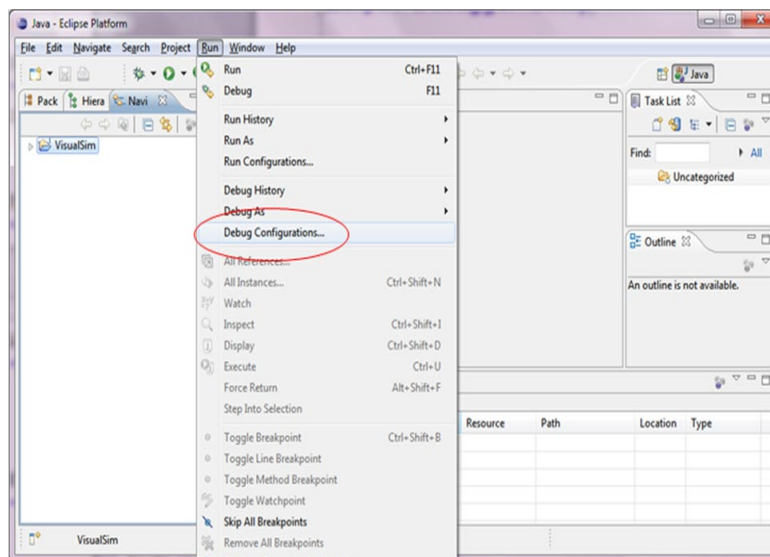


Figure 1.9.6

Choose “Remote Java Application”. You can use default settings. Port should correspond to <debug port> in Java debug options.

Click “Debug”

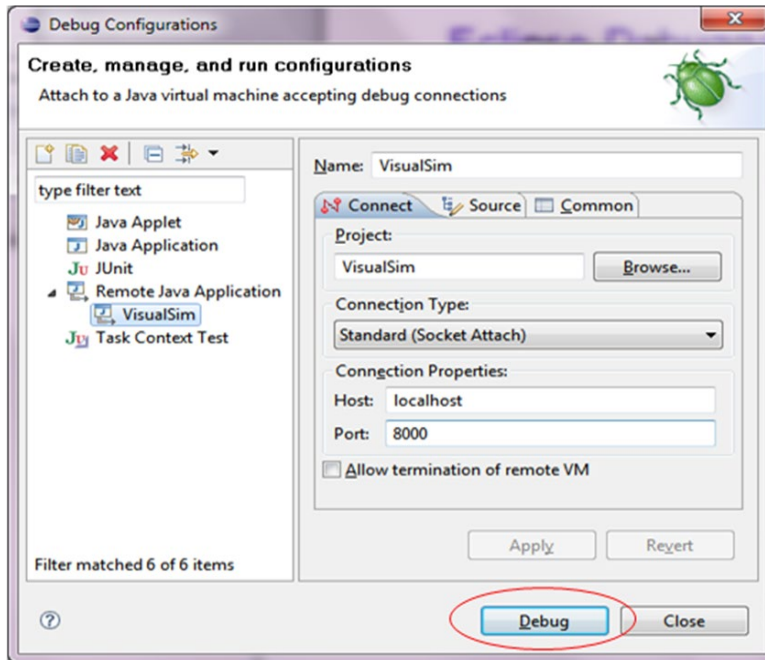


Figure 1.9.7

Now you are ready to debug your code!

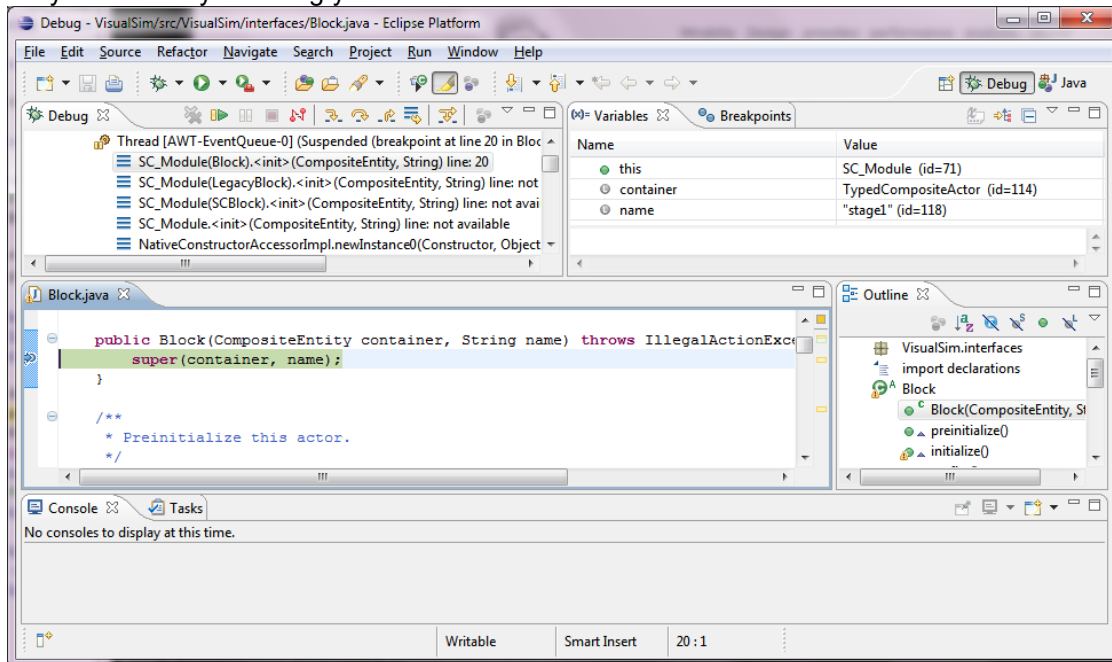


Figure 1.9.8

2 Adding a Block to ModelBuilder Library List

Below are instructions for adding a block to VisualSim and making it visible in ModelBuilder. For this example, we are going to take the Ramp block and change the default step from 1 to 2. The new block will be called Ramp2.

Below are the steps necessary to add a block:

1. Create the new .java file that implements your block:

In this case, we are just copying a Ramp.java to Ramp2.java

```
cd "$VS/VisualSim/actor/lib"  
cp -p Ramp.java $VS/User_Library/lib/Ramp2.java
```

We have copied the java code from a *different directory* and we would have to change the package statement (usually near line 31) in the java code. This is good to keep in mind since there is no error message to clue you in to this particular error.

2.1 Edit Ramp2.java and change:

- 1)

```
package VisualSim.actor.lib  
    ■ to  
package User_Library.lib
```
- 2)

```
public class Ramp extends SequenceSource {  
    ■ to  
public class Ramp2 extends SequenceSource {
```
- 3)

```
public Ramp(CompositeEntity container, String name)  
throws NameDuplicationException, IllegalArgumentException {  
    ■ to  
public Ramp2(CompositeEntity container, String name)  
throws NameDuplicationException, IllegalArgumentException {
```
- 4)

```
step = new Parameter(this, "step", new IntToken(1));  
    ■ to  
step = new Parameter(this, "step", new IntToken(2));
```
- 5)

```
Ramp newObject = (Ramp)super.clone(workspace);  
    ■ to  
Ramp2 newObject = (Ramp2)super.clone(workspace);
```

2.2 Library Palette Addition

After the block has been compiled the next step is to include it in the library palette of the ModelBuilder. From the Block Diagram menubar, select Instantiate Entity. Enter the complete path to the class file for the block. This will show up on the screen. Now save this block in the UserLibrary. It is now available for use directly from this library.

2.3 Testing Addition

Start up ModelBuilder.

In ModelBuilder, click on File->New->Graph Editor

In the Graph Editor window, click on "User Library". The Ramp2 block will appear.

To test the Ramp2 block:

- Drag the Ramp2 block over to the main canvas on the right
- Click on Basic library -> Display and drag the Display block over to the main canvas
- Connect the two blocks by left clicking on the output of the ramp2 block and dragging over to the input of the Display block
- Select simulator library -> SDF and drag the SDF simulator over to the right window
- Select View -> Simulation Cockpit and set the iteration to 10, then hit the GO button.
- You should see the numbers from 0 to 18 in the display.
- The Ramp2 has been added properly.

2.4 Adding a new palette

The palette on the left side of the Graph editor lists the utilities, simulators and blocks available for use in ModelBuilder.

To add a new set of blocks, we first create a *.xml file that lists the block. In this case, the file is called \$VS/User_Library/lib/myblock.xml, and it contains one block, Ramp2:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE plot PUBLIC "-//Mirabilis Design//DTD MoML 1//EN"
"http://www.mirabilisdesign.com/xml/dtd/MoML_1.dtd">
<entity name="myblocks" class="VisualSim.moml.EntityLibrary">
  <configure>
    <?moml
      <group>
        <doc>My Blocks</doc>
        <entity name="Ramp2" class="VisualSim.actor.lib.Ramp2">
          <doc>Create a sequence of tokens with increasing value of 2</doc>
        </entity>
      </group>
    ?>
  </configure>
</entity>
```

We want to add our new palettemyblock.xml, to the block library palette so we will add myblock.xml to \$VS/User_Library/lib/main.xml. Note that we want our new palette to be a sub pallet of the User Library palette, just as the "Display and Probe" palette is. Sub-palettes are caused by the entity statement in the 4th line of the main.xml file.

To add additional libraries, create a new entity library. This can be used to manage blocks from different sources such as users, projects or location. The following is a simple mechanism to add a new library.

In User_Library/lib/main.xml we add

```
<doc>User Library</doc>
<input source="User_Library/lib/myblock.xml"/>
```

Now restart ModelBuilder, and the "myblocks" sub-palette will appear under "User Library".

2.5 Hints on Adding Blocks

The VisualSim MoML definition provides some nifty features to enhance the reuse of library blocks.

- ◆ Display the value of a key parameter in the center of the icon for a particular block
- ◆ Make the values of a parameter in the block pull-down as opposed to user created

These two capabilities are handled in the XML file that adds the block to the palette.

2.5.1 Icon Display

In `$VS/User_Library/lib`, there is a file `main.XML` for adding user-defined blocks to the palette. The information for the “Const1” block is provided in it. The following lines define the properties of the icon:

```
<property name="_icon" class="VisualSim.ModelBuilder.icon.BoxedValueIcon">
  <property name="attributeName" value="value"/>
  <property name="displayWidth" value="40"/>
</property>
```

The first line defines the type of information to be displayed in the block Icon and the class can be of two types- `VisualSim.ModelBuilder.icon.AttributeValueIcon` or `VisualSim.ModelBuilder.icon.BoxedValueIcon`. The boxed value refers to user input while attribute value is from a pull-down. Look at the entry for “Queue One” in `$VS/VisualSim/actor/lib/perf/perf.XML`.

The display value is the information selected in the attribute name “value”. The property definition starts with the `<property name="" class="">` and ends with `</property>`. There can be multiple property lines within this single definition and each will start with `<property name="" value="">`.

2.5.2 Parameter Pull-Down

Attributes of the blocks can be pull-down or user-entered information. To create a pull-down, the following property needs to be entered in the entity definition of the block. This information is extracted from the Queue One definition in `$VS/VisualSim/actor/lib/perf/perf.XML`.

```
<property name="Initial_Queue_State">
  <property name="style" class="VisualSim.actor.gui.style.ChoiceStyle">
    <property name="First-Token-Flow-Through"
      value="First-Token-Flow-Through"
      class="VisualSim.kernel.util.StringAttribute"/>
    <property name="First-Token-Enqueue"
      value="First-Token-Enqueue"
      class="VisualSim.kernel.util.StringAttribute"/>
  </property>
</property>
```

The pull-down options are the property names embedded within the “Initial Queue State” definition above. The property definition starts with the `<property name="" class="">` and ends with `</property>`. There can be multiple property lines within this single definition and each will start with `<property name="" value="" class="VisualSim.kernel.util.StringAttribute">`. If the attribute requires a user-entered value, then the attribute does not have to be defined in the .XML file. Note: A particular block can have multiple properties in the .XML file.

3 Creating Applets

3.1 Introduction

VisualSim models can be embedded in applets. In most cases, the MoMLApplet class can be used. For the MoMLApplet class, the model is given by a MoML file, which can be created using ModelBuilder. The URL for the MoML file is given by the modelURL applet parameter in the HTML file.

Occasionally, however, it is useful to create an applet that exercises more control over the display and user interaction, or constructs or manipulates VisualSim models in ways that cannot be done in MoML. In such cases, the VisualSimApplet class can be useful. Developers may either use VisualSimApplet directly or extend it to provide a more sophisticated user interface or a more elaborate method for model construction or manipulation.

The VisualSimApplet class provides four applet parameters:

- *background*: The background color, typically given as a hex number of the form "#rrggbb" where rr gives the red component, gg gives the green component, and bb gives the blue component.
- *controls*: This gives a comma-separated list of any subset of the words "buttons", "topParameters", and "directorParameters" (case insensitive), or the word "none". If this parameter is not given, then it is equivalent to giving "buttons", and only the control buttons mentioned above will be displayed. If the parameter is given, and its value is "none", then no controls are placed on the screen. If the word "topParameters" is included in the comma-separated list, then controls for the top-level parameters of the model are placed on the screen, below the buttons. If the word "directorParameters" is included, then controls for the director parameters are also included.
- *modelClass*: The fully qualified class name of a Java class that extends NamedObj. This class defines the model.
- *orientation*: This can have value "horizontal", "vertical", or "controls_only" (case insensitive). If it is "vertical", then the controls are placed above the visual elements of the Placeable blocks. This is the default. If it is "horizontal", then the controls are placed to the left of the visual elements. If it is "controls_only" then no visual elements are placed.

The use of these applet parameters is explained in more detail below. For a visual description of the usage of these models as Applets, visit the [Applet Usage Page](#).

3.1.1 HTML Files Containing Applets

An applet is a Java class that can be referenced by an HTML file and accessed either locally or over the web and run in a secure manner on the local machine in a web browser. Unfortunately, many browsers available today are shipped with an earlier version of Java that does not provide features that VisualSim requires. The work around is to use Sun's Java Plug-In, which invokes the 1.4.1_01 version of the Java Runtime Environment (JRE), instead of the default Java runtime that is shipped with the browser. The Java Plug-in is installed when the JRE or the Java Development Kit (JDK) is installed. Unfortunately, using the Java Plug-in makes the applet HTML more complex. There are two choices:

1. Use fairly complex JavaScript to determine which browser is running and then to properly select one of three different ways to invoke the Java Plug-in. This method works on the most different types of platforms and browsers. The JavaScript is so complex, that rather than reproduce it here, please see one of the demonstration html files such as \$VS/ demos/signal/Butterfly/Butterfly.htm. Sun provides a free tool called HTMLConverter that will automatically generate the html code, see the Java Plug-in home page at <http://java.sun.com/products/plugin/>.
2. Use the much simpler <applet> ...</ applet> tag to invoke the Java Plug-in. This method works on many platforms and browsers, but requires a more recent version of the Java Plug-in, and will not work under Netscape Communicator 4. x. However, all is not lost for

Netscape Communicator 4.x users, since the **appletviewer** command that is included with the Java Development kit will display applets written using the simpler format. For details about the above two choices, see [http:// java.sun.com/ products/ plugin/ versions.html](http://java.sun.com/products/plugin/versions.html). Sample HTML for the `<applet> . . . </ applet>` style of custom applet is shown in Table 1. An HTML file containing the segment shown in Table 1 can be found in `$VS/doc/tutorial/TutorialApplet1.htm`, where `$VS` is the home directory of the VisualSim installation. Also in that directory are a number of sample Java files for applets, each named `TutorialAppletn.java`, where `n` is an integer starting with 1. These files contain a series of applet definitions, each with increasing sophistication that is discussed below. Each applet has a corresponding `TutorialAppletn.htm` file.

Since our example applets are in a directory `$VS/doc/tutorial`, the codebase for the applet is `"../.."` in Table 1 which is the directory `$VS`. This permits the applets to refer to any class in the VisualSim tree. There are some parameters in the HTML in Table 1 that you may want to change. The width and the height, for example, specify the amount of space on the screen that the browser gives to the applet.

VisualSim provides a unique mechanism to embed VisualSim models in documentation. These models can be simulated within the document environment. The parameters can be modified and re-simulated. To enable this function, users must have access to a license of the VisualSim Explorer. The physical documents must have access to the install and must be provided a relative pointer or URL address.

The document can be created in any preferred Editor including Frame maker, Word and WordPerfect. Once the document has been created the file must be converted to a HTML document. This document must be edited in a HTML editor such as Netscape Navigator and FreeHTML. The specific location of the VisualSim model must be identified. The XML file must be on a remote location where the software package is installed. It cannot be placed on the local machine. If the model development is an ongoing process, then provide the relative path to the library directory/filename.xml.

There are three types of models that can be embedded in the document:

- ◆ View of the VisualSim model with access to that layer parameters –Figure 12
- ◆ View of the VisualSim model simulation cockpit window. This includes the top-level parameters and the analysis windows. This view can be simulated and analyzed. -Figure 13
- ◆ Combined view of the above two.- Figure 14

The Java script to embed these three variations is provided. The details of the various terms in this document are provided in the next section- Creating Applets.

3.2 Java Script for embedding the VisualSim Model

The following are the steps to embed the VisualSim models in your HTML document:

1. Identify the location in the document to place the model.
2. Now, copy the appropriate code from below.
3. Modify the following lines to reflect the install of VisualSim Analyzer Server and the models:

- ◆ `<PARAM NAME = "codebase" VALUE = "." > \`

Change the above `"."` to the relative location of the VisualSim Analyzer install.

- ◆ `<PARAM NAME = "NAME" VALUE = "Dual Processor Server" > \`

Replace the Dual Processor Server with a suitable name for your model. This will become the title of the model.

- ◆ `<PARAM NAME = "modelURL" VALUE = "Dual_Processor_MUX1.xml" > \`

Name of the model file to be included. Remember to include the `.xml` extension for the filename.

- ◆ `MoML specification for this model. \`

Name of the model file to be included. Remember to include the `.xml` extension for the filename.

- ◆ `NAME = "Dual Processor Server" background = "#faf0e6" modelURL = "Dual_Processor_MUX1.xml" \`

Modify the name of the model and the filename.

- ◆ `<PARAM NAME = "model" VALUE = "Dual_Processor_MUX1.xml" > \`

This is the name of the model file to be included. Remember to include the .xml extension for the filename.

- ◆ `width = "600" \`
`height = "650" \`

Change the height and width of the model to make sure the entire model is visible in the window. ON the other hand you can also adjust the size of the model within ModelBuilder.

The model should be visible and execute within the document.

```
<script language="JavaScript">
<!--
  var _info = navigator.userAgent;
  var _ns = false;
  var _ns6 = false;
  var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0 && _info.indexOf("Windows 3.1") < 0);
  //-->
</script><script language="JavaScript">
<!--
  if (_ie == true) document.writeln("\
    <OBJECT \
      classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" \
      width = "800" \
      height = "600" \
      <codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-
i586.cab#Version=1,4,1,mn"> \
      <PARAM NAME = "code" \
        VALUE = "VisualSim.ModelBuilder.MoMLViewerApplet" > \
      <PARAM NAME = "codebase" VALUE = ".." > \
      <PARAM NAME = "archive" \
        VALUE = "VSupport.jar,diva.jar,ModelBuilderApplet.jar,diva.jar,simulators.jar" > \
      <PARAM NAME = "type" \
        VALUE = "application/x-java-applet;jpi-version=1.4.1_01" > \
      <PARAM NAME = "scriptable" VALUE = "false" > \
      <PARAM NAME = "NAME" VALUE = "Dual Processor Server" > \
      <PARAM NAME = "background" VALUE = "#faf0e6" > \
      <PARAM NAME = "modelURL" VALUE = "Dual_Processor_MUX1.xml" > \
      <a href="Dual_Processor_MUX1.xml">MoML specification for this model.</a> \
    </OBJECT> \
  ');
  else if (_ns == true && _ns6 == false)
    document.writeln("\
    <COMMENT> \
    <EMBED \
      type = "application/x-java-applet;jpi-version=1.4.1_01" \
      code = "VisualSim.ModelBuilder.MoMLViewerApplet" \
      codebase = ".." \
      archive = "VSupport.jar,diva.jar,ModelBuilderApplet.jar,diva.jar,simulators.jar" \
      width = "8" \
      height = "600" \
      NAME = "Dual Processor Server" background = "#faf0e6" modelURL = "Dual_Processor_MUX1.xml" \
    \
      scriptable = "false" \
      pluginspage = "http://java.sun.com/j2se/1.4.1/download.html"> \
    <NOEMBED> \
    <a href="Dual_Processor_MUX1.xml">MoML specification for this model.</a> \
    </NOEMBED> \
    </EMBED> \
    <COMMENT> \
```

```

');
else document.writeln("\
<APPLET \
  code = "VisualSim.ModelBuilder.MoMLViewerApplet" \
  codebase = ".." \
  archive = "VSsupport.jar,diva.jar,ModelBuilderApplet.jar,diva.jar,simulators.jar" \
  width = "800" \
  height = "600" \
  > \
  <PARAM NAME = "NAME" VALUE = "Dual Processor Server" > \
    <PARAM NAME = "background" VALUE = "#faf0e6" > \
    <PARAM NAME = "modelURL" VALUE = "Dual_Processor_MUX1.xml" > \
    <a href="Dual_Processor_MUX1.xml">MoML specification for this model.</a> \
  </APPLET> \
');
//-->
</script><!-- HTML CONVERTER -->

```

Figure 3-1 Java Script for VisualSim Model

```

<script language="JavaScript">
<!--
  var _info = navigator.userAgent;
  var _ns = false;
  var _ns6 = false;
  var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0 && _info.indexOf("Windows 3.1") < 0);
//-->
</script><script language="JavaScript">
<!--
  if (_ie == true) document.writeln("\
    <OBJECT \
      classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" \
      width = "800" \
      height = "480" \
      <codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-
i586.cab#Version=1,4,1,mn"> \
      <PARAM NAME = "code" \
        VALUE = "VisualSim.actor.gui.MoMLApplet" > \
      <PARAM NAME = "codebase" VALUE = ".." > \
      <PARAM NAME = "archive" \
        VALUE = "VSsupport.jar,simulators.jar" > \
      <PARAM NAME = "type" \
        VALUE = "application/x-java-applet;jpi-version=1.4.1" > \
      <PARAM NAME = "scriptable" VALUE = "false" > \
      <PARAM NAME = "name" VALUE = "Dual Processor Server" > \
        <PARAM NAME = "background" VALUE = "#faf0e6" > \
          <PARAM NAME = "controls" VALUE = "buttons, topParameters" > \
          <PARAM NAME = "orientation" VALUE = "horizontal" > \
          <PARAM NAME = "model" VALUE = "Dual_Processor_MUX1.xml" > \
          <a href="Dual_Processor_MUX1.xml">MoML specification for this model.</a> \
        </OBJECT> \
      ');
  else if (_ns == true && _ns6 == false)
    document.writeln("\
    <COMMENT> \
    <EMBED \
      type = "application/x-java-applet;jpi-version=1.4.1" \
      code = "VisualSim.actor.gui.MoMLApplet" \
      codebase = ".." \
      archive = "VSsupport.jar,simulators.jar" \

```

```

width = "600" \
height = "480" \
name = "Dual Processor Server" background = "#faf0e6" model = "Dual_Processor_MUX1.xml"
controls = "buttons, topParameters" orientation = "horizontal" \
scriptable = "false" \
pluginspage = "http://java.sun.com/j2se/1.4.1/download.html"> \
<NOEMBED> \
<a href="Dual_Processor_MUX1.xml">MoML specification for this model.</a> \
</NOEMBED> \
</EMBED> \
<COMMENT> \
');
else document.writeln(\
<APPLET \
code = "VisualSim.actor.gui.MoMLApplet" \
codebase = ".." \
archive = "VSsupport.jar,simulators.jar" \
width = "600" \
height = "480" \
> \
<PARAM NAME = "name" VALUE = "Dual Processor Server" > \
<PARAM NAME = "background" VALUE = "#faf0e6" > \
<PARAM NAME = "controls" VALUE = "buttons, topParameters" > \
<PARAM NAME = "orientation" VALUE = "horizontal" > \
<PARAM NAME = "model" VALUE = "Dual_Processor_MUX1.xml" > \
<a href="Dual_Processor_MUX1.xml">MoML specification for this model.</a> \
</APPLET> \
');
//-->
</script><!-- HTML CONVERTER -->

```

Figure 3-2 Java Script for embedding VisualSim Simulation Cockpit

```

<P><!-- HTML CONVERTER --><SCRIPT LANGUAGE="JavaScript"><!--
var _info = navigator.userAgent;
var _ns = false;
var _ns6 = false;
var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0 && _info.indexOf("Windows 3.1") < 0);
//--></SCRIPT><COMMENT> <SCRIPT LANGUAGE="JavaScript1.1"><!--
var _ns = (navigator.appName.indexOf("Netscape") >= 0 && ((_info.indexOf("Win") > 0 &&
_info.indexOf("Win16") < 0 && java.lang.System.getProperty("os.version").indexOf("3.5") < 0) ||
(_info.indexOf("Sun") > 0) || (_info.indexOf("Linux") > 0) || (_info.indexOf("AIX") > 0) || (_info.indexOf("OS/2")
> 0) || (_info.indexOf("IRIX") > 0)));
var _ns6 = ((_ns == true) && (_info.indexOf("Mozilla/5") >= 0));
//--></SCRIPT></COMMENT> <SCRIPT LANGUAGE="JavaScript"><!--
if (_ie == true) document.writeln(\
<OBJECT \
classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" \
width = "1000" \
height = "1400" \
codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-
i586.cab#Version=1,4,1,mn"> \
<PARAM NAME = "code" \
VALUE = "VisualSim.ModelBuilder.MoMLViewerApplet" > \
<PARAM NAME = "codebase" VALUE = "../.." > \
<PARAM NAME = "archive" \
VALUE = "VSsupport.jar,ModelBuilderApplet.jar,diva.jar,simulators.jar" > \
<PARAM NAME = "type" \
VALUE = "application/x-java-applet;jpi-version=1.4.1_01" > \
<PARAM NAME = "scriptable" VALUE = "false" > \

```

```

    <PARAM NAME = "NAME" VALUE = "SNS RF Control System" > \
      <PARAM NAME = "background" VALUE = "#faf0e6" > \
      <PARAM NAME = "controls" VALUE = "buttons, topParameters" > \
      <PARAM NAME = "includeRunPanel" VALUE = "true" > \
      <PARAM NAME = "modelURL" VALUE = "Sensor_Model.xml" > \
      <a href="Sensor_Model.xml">MoML specification for this model.</a> \
    </OBJECT> \
  ');
  else if (_ns == true && _ns6 == false)
    document.writeln("\
    <COMMENT> \
    <EMBED \
      type = "application/x-java-applet;jpi-version=1.4.1_01" \
      code = "VisualSim.ModelBuilder.MoMLViewerApplet" \
      codebase = "../.." \
      archive = "VSsupport.jar, ModelBuilderApplet.jar, diva.jar, simulators.jar" \
      width = "1000" \
      height = "1400" \
      NAME = "SNS RF Control System" background = "#faf0e6" controls = "buttons, topParameters "
includeRunPanel = "true" modelURL = "Sensor_Model.xml" \
      scriptable = "false" \
      pluginspage = "http://java.sun.com/j2se/1.4.1/download.html"> \
      <NOEMBED> \
      <a href="Sensor_Model.xml">MoML specification for this model.</a> \
      </NOEMBED> \
    </EMBED> \
    <COMMENT> \
  ');
  else document.writeln("\
  <APPLET \
    code = "VisualSim.ModelBuilder.MoMLViewerApplet" \
    codebase = "../.." \
    archive = "VSsupport.jar, ModelBuilderApplet.jar, diva.jar, simulators.jar" \
    width = "1000" \
    height = "1400" \
  > \
    <PARAM NAME = "NAME" VALUE = "SNS RF Control System" > \
      <PARAM NAME = "background" VALUE = "#faf0e6" > \
      <PARAM NAME = "controls" VALUE = "buttons, topParameters" > \
      <PARAM NAME = "includeRunPanel" VALUE = "true" > \
      <PARAM NAME = "modelURL" VALUE = "Sensor_Model.xml" > \
      <a href="Sensor_Model.xml">MoML specification for this model.</a> \
    </APPLET> \
  ');
  //--></SCRIPT><!-- /HTML CONVERTER --></CENTER></P>

```

Figure 3-3 Java Script to combine the VisualSim Model and Simulation Cockpit together.

3.2.1 Using JavaScript in files

It is possible to put the javascript in a file and call it from the html file. This reduces clutter in the html document and also provides increased flexibility in adding functions. In this case the html file will contain the following lines where the model needs to be displayed:

```
<script src="video_model.js"></script>
```

The video_model.js contains the same javascript as shown above. A number of demonstration systems on the Web are currently using this method.

4 Templates

This section contains templates that can be used as a starting point for custom applets and block development:

- ◆ [Java Block](#) with code format and Javadoc inclusions
- ◆ [Text Html](#) for creating documentation with limited text
- ◆ [Applet Html](#) for creating Applets

The following contains sample code that is discussed in the Creating Custom Applet and creating models in Java in this document.

[Tutorial 1](#): This is a simple model containing 2 blocks, a simulator with the Stop Time parameter set and the connection between the blocks. This is the view of this model as an [Applet in a Web Browser](#).

[Tutorial 2](#): This is the above model but implemented as an Applet. This is the view of this model as an [Applet in a Web Browser](#).

[Tutorial 3](#): This model contains the two blocks, two top-level parameters, parameters of the block linked to the model parameters, a plotter and the connection between the blocks. This is the view of this model as an [Applet in a Web Browser](#).