.MIRABILIS DESIGN INC.

Advanced Modeling Guide

VisualSim Documentation

Mirabilis Design Inc.
2010 El Camino Real, Suite 1061
Santa Clara, CA 95050

Table of Contents

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 avail-able 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

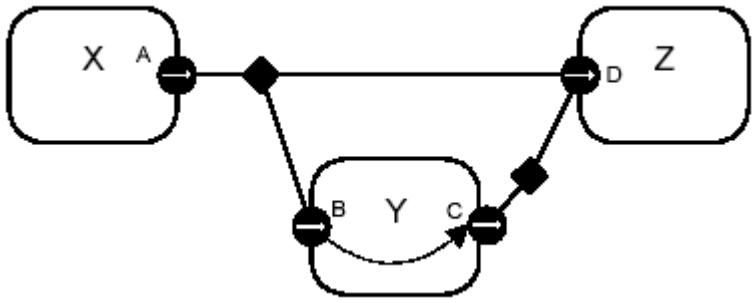


Figure 1 If there are simultaneous events at B and D, then the one at B will have higher priority because it may trigger another simultaneous event at D.

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

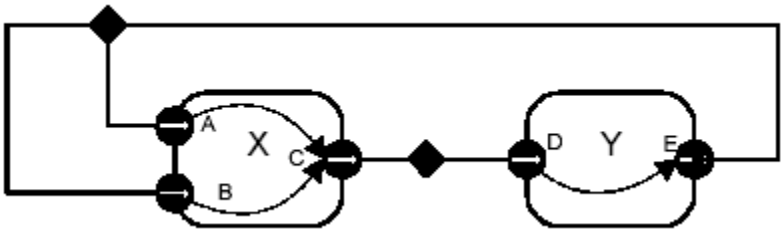


Figure 2 Example of a directed zero-delay loop.

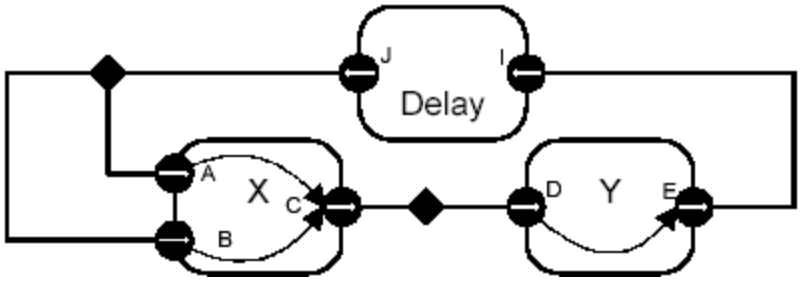


Figure 3 A Delay block can be used to break a zero-delay loop

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 IllegalActionException {
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 IllegalActionException {
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 IllegalActionException {
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 DEThreadActor {
    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 DEThreadActor 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-

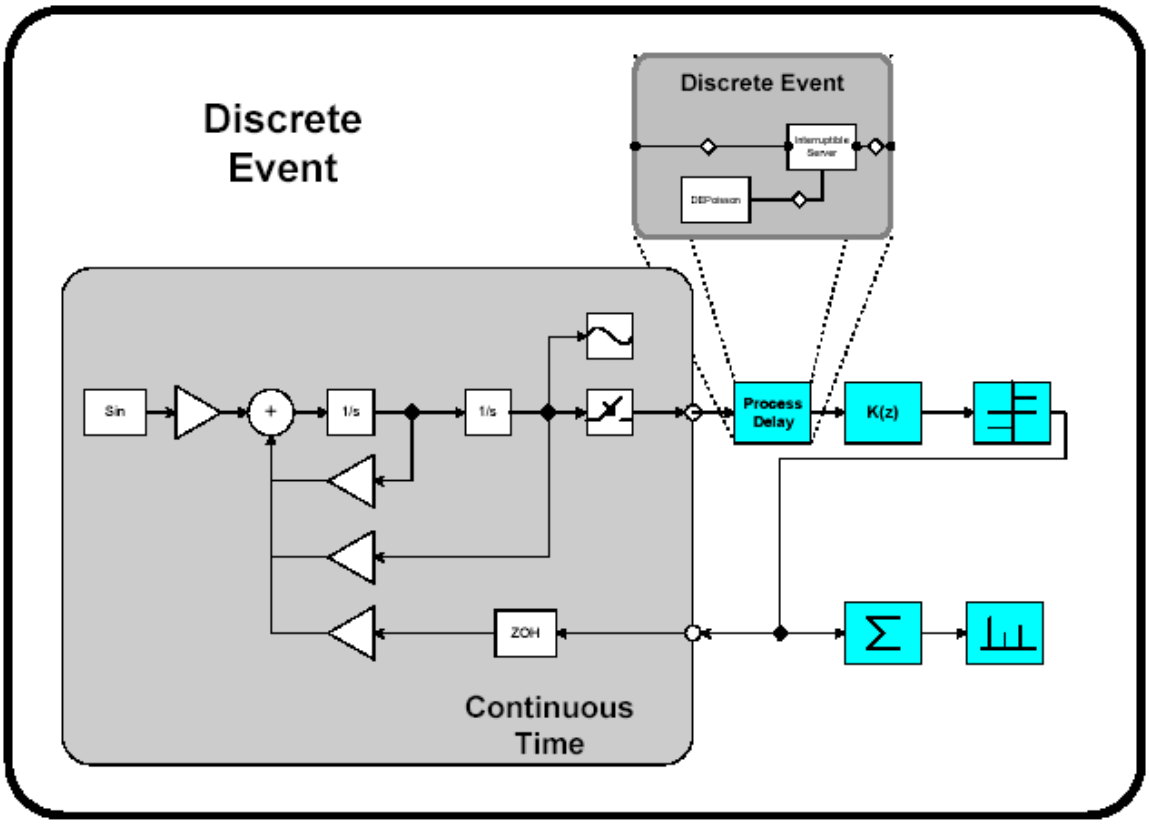


Figure 8 An example of heterogeneous and hierarchical composition. The CT subsystem and DE subsystem are inside an outermost DE system.

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.

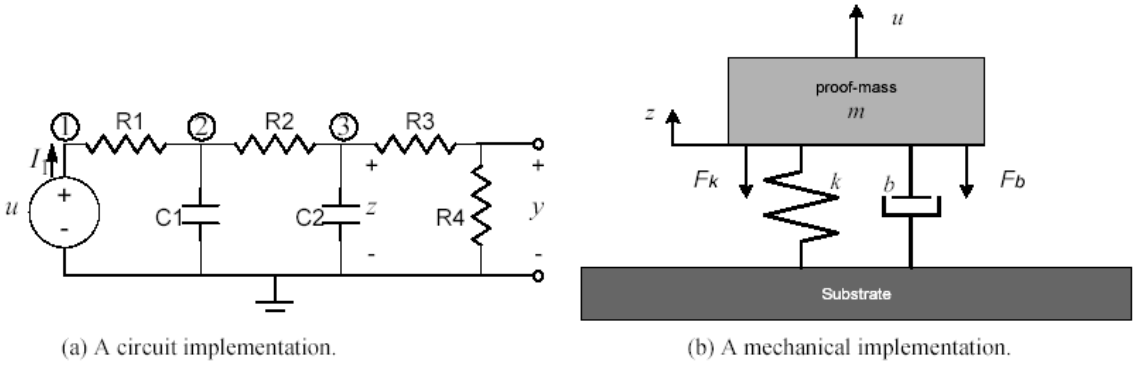
2 CT Simulator

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,

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

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



(a) A circuit implementation. (b) A mechanical implementation.

Figure 9 Possible implementations of the system equations.

of node 3, and

$$m = R1 \cdot R2 \cdot C1 \cdot C2 \tag{2}$$

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

$$b = 1$$

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

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:

$$\dot{x} = f(x, u, t) \tag{3}$$
$$y = g(x, u, t) \tag{4}$$
$$x(t_0) = x_0, \tag{5}$$

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

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

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

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

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

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

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

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

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 integrators[1]. 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 and 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.

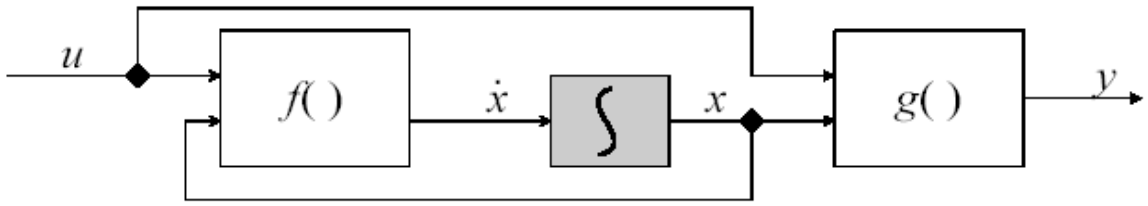


Figure 10 A conceptual block diagram for continuous time systems.

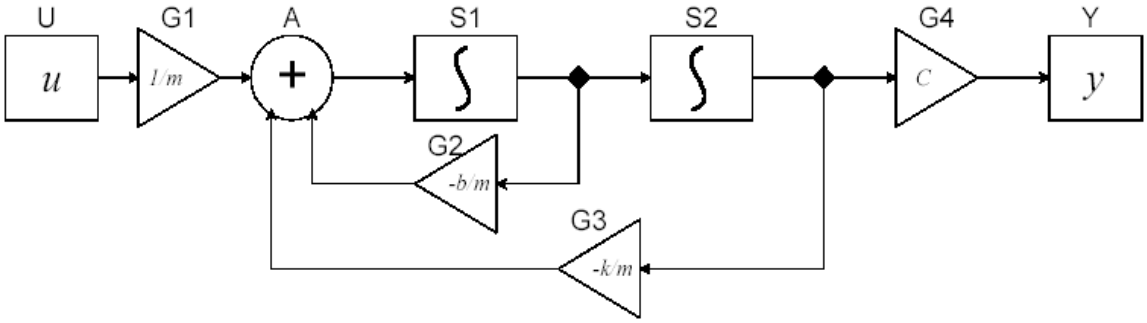


Figure 11 The block diagram for the example system.

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 $\Re$ which contains at most a finite number of points per unit interval, and let u be piecewise-continuous on $\Re - D$. Then, for any fixed u(t), if f is also piecewise-continuous on $\Re - D$, 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

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

where

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

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: he *n*-th time point, to explicitly show the discretization of time. However, *t* is specified, if the index *n* is not important.
- *X[ti, tj]*:the precise (continuous) state trajectory from time t_i to t_j;
- *X(tn)* he 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 $x_{t_0} \ldots x_{t_{n-1}}$ 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 $x_{t_0} \ldots x_{t_{n-1}}$. 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 $x_{t_0}, \ldots, x_{t_{n-1}}, x_{t_n}$, i.e.

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

Plugging (3) in this, we get

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

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:

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

or,

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

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_o be in the contraction radius, then a unique fixed point exists and can be found by iteratively computing:

$$\sigma_1 = F_E(\sigma_0),\ \sigma_2 = F_E(\sigma_1),\ \sigma_3 = F_E(\sigma_2),\ \ldots \tag{16}$$

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:

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

2. Backward Euler solver:

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

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

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

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

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

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

with error control:

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

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

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

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

4. Trapezoidal Rule solver:

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

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:

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

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

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 once[2]. 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.

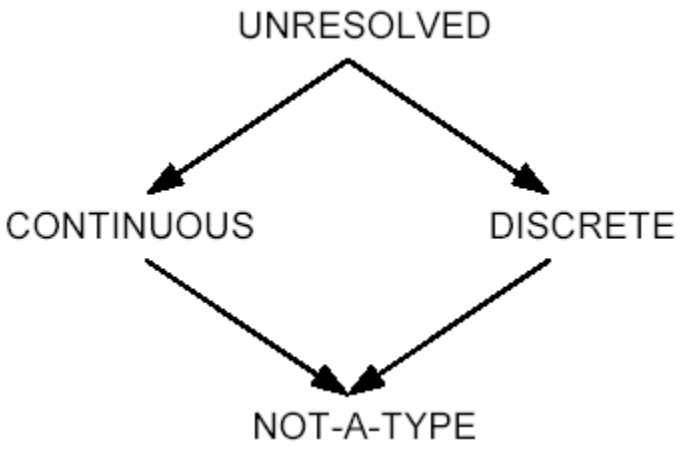


Figure 12 A signal type lattices

[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 CTEmbeddedDirector. 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 CTEmbeddedDirector 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 CTEmbeddedDirector 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 CTEmbeddedDirector 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

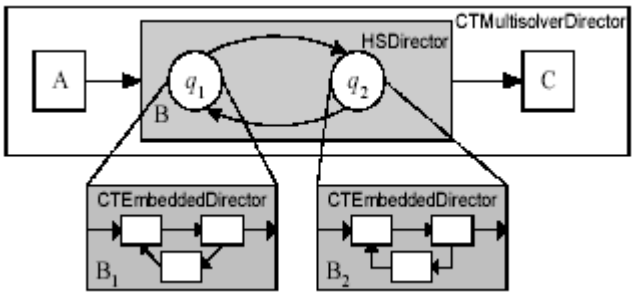


Figure 13 Hybrid system modeling

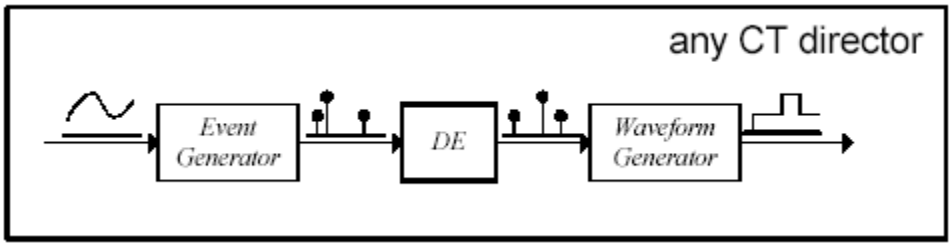


Figure 14 Embedding a DE component in a CT system

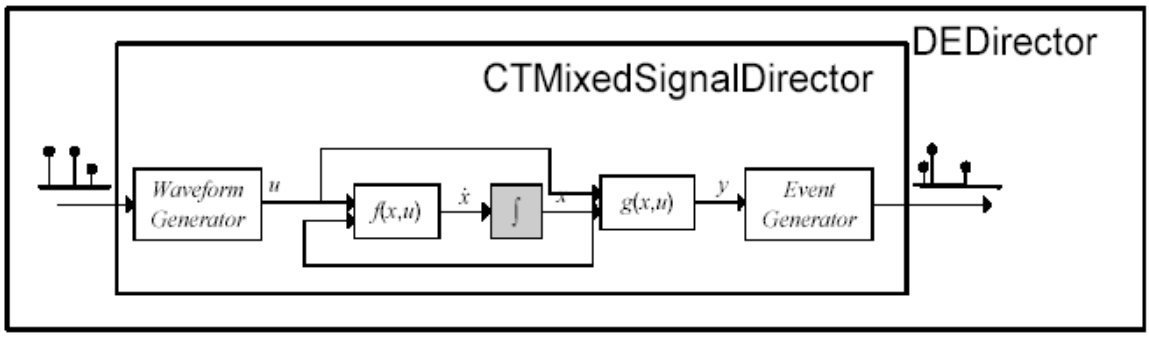


Figure 15 Embedding a CT component in a DE system

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

$$\dot{x} = f(x, t) \quad . \tag{30}$$
$$x(t_0) = x_0$$

If f satisfies the conditions:

- [Continuity Condition] Let D be the set of possible discontinuity points; it may be empty. For each fixed $x \in \Re^n$ and $u \in \Re^m$, 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 $k : \Re \rightarrow \Re^+$, where $\Re^+$ is the set of non-negative real numbers, such that $\forall t \in \Re, \forall \zeta, \xi \in \Re^n, \forall u \in \Re^m$

$$\|f(\xi, u, t) - f(\zeta, u, t)\| \leq k(t)\|\xi - \zeta\| \, . \tag{31}$$

Then, for each initial condition $(t_0, x_0) \subseteq \Re \times \Re^n$ there exists a unique continuous function $\psi : \Re \rightarrow \Re^n$ such that,

$$\psi(t_0) = x_0 \tag{32}$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t) \qquad \forall t \in \Re \backslash D \, . \tag{33}$$

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

Theorem 2. [Contraction Mapping Theorem.] If $F : \Re^n \rightarrow \Re^n$ 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 $\sigma \in \Re^n$, $\|\sigma - x\| \leq \varepsilon$, such that $\sigma = F(\sigma)$. And $\forall \sigma_0 \in \Re^n, \|\sigma_0 - x\| \leq \varepsilon$, the sequence

$$\sigma_1 = F(\sigma_0), \sigma_2 = F(\sigma_1), \sigma_3 = F(\sigma_2), \ldots \tag{34}$$

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.

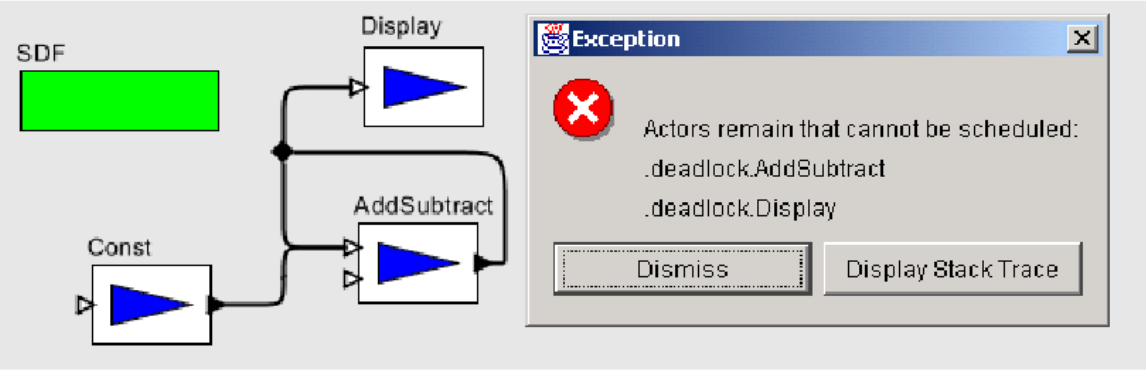


Figure 16 An SDF model that deadlocks

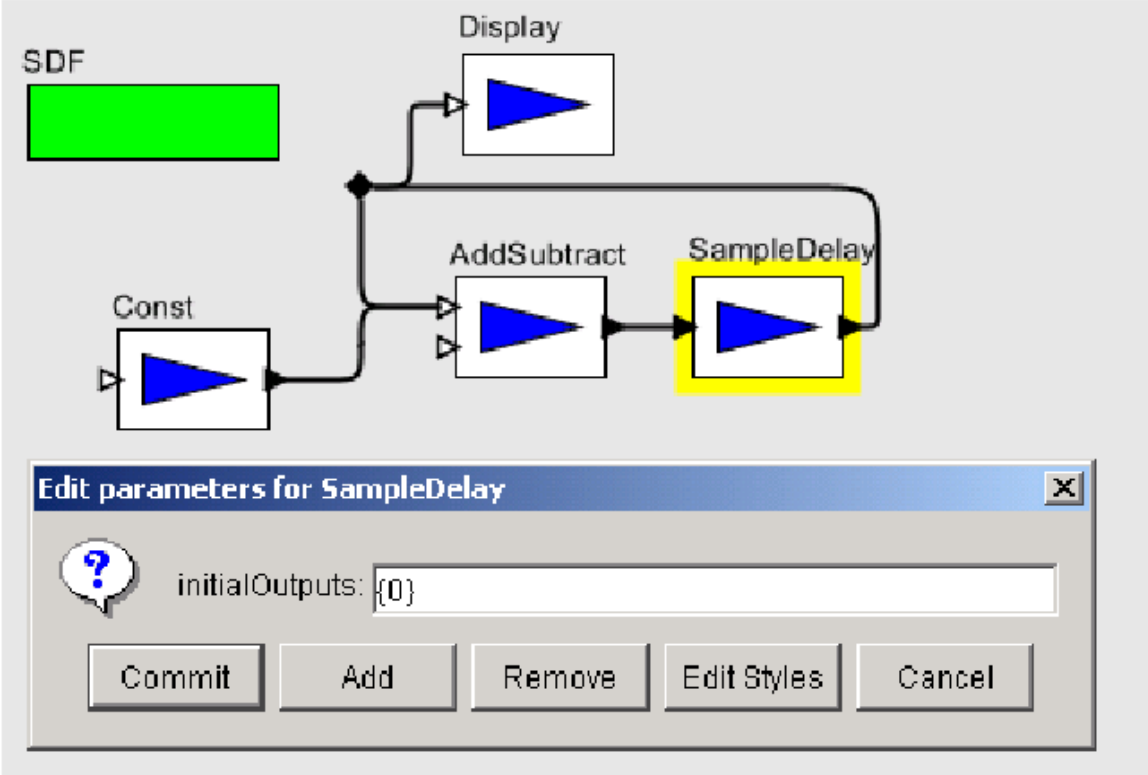


Figure 17 The model of Figure 17. 1 corrected with an instance of SampleDelay in the feedback loop

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*.

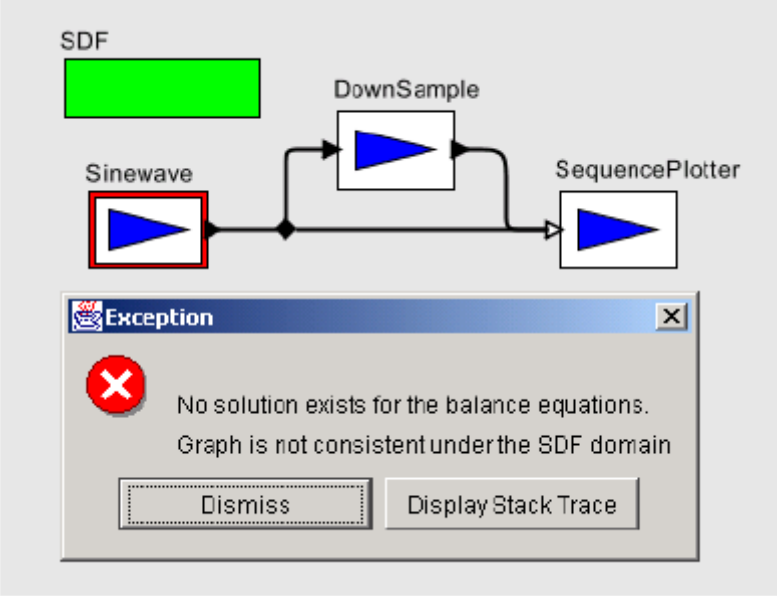


Figure 18 An SDF model with inconsistent rates.

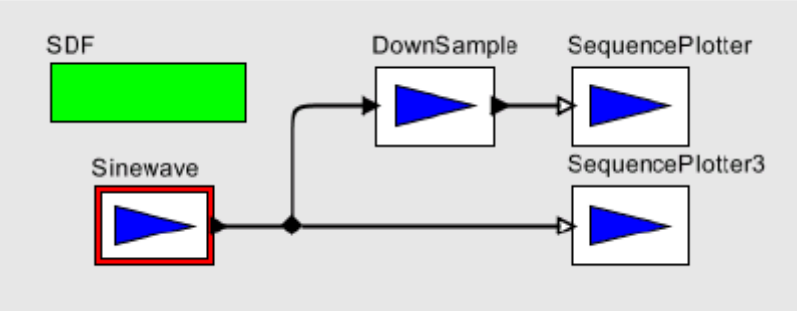


Figure 19 Figure 18 modified to have consistent rates.

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

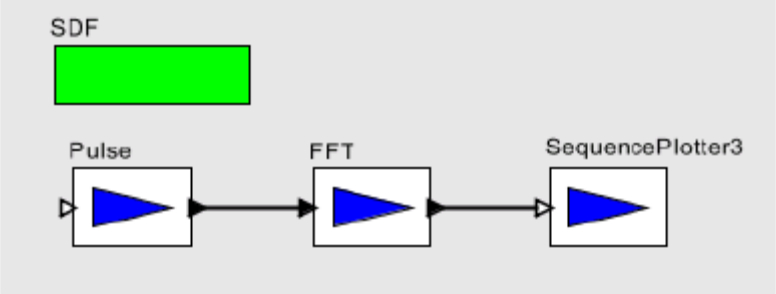


Figure 20 A model that plots the Fast Fourier Transform of a signal. A single iteration must be executed to plot all 256 values of the FFT, since the FFT block produces and consumes 256 tokens each firing.

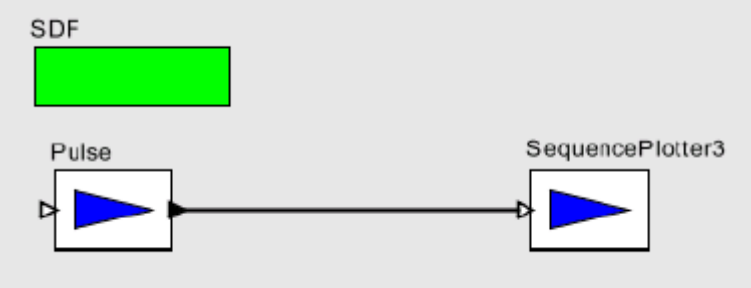


Figure 21 A model that plots the values of a signal. 256 iterations must be executed to plot the entire signal.

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 timeusing an FIR filter. Such an 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 firing[3]. 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.

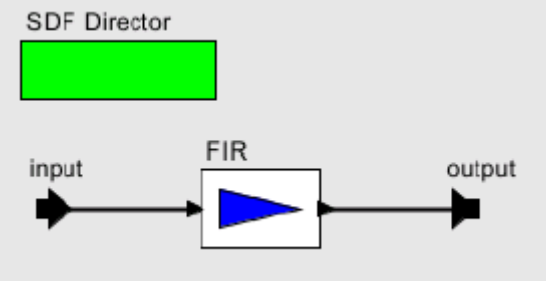


Figure 22 A model that implements a block FIR filter. The *vectorizationFactor* parameter of the Simulator is set to the size of the block.

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) × ProductionRate(A1) = Firings(B) × ConsumptionRate(B1)
Firings(A) × ProductionRate(A2) = Firings(C) × ConsumptionRate(C1)
Firings(C) × ProductionRate(C2) = Firings(B) × 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 acyclic[4]. 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 look 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

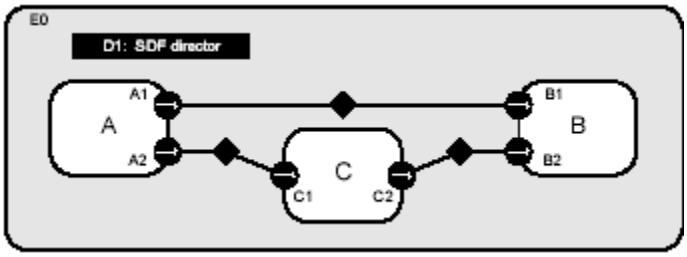


Figure 23 An example SDF model.

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

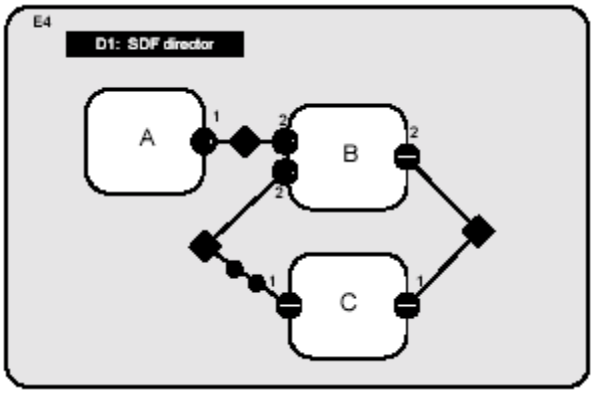


Figure 24 A consistent cyclic graph, properly annotated with delays. A one token delay is represented by a black circle. Block C is responsible for setting the *tokenInitProduction* parameter on its output port, and creating the two tokens during initialization. This Block Diagram can be executed using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

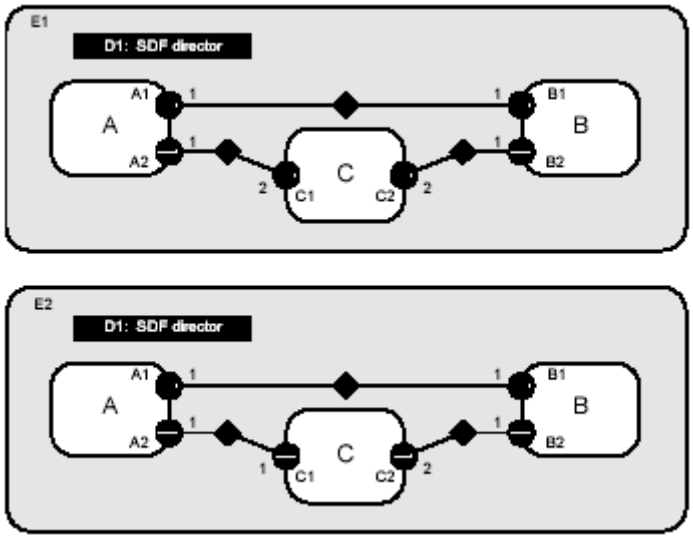


Figure 25 Two models, with each port annotated with the appropriate rate properties. The model on the top is consistent, and can be executed using the schedule A, A, C, B, B. The model on the bottom is inconsistent because tokens will accumulate between ports C2 and B2.

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.

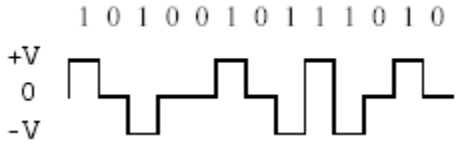
Based on the VisualSim philosophy of hierarchical composition of heterogeneous models of computation, the *charts[5] formalism allows embedding hierarchical FSMs within a variety of concurrency models. If tight synchronization is possible and desirable, then FSMs can be composed by the synchronous-reactive 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.

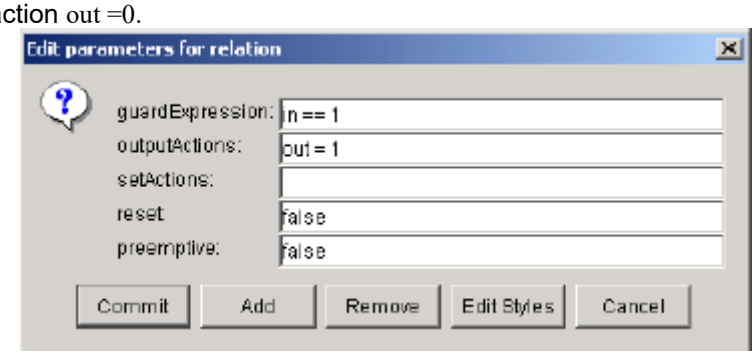


Figure 26 The dialog box for editing parameters of a transition.

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}.`

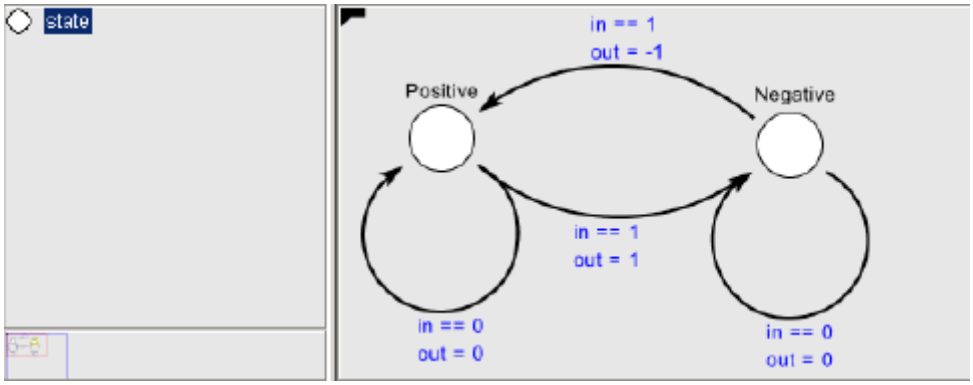


Figure 27 ModelBuilder FSM editor showing the AMICoder.

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, 2n 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.

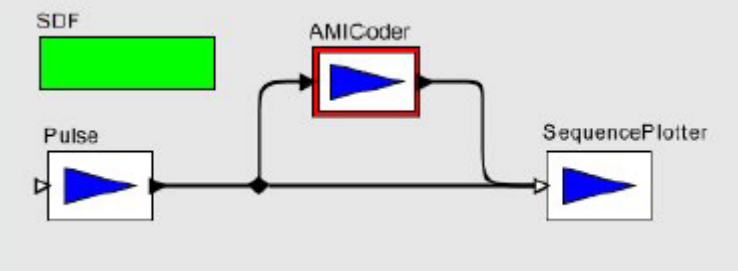


Figure 28 An SDF model with the AMICoder.

- 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.

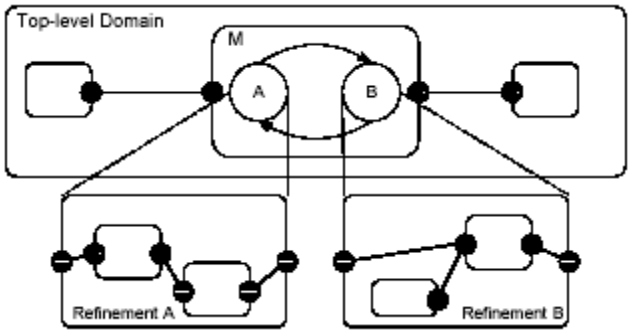


Figure 29 A Hierarchical FSM example.

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-Hierarchical [6] 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.

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

- Add an input port named in to Controller. Add an output port named out for both RefinementP and RefinementN.
- 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`.

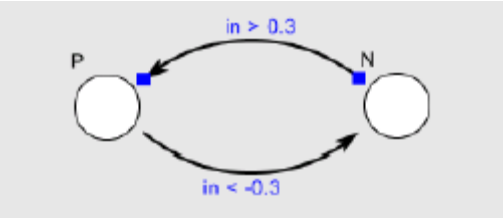


Figure 30 The mode controller for SchmidtTrigger.

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.

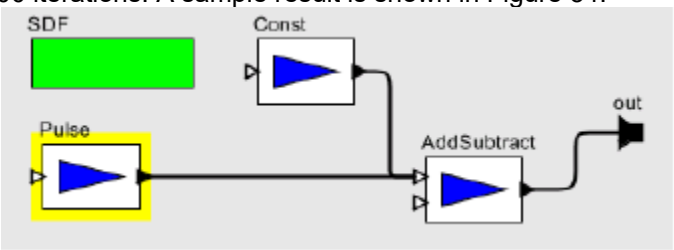


Figure 31 Model for the refinements in SchmidtTrigger.

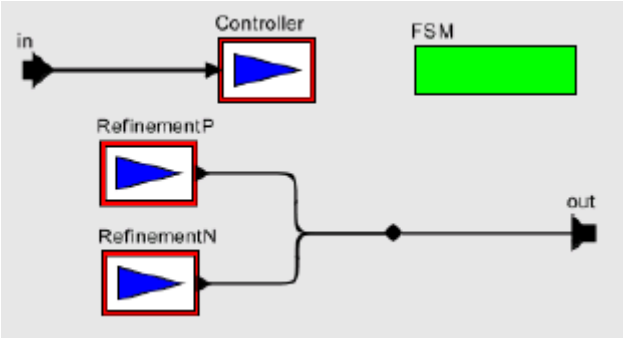


Figure 32 The SchmidtTrigger modal model.

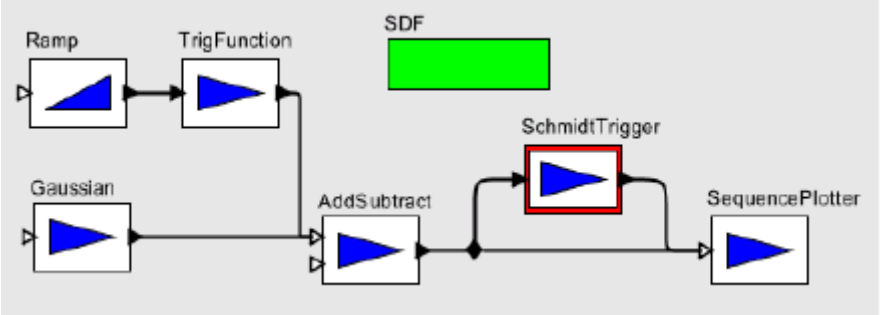


Figure 33 The top-level model with the SchmidtTrigger.

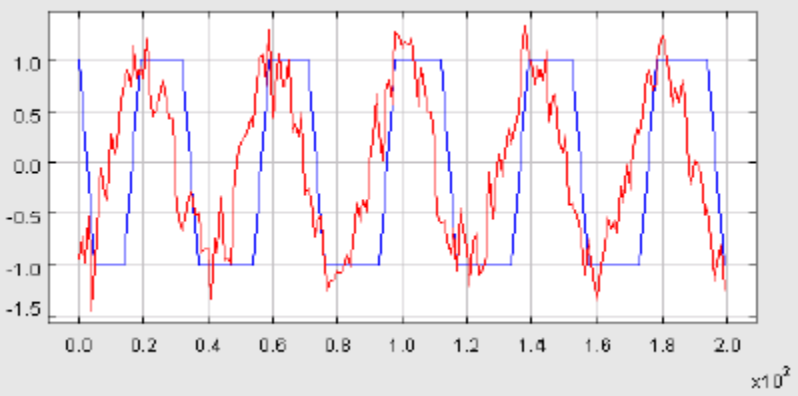


Figure 34 Sample result of the model shown in Figure 33.

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 Tool Features

5 Batch Mode Simulation Execution

5.1 Introduction

VisualSim models can be executed from a script to run in a non-interactive mode. The sequence to simulation model can be executed using any script language. The batch file can contain multiple lines of this script with different parameter values and model file names.

The batch mode simulation cannot handle any graphical display- text or waveforms. So, all text and waveform plots must be set to Save mode before starting the execution. No spaces are permitted in the file names and file paths. Otherwise an report will be reported.

In addition to the standard script format, the user must add any Java options that are used in the VisualSim.bat or VisualSim.sh file that invokes the graphical VisualSim install. This ensures the simulation speed is the same for the graphical and non-graphical editions. The common options would be “--add-opens java.desktop/sun.font=ALL-UNNAMED -Dvs.lic=default -Duser.language=en-US -Djava.security.policy=bin\policyAll -server -Xms1024m -Xmx2048m -XX:MinHeapFreeRatio=50 -XX:MaxHeapFreeRatio=50”.

5.2 Error checking

The Batch mode simulation assumes you have checked all the parameter names, values, paths, and the file name correctly. If you have any error, the simulation simply terminates. Most of the time, it will not generate any errors.

5.3 Batch mode simulation script format

Manually:
The basic format for the Batch Run Execution is:

```
<Path to Java bin directory + java> --add-opens
java.desktop/sun.font=ALL-UNNAMED -Dvs.lic=default -
Duser.language=en-US -Dvs.lib="%VS_C_Library%" -
Djava.security.policy=bin\policyAll -classpath <List of paths>
VisualSim.actor.gui.VisualSimBatchModeSimulator –resultpath <Path to
save the simulation summary> -<Parameter Name> <Parameter Value>
..... <Model Name with location>
```

where

Option	Required?	Format
<Path to Java bin directory>	Yes	**C:\java\jdk1.6.0_07\bin\java- Windows /usr/java/jdk1.6.0_07/bin/java- UNIX**
Required items to support Java 17 and above	Yes	**--add-opens java.desktop/sun.font=ALL-UNNAMED -Dvs.lic=default -Duser.language=en-US**

Option	Required?	Format
		-Dvs.lib="%VS_C_Library%" -Djava.security.policy=bin\policyAll
<List of Paths>	Yes	**VisualSim Install directory is required. Others fields are optional. (see below for the paths to include)**
-resultsummary	Optional	**This is a keyword and must not be used as a parameter name. This is used when the user saves the summary for each simulation run in a separate directory.**
<Path to save the simulation summary>	Optional	**Used only with – resultpath and must follow the keyword. This is the path in the standard OS path structure to save the summary for each simulation run in a separate directory.**
-<Parameter Name>	Optional	**-Simulation_Time (One corresponding value required) (See below for parameter support)**
<Parameter Value>	Optional	**1.0 (One per parameter name) (See below for data value support)**
<Model Name>	Yes	**Path + Model Name (See below for format details)**

Example:
D:\jdk1.20.0_24\bin\java --add-opens java.desktop/sun.font=ALL-UNNAMED -Dvs.lic=default -Duser.language=en-US -Djava.security.policy=bin\policyAll -classpath D:/VisualSim/VS_AR VisualSim.actor.gui.VisualSimBatchModeSimulator –resultpath C:\VisualSim\User_Library -Input_Rate 1.0 -Execution_Time 2.5 D:/VisualSim/VS_AR/doc/Training_Material/Tutorials/Part3.xml

5.3.1 List of Paths

The -classpath contains the VisualSim install directory. Other directories are not required but can be added as required. The format for Windows is **-classpath file_path1;file_path2… without the** "" and for UNIX is **-classpath file_path1:file_path2…**.

Example of VisualSim install directory format:

C:\VisualSim\VisualSim14- Windows

/usr/VisualSim/VisualSim14- UNIX

5.3.2 Path to save the Batch simulation results file

The -resultpath is an optional argument for the simulation execution control. The format for Windows is **-resultpath file_path1** without any "".

Example of directory format:

C:\VisualSim\VisualSim14- Windows

/usr/VisualSim/VisualSim14- UNIX

5.3.3 Parameter Name

The script can accept top-level parameters, parameter in hierarchical block or a block parameter. The format is “-parameter_name parameter_value -parameter_name parameter_value, -Block_name.parameter_name parameter_value …”. The list can contain any number of parameters. It is necessary to include only those parameters that need to be modified. Lower level parameters can be included by providing their Window Name in the order to reach the lowest hierarchical level.

1. Top-level: Simulation_Time
2. Hierarchical Block: Hierarchical Block Name.Parameter Name (Eg:

   Processor_Block.CPU_Speed)
3. Block Parameter**:** Digital.startTime

5.3.4 Parameter Value

All standard data types are supported here. The format must match whatever the default value used in model when initiatlizing the parameters. Parameters listed in the parameter set are also supported here. The only restriction is that strings used inside a array or otherwise must be enclosed in \”. For example, “DRAM1” must be written as “\”DRAM1”\”.
Data types supported:
Basic: string, integer, double, long, boolean,
Complex: data structures (all types), matrix (all types), arrays (all types)
If the script contains a variable that is set first and then used as a value in the simulation line, make sure there are no spaces after the “=” or extra spaces on Linux side.
For example,
this line is correct:

```
set num=5
java -classpath C:/VS VisualSim.actor.gui.VisualSimBatchModeSimulator -Param
%num% C:/VS /Model.xml
```

while, this line is incorrect:

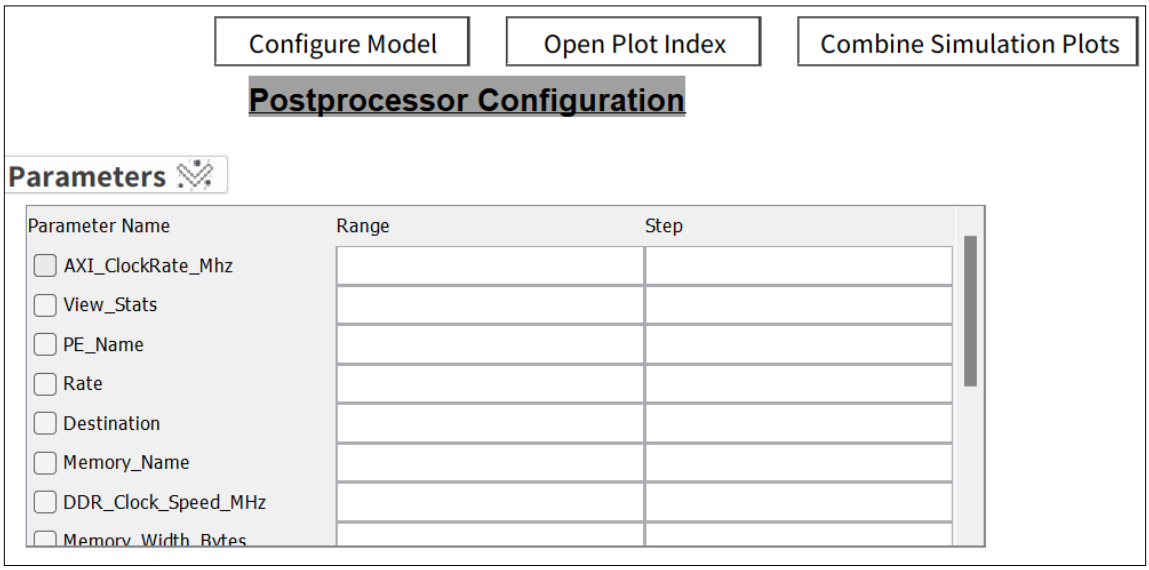
```
set num= 5  (Space is not permitted)
java -classpath C:/VS VisualSim.actor.gui.VisualSimBatchModeSimulator -Param
%num% C:/VS /Model.xml
```

5.3.5 Model Name

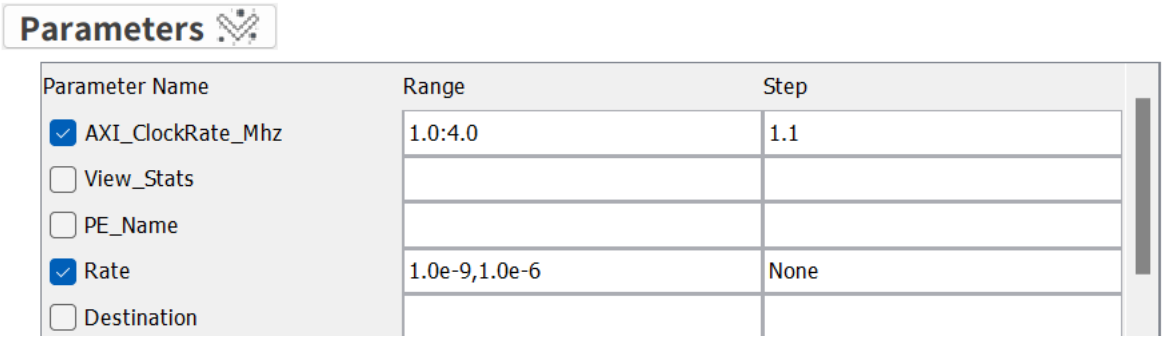
The last item in the line is the file name. The preferred format is to use the URL technique. This would be “D:/VisualSim/Model_Example.xml” on Windows and “/VisualSim/Model_Example.xml” on Linux.
If the model file is within the VisualSim install, the format for Windows is `"User_Library/Model/Command_Line_Modeling.xml" and for UNIX is "./User_Library/Model/Command_Line_Modeling.xml "`.
An alternate option is to use
“$VS/`User_Library/Model/Command_Line_Modeling.xml`”.
To get the URL format, open the file in VisualSim Architect or in a Web Browser, and copy the URL address shown.

5.3.6 Using Post Processor to create the batch file .

1. Select Configure Model button in Post Processor and choose the model that you want run in batch mode. Make sure the text displays and plots are disabled or set to save option.



2. Enable the required parameters for batch run and add range or fixed set of values for the enabled parameters. In the following image the AXI_ClockRate_Mhz is configured with range and the step value is 1.1. So there will be entries with 1.0, 2.1 and 3.2 as its value with other combinations. Now for the Rate parameter fixed set of values are defined that is 1.0e-9 and 1.0e-6, so the step is set to None.



3. Provide appropriate type of values to for the selected parameters.
4. Once the parameters are filled up, select Ok.

5. Pop window with “Batch File Generated” will be shown, click Ok.
6. In the Model directory (ModelName)_index.xml file will be generated along with Sim_Batch_Run.bat / Sim_Batch_Run.sh file.
7. To run the batch file on different CPU core check section 6

5.4 Viewers, Ploters and Text Displays

Models run in batch mode must not contain any graphical viewers or interactive windows. All plots, results and statistics must be written to file. If you have ViewPlot or ViewText enabled in any display, you get the following error:

```
---------------------- Exception Reported ---------------------
VisualSim.kernel.util.IllegalActionException: Cannot find effigy for top level:
.Model1
  in .Model1.Hblock1.HHBlock1.Display_Plotter
---------------------------------------------------------------
```

The paths in any file write is relative to the <VisualSim install directory> from where the batch file is being executed. If the VisualSim_batch_mode_run.bat(.sh) is executed from C:/VisualSim/VS (/VisualSim/VS), then the file paths for all the writes are relative to this position. You get the following error if the directory does not exist.

```
---------------------- Exception Reported ---------------------
VisualSim.kernel.util.IllegalActionException: Exception, writeFile, File Path:
Stats_of_Controller.txt
Exception Reported: java.lang.IllegalArgumentException: URI is not absolute
---------------------------------------------------------------
```

The best file name options are:

1. Filename.txt
2. Results/Filename.txt (Saved in Results folder)
3. C:/VisualSim/Models/Filename.txt or /VisualSim/Models/Filename.txt

5.5 Output

5.5.1 Model statistics

All statistics, plots and files are saved in the directories specified by the respective block parameters.

5.5.2 Summary

The success or failures of the simulation runs are reported on the terminal windows executing the script and in the `"Batch_Mode_Results_Summary.txt"`. **`This file is saved in the VisualSim install directory or in the path provided by -resultpath parameter.`**
If the file does not exist, a new file is created. The file format is as follows:

```
Start Model: MyModel    2023.10.06 AD at 02:15:18 IST
```

```
End  Model: MyModel     2023.10.06 AD at 02:17:28 IST
Parsing Parameter Name: Byte_Config Value: 0
Parsing Parameter Name: Command_Config    Value: 0
Parsing Parameter Name: Address_Config    Value: 0

Start Model: MyModel    2023.10.06 AD at 02:15:18 IST
   ----------------------- Exception Reported ----------------------
   "Exception message"
   -----------------------------------------------------------------
End  Model: MyModel     2023.10.06 AD at 02:17:28 IST
Parsing Parameter Name: Byte_Config Value: 0
Parsing Parameter Name: Command_Config    Value: 0
Parsing Parameter Name: Address_Config    Value: 0
```

5.5.3 Saving text and Plots

The saving folder must exist. The following formats work correctly in the Command-Line version:

$VS/User_Library/Folder/results
$CLASSPATH/User_Library/Folder/results
FileName

5.6 Detailed Step by Step procedure for Batch mode simulation

The following image shows a demo model that we will use to explain the batch mode simualtion process.

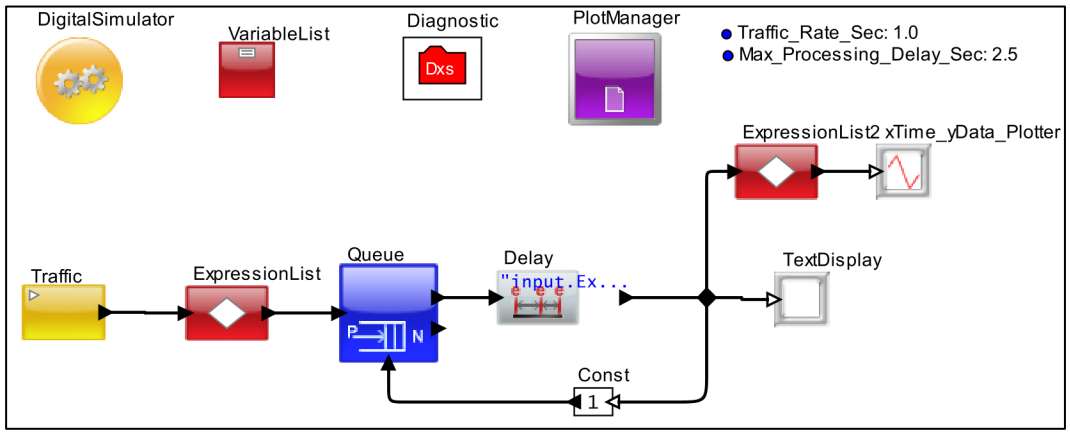


Figure 1: VisualSim demo model

We make sure the “PlotManager” block is added in the model for creating an index file which keeps track of all the parameter combinations and links it with the statistics generated. We will use existing parameter or add parameters to the model so that its value can be changed from the top level without making changes to the block parameters or to the ExpressionList.

NOTE: After adding the block “PlotManager”, we need to setup the configuration file using Post Processor utility. Else the model will not run.

The construction steps required to build the model as shown in Fig. 1 is listed below:

- Open the model that you want to run batch mode simulation.
- Demo model used for this located at VS_AR\doc\Training_Material\Tutorial\WebHelp\Tutorial\Getting_Started\Part2.xml
- Do File->Save As and save the model as “Part3.xml”
- Add PlotManager block to be Block Diagram Editor (BDE) window
    - PlotManager block can be obtained from: Results->PlotManager

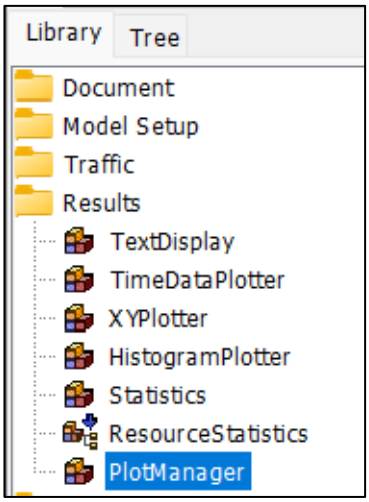


Figure 2: PlotManager block location

- Double click on the PlotManager to view the parameters of PlotManager, as shown in Fig. 3:

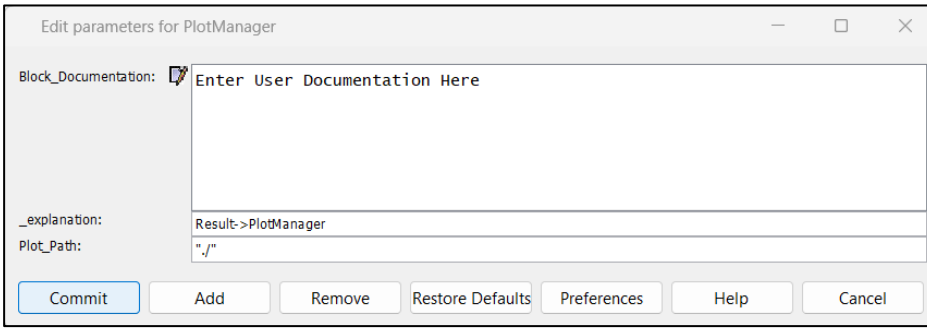


Figure 3: PlotManager parameter list

- We need to define the Plot_Path correctly. In Fig.3, we have used a relative path. If we use an absolute path, make sure to use “/” instead of “\”.
- Click on Commit

- Drag and drop Parameter into the model.

- Location: Model Setup -> Parameter

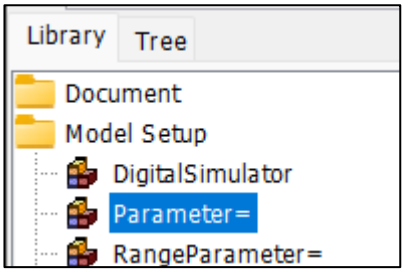


Figure 4: Parameter location

- Right Click on Parameter -> Customize Name

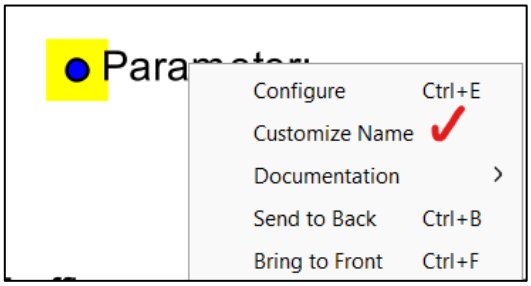
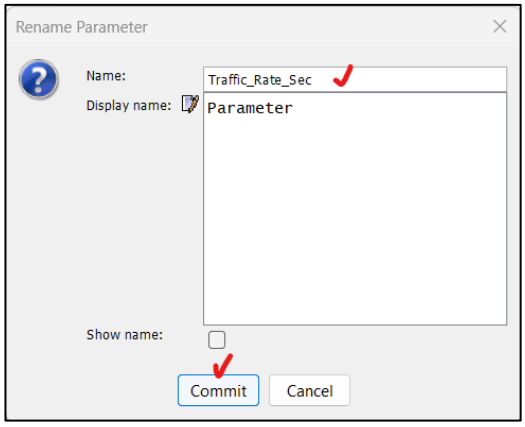


Figure 5: Customize Name on the Parameter

- Give the name as Traffic_Rate_Sec

Figure 6: Renaming Parameter into application specific name



- Click on Commit
- Double click on Traffic_Rate_Sec parameter
- Configure the value as 1.0 and click on Commit

- Double click on Traffic block and link Traffic_Rate_Sec to “Value_1” parameter

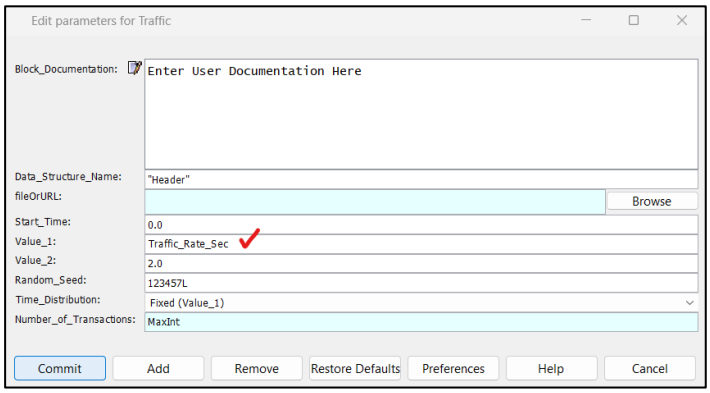


Figure 7: Updated Traffic parameter list

- As above, drag and drop another parameter and rename it as “Max_Processing_Delay_Sec”
  - Set it to 2.5 seconds

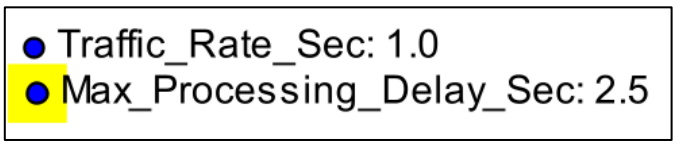


Figure 8: Parameters after configuring them with the required values

  - Link this parameter to the logic in ExpressionList

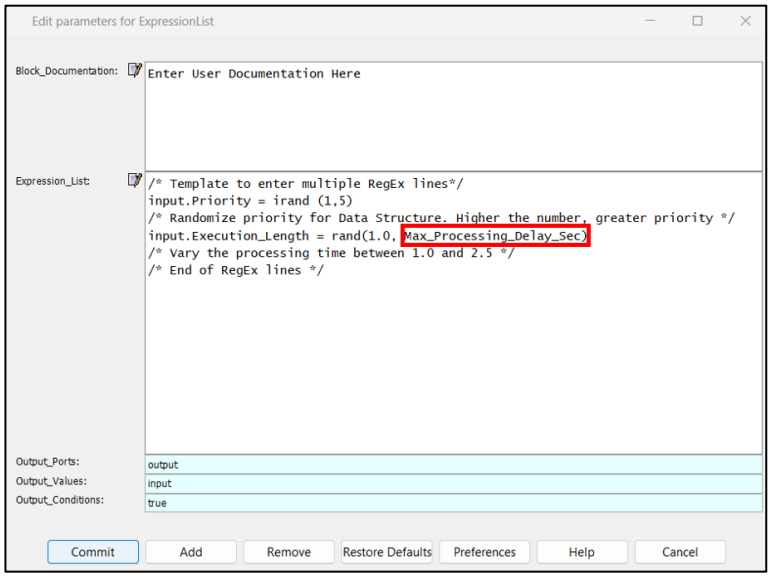


Figure 9: Max_Processing_Delay_Sec parameter is linked to the logic in ExpressionList

  - Click on commit

- Save the model and close it.
- Go to PostProcessor utility
    - PostProcessor can be accessed from the home page of the tool

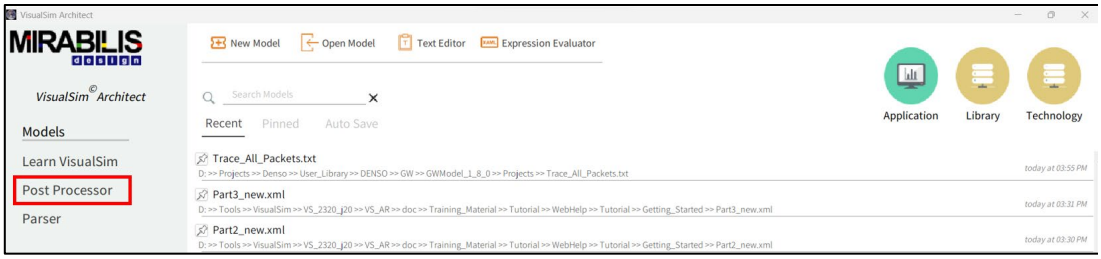


Figure 10: PostProcessor Utility in the VisualSim Architect home page

    - Click on the PostProcessor link
    - This will open a different GUI which has 3 options on top. Click on “Configure Model” option

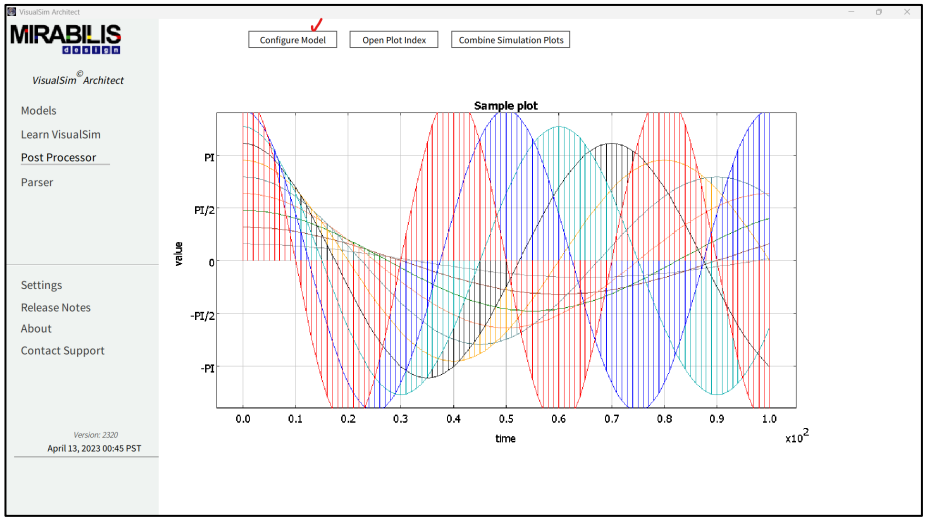


Figure 11: Configure Model option in PostProcessor

- This will bring up a browse window through which we have to navigate to the folder where the Part3.xml is located and select Part3.xml

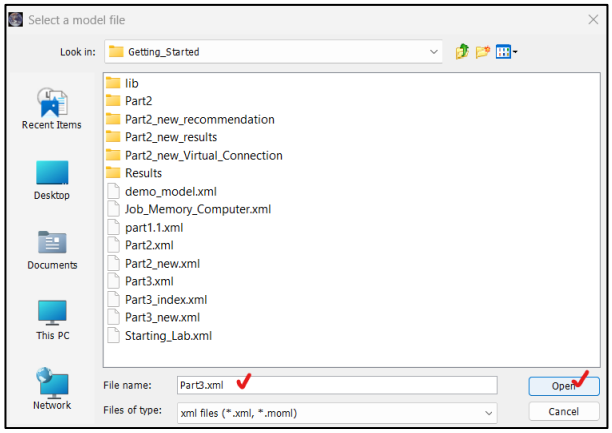


Figure 12: Select the demo model

- This will bring up a panel which has the top level parameter of the demo model listed

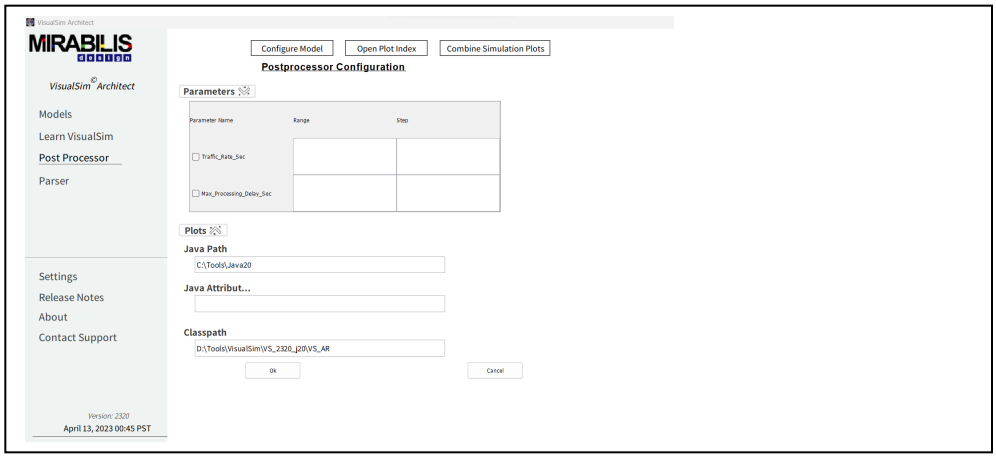


Figure 13: Configuration panel to set up the parameter sweep

- Select the parameters that we want to sweep across by enabling them and set the range and step.

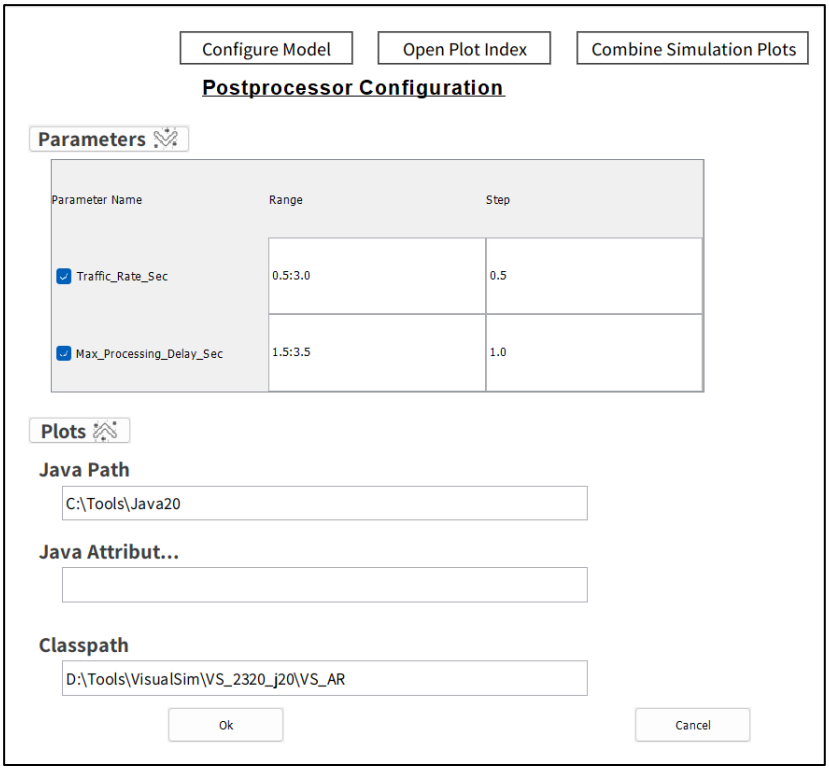


Figure 14: Setting up Parameter ranges and step size

- Select the statistics that needs to be saved for each of the simulation. These statistics can be viewed after the regression run and these results can be compared against each other.

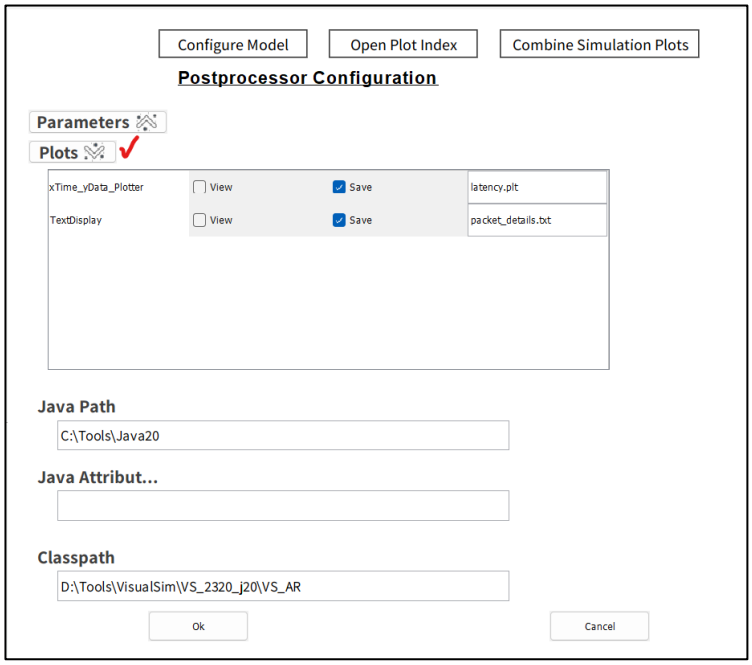


Figure 15: Configuring the statistics that need to be saved

NOTE: The plots that needs to be saved must specify the file name with the file extension ".plt" at the end and for the text displays, we need to use the file extension ".txt". Also, since the regression runs are executed from commandline, we need to make sure the "View" option is disabled.

- Click on Ok. This will show a message "Batch File Generated"

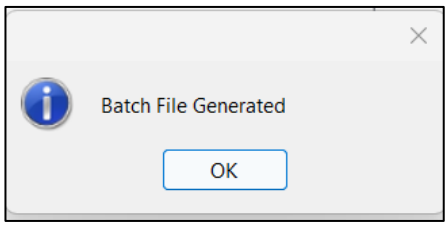


Figure 16: Batch File Generation Pop up

- Click on Ok.
- We need to the folder where the demo model is located. There will be two .bat (windows) / .sh (linux/Mac OS) created.

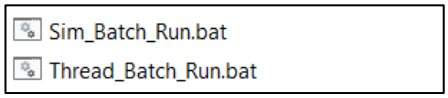


Figure 17: Generated batch files

- Sim_Batch_Run.bat file contains all the simulation configurations set up for this parameter sweep (regression run). While the Thread_Batch_Run.bat is used to run each of the simulation runs across multiple cores. This will speed up the regression runs as multiple simulations will be then executed in parallel. If we have a powerful workstation or a server with lot of processor cores, then this can be used to speed up the simulation.

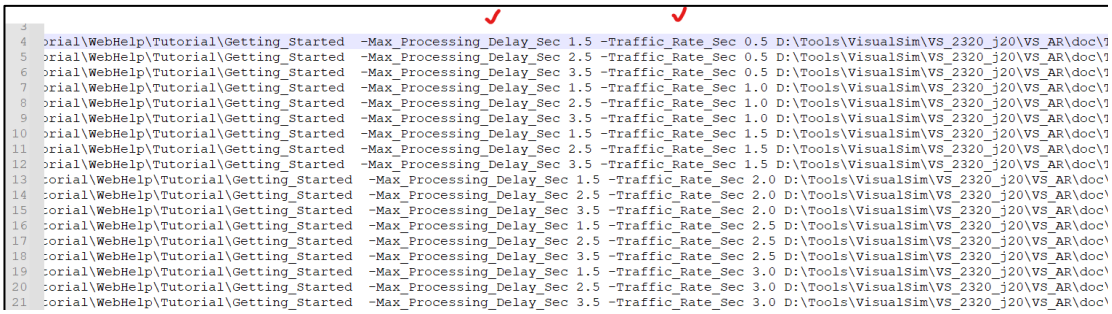


Figure 18: Sim_Batch_Run.bat file. The simulation configurations are highlighted above

- To execute the .bat file, we just need to double click on the .bat file. We will use Sim_Batch_Run.bat:

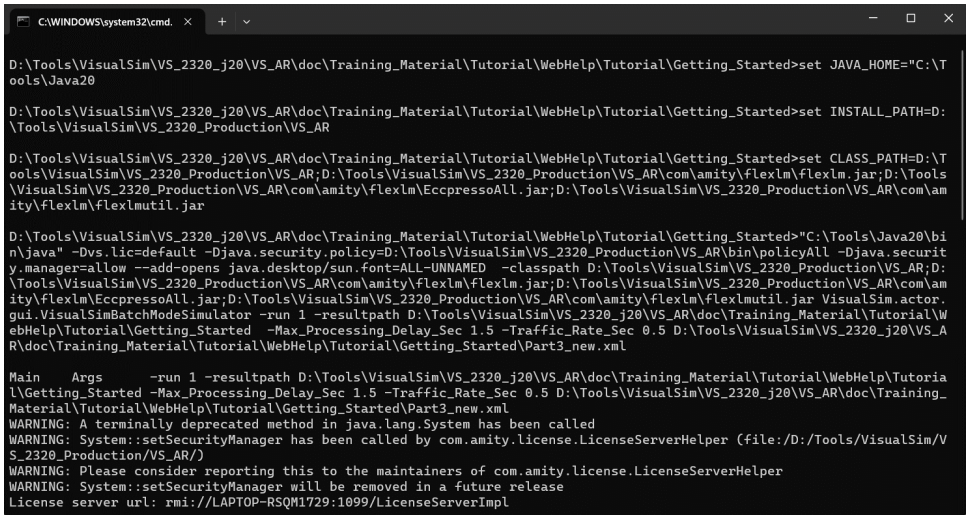


Figure 19: Command line execution window

- Once the simulations completed, we can find all the generated statistics being saved to the local folder where the demo model is placed.

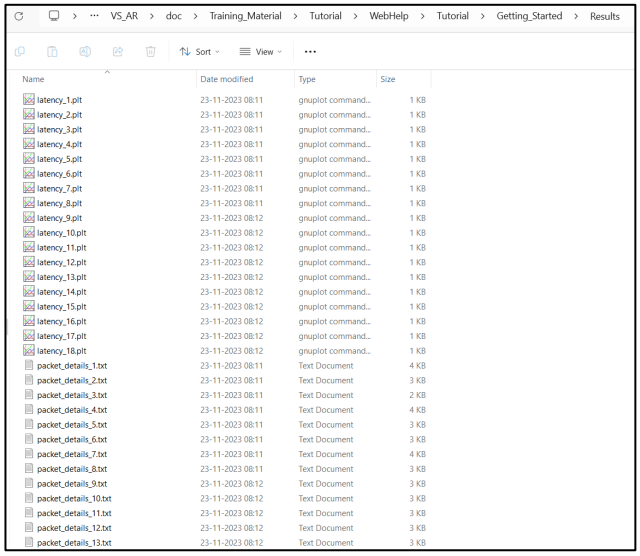
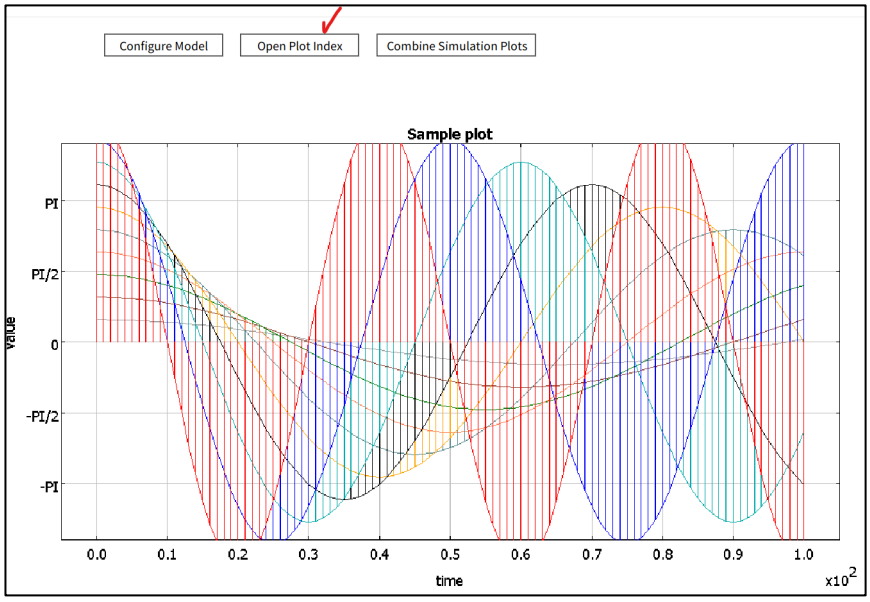


Figure 20: Generated statistics

- Since we had set up “Requirement Tracker” in Part2.xml, all of the simulations we executed in the regression run will check for the requirements we had specified and the results will be generated in the <Model_Name>_results folder.



Name	Date modified	Type	Size
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:00	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:02	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Resource_Type_Part...	23-11-2023 13:02	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:00	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB
Diagnostic_Stats_Script_Non-Array_P...	23-11-2023 13:01	Microsoft Excel Co...	1 KB

Figure 21: Requirement tracker output files generated for each of the simulation runs

- We can compare the .plt files generated using the PostProcessor utility

Figure 22: Open plot index

This will bring up the browse window through which you need to select the Part3_index.xml file. This will be located in the path we have set up in PlotManager.

- We need to select the simulation configurations for which we wish to compare the results and select the statistics that needs to be compared using the new window that comes up:

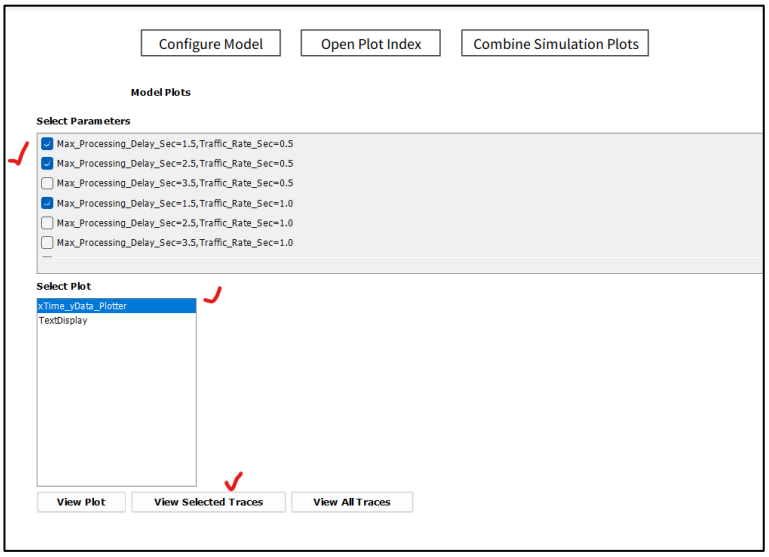


Figure 23: Result comparison panel

- This will being up the plot which compares the results for the selected simulations:

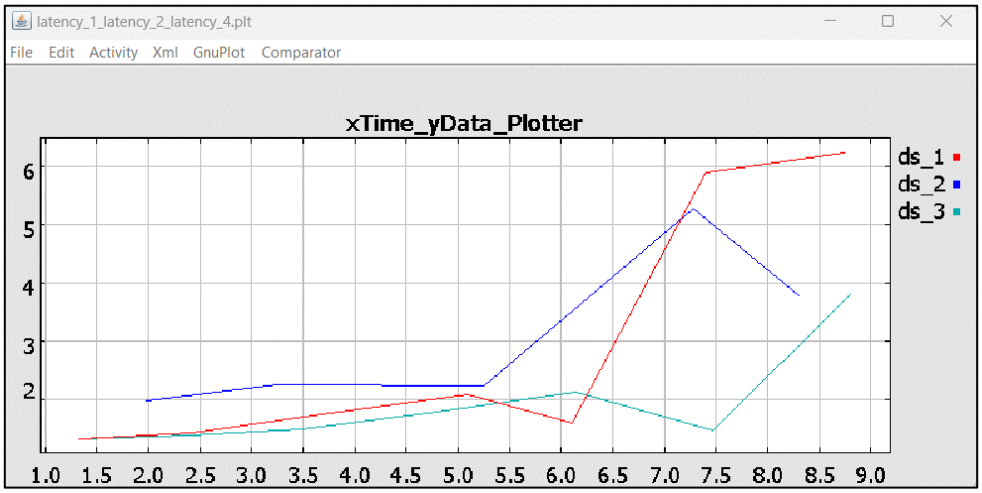
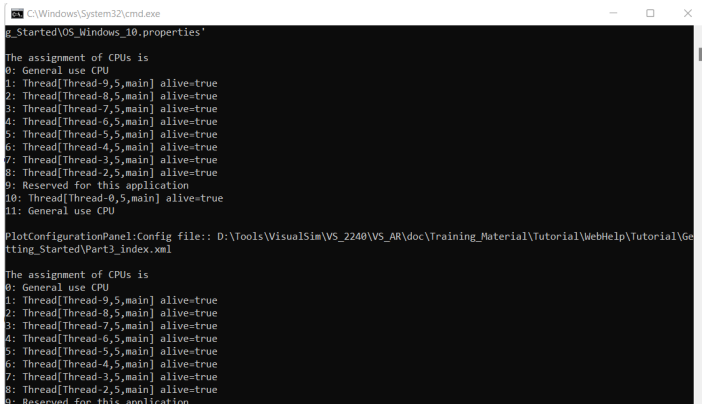


Figure 24: Results from multiple simulations being compared

- We can execute the “Thread_Batch_Run.bat” similarly as well.
  Figure 24: Multi core simulation using Thread_Batch_Run.bat



6 Simulation on Multi Core

The Batch mode simulation can be accelerated further by running the simulation on multiple cores of your system. Once you have created the batch file and index file as mentioned in section 5 (above). the Thread_Batch_Run.bat file will be available with the Sim_Batch_Run.bat file.

By double clicking the Thread_Batch_Run.bat the batch runs are assigned to different cores and the simulations are performed parallely. This increase the parallelism and completes the batch runs faster.

Custom Batch file format for parallel execution:

The default format of Sim_Batch_Run.bat that is generated through the post processor looks like the following.

```
set JAVA_HOME="C:\Tools\Java20
set INSTALL_PATH=D:\Tools\VisualSim\VS_2410\VS_AR
set CLASS_PATH=D:\Tools\VisualSim\VS_2410\VS_AR;%INSTALL_PATH%\com\amity\flexlm\flexlm.jar;%INSTALL_PATH%\com\amity\flexlm\EccpressoAll.jar;%INSTALL_PATH%\com\amity\flexlm\flexlmutil.jar
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.
%JAVA_HOME%\bin\java" -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED  -classpath %CLASS_PATH% VisualSim.actor.

```

The batch scripts in this format can be executed across multiple cores using ThreadSimCore.bat. For that, we execute the following line in ThreadSimCore.bat:

D:\Tools\VisualSim\VS_2410\VS_AR\ThreadSimCore.bat D:\Tools\VisualSim\VS_2410\VS_AR\doc\Training_Material\Tutorial\WebHelp\Tutorial\Getting_Started\Sim_Batch_Run.bat

But this will not work with a custom batch file that is created by the user. To Run the simulation on multiple cores parallely user have to add the “start” command. Please check the following image for reference.

```
@echo off
set JAVA_HOME="C:\Tools\Java20"
set INSTALL_PATH=D:\Tools\VisualSim\VS_2410\VS_AR
set CLASS_PATH=D:\Tools\VisualSim\VS_2410\VS_AR;%INSTALL_PATH%\com\amity\flexlm\flexlm.jar;%INSTALL_PATH%\com\amity\flexlm\EccpressoAll.jar;%INSTALL_PATH%\com\amity\flexlm\flexlmutil.jar

start "Job 1" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 2" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 3" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 4" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 5" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 6" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 7" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 8" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 9" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vis
start "Job 10" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vi
start "Job 11" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vi
start "Job 12" %JAVA_HOME%\bin\java -Dvs.lic=default -Djava.security.policy=%INSTALL_PATH%\bin\policyAll -Djava.security.manager=allow --add-opens java.desktop/sun.font=ALL-UNNAMED -classpath %CLASS_PATH% VisualSim.actor.gui.Vi

```

7 Diagnostic Engine

Recommendation Engine Block

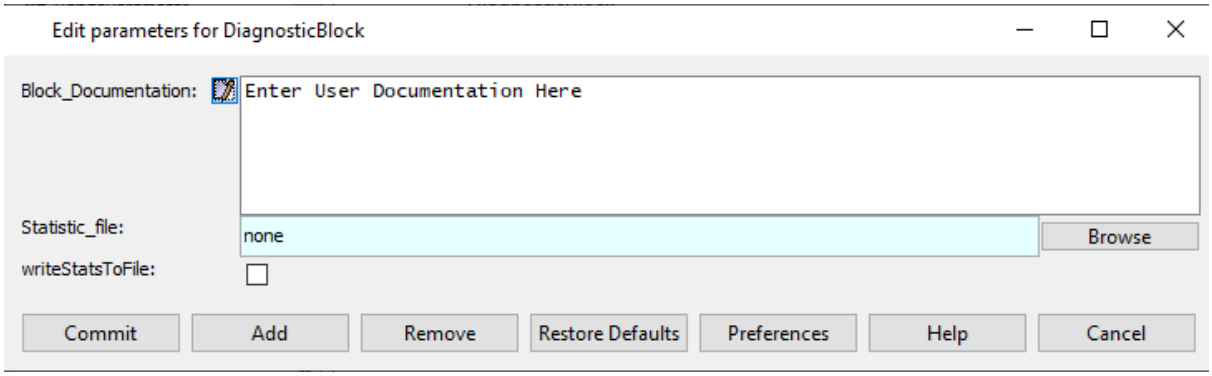
Directory: $VS\VisualSim\actor\arch\Diagnostic_Block\

7.1 Features of Diagnostic Block:

1. Statistic Generation for Resource blocks and hardware blocks in the library.
2. Analyzing variables in Script Block.
3. Error detection in Resource block.
4. Listing of Virtual Connections.

7.2 File required for generating Statistics:

1) Input CSV file: .csv statistic file (below image: Statistic_file parameter) for providing file name.



a) **Mandatory Columns :** Block, Metrics, Constraint, Constraint Value, Statistic Type
b) **Optional Column** **:** Reference Variable

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE	REFERENCE VARIABLE
MC_DRAM	DRAM_00_Total_Requests	>	10	Max	
MC_DRAM	DRAM_10_Max_Queue_Usage	>	10	Max	
MC_DRAM	DRAM_30_DRAM_Delay_Mean	>	2.50E-08	Max	
MC_DRAM	DRAM_02_Total_MB_per_Second	<	30	Max	
AXI_Top	Master_1_Read_Data_Bytes	>	300	Max	

Statistic evaluations supported in Diagnostic block.

a) Time_in_Block
b) Utilization
c) Buffer_Occupancy
d) Script Block Variables - Int, Double & Array Data Types
e) Mean, Maximum & Minimum values of all the aforementioned stats
f) Max, min, mean and standard deviation of the hardware stats that are sent to architecture setup.

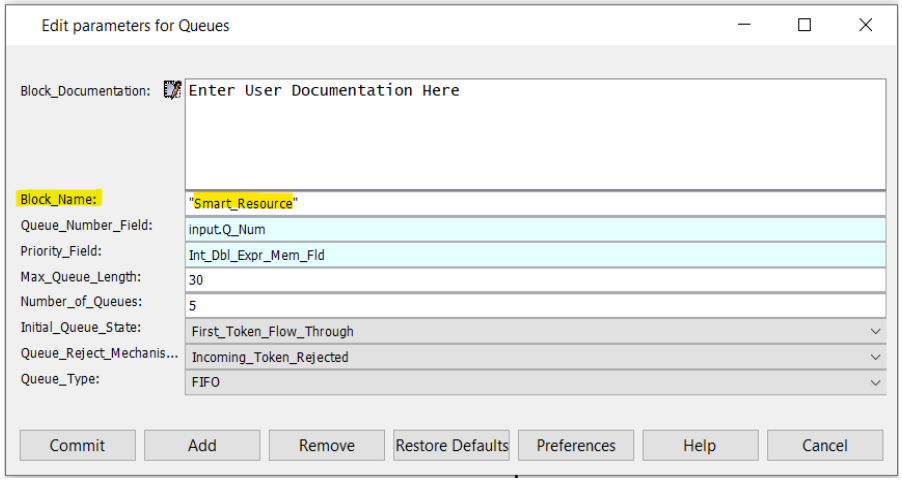
Constraint supported in Diagnostic block-
a) <
b) >
c) >=
d) <=
e) =
f) !=

7.3 Columns Description –

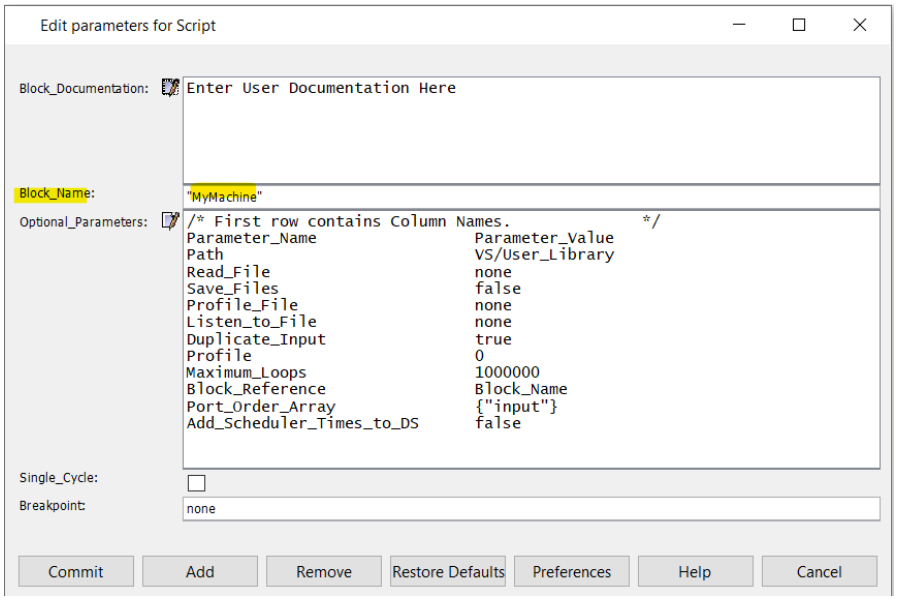
7.3.1 Block Name:

The "Block_Name" parameter value of the module which we want to monitor. Every component we use in a demo model will be having a unique name. We need to provide it here.

Eg: Resource blocks (Queue)



Eg: Script

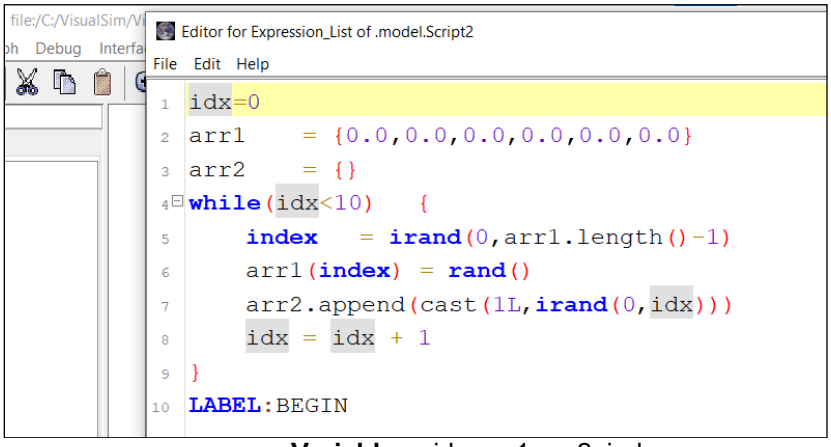


7.3.2 Metrics:

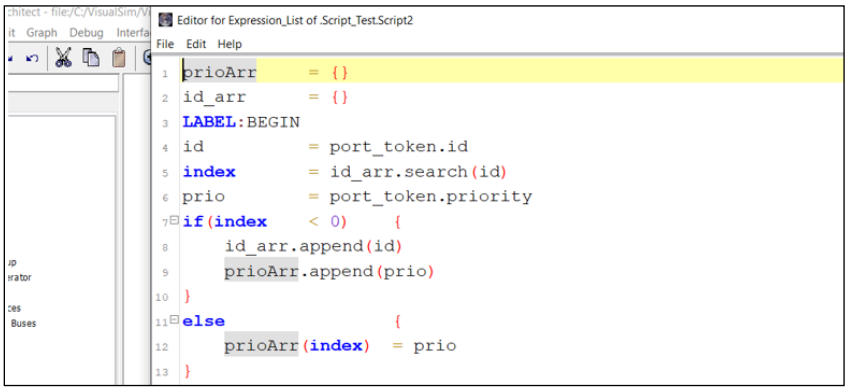
For standard resource blocks (like Queue block, Server block, System resource block, system resource extend block), following list of stats are only supported:

1. Time_in_Block
2. Utilization
3. Buffer_Occupancy

For Script blocks, we can use any variable defined within the script under this column



Variables - idx, arr1, arr2, index



Variables - prioArr, id_arr, id, index, prio

7.3.3 Constraints:

Following types are supported:

1. >
2. <
3. >=
4. <=
5. !=
6. =

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Time_in_Block	>	3.05	All
Buff_1_ch_	idx	<	1	Mean
Buff_1_ch_	idx	>=		All
Buff_1_ch_	idx	<=	0	Max
Buff_1_ch_	idx	!=	0	Min
Buff_1_ch_	arr	>	{0=10;2=20;default=5}	Mean
Buff_1_ch_	arr	<	0	All
Buff_1_ch_	arr	!=	0	Max
Buff_1_ch_	arr	>	0	Min

7.3.4 Constraint Value:

Specifies the threshold value to which the comparison is being made by diagnostic block.

Supported values:

1. Integer
2. Double
3. A structure (for array alone)

The structure should be of the following format:

{id1=value;id2=value;id3.....;default=value}

default value must be specified if user is choosing this format. This is because, if an id which is not specified in the value structure comes in, then the default value will be used as the threshold value.

id1, id2 etc. used in the above structure can either correspond to an **index** value in the array (starting from 0 to n-1, where n is the length of the array) or to the corresponding value in the **correlated** array mentioned in the *optional* **'Correlate'** column for the array.

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Time_in_Block	>	3.05	All
Buff_1_ch_	idx	<	1	Mean
Buff_1_ch_	idx	>=		All
Buff_1_ch_	idx	<=	0	Max
Buff_1_ch_	idx	!=	0	Min
Buff_1_ch_	arr	>	{0=10;2=20;default=5}	Mean
Buff_1_ch_	arr	<	0	All
Buff_1_ch_	arr	!=	0	Max
Buff_1_ch_	arr	>	0	Min

All Stats_Name have 'Constraint Value' in Integer/Double except array.
arr (for **Mean '**Type**')**
arr has the value in the *structured* form. Here '0' and '2' signifies the indexes of the array.

While '10' and '20' signifies the threshold values for indexes 0 and 2, and the other indexes have the threshold value as the value mentioned against **default** value.
For **Correlate** structured value :

BLOCK	METRICS	CONSTRAI	CONSTRAINT VALUE	STATISTIC	REFERENCE VARIABLE
Pack_1_la	prioArr	>=	{0x13 = 5,0x15 = 6, default= 5}	All	id_arr
Buff_1_ch_	arr	>=	0	Min	
Buff_1_ch_	idx	>=	0	Mean	

Input File

BLOCK	VARIABLE	REFERENCE VALUE	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAINT	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
Pack_1_la	prioArr	{ 0x13 }	{ 0x13_6.0 }	>=	0	{ 6.0 }	All	TRUE
Pack_1_la	prioArr	{ 0x13 || 0x10 }	{ 0x13_6.0 || 0x10_1.0 }	>=	0	{ 6.0 || 1.0 }	All	TRUE
Pack_1_la	prioArr	{ 0x13 || 0x10 || 0x20	{ 0x13_6.0 || 0x10_1.0 || 0x20_8.0 }	>=	0	{ 6.0 || 1.0 || 8.(	All	TRUE
Pack_1_la	prioArr	{ 0x13 || 0x10 || 0x20	{ 0x13_6.0 || 0x10_1.0 || 0x20_8.0 || 0x14_1.0 }	>=	0	{ 6.0 || 1.0 || 8.(	All	TRUE

Output File

So from the above **Output** file, we can see the value '6.0' correlated to '0x13' present in *id_arr* variable. According to this we create the Structure value in the **Input** file with the threshold values.

7.3.5 Statistic Type:

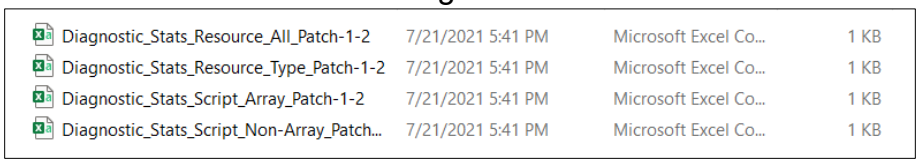
Specifies the type of calculation we have to do.
Supported values:

1. Max
2. Min
3. Mean
4. All

"All" simply means that we will be monitoring instantaneous value of a variable or a stat (Time_in_Block/Buffer Occupancy/ Utilization).

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Time_in_Block	>	3.05	All
Buff_1_ch_	idx	<	1	Mean
Buff_1_ch_	idx	>=		All
Buff_1_ch_	idx	<=	0	Max
Buff_1_ch_	idx	!=	0	Min
Buff_1_ch_	arr	>	{0=10;2=20;default=5}	Mean
Buff_1_ch_	arr	<	0	All
Buff_1_ch_	arr	!=	0	Max
Buff_1_ch_	arr	>	0	Min

Results generated :



Results -> Resource with Type set to be "All"

GENERATED_TIME	BLOCK	SOURCE	ID	Buffer_Occupancy	Buffer_Size	Time_in_Block	Utilization_Mean
1.6	Smart_Res	Traffic	4	{1 4 0 0 0}	{30 30 30 30 30}	1.3	0
2.2	Smart_Res	Traffic	6	{3 3 0 0 1}	{30 30 30 30 30}	1.7	0
3	Smart_Res	Traffic	11	{2 3 0 0 2}	{30 30 30 30 30}	2	0
3.3	Smart_Res	Traffic	23	{2 4 0 0 1}	{30 30 30 30 30}	1.1	0

Results -> Resource with Type set to be "Max", "Min" or "Mean"

BLOCK	METRICS	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC	RESULT
Smart_Res	Buffer_Oc	20	>	10	Max	TRUE
Smart_Res	Utilization	0	>	1	Min	FALSE

Results -> Script variables (no array variable) with Type set to be "Max", "Min", "Mean" or "All"

BLOCK	VARIABLE	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
Buff_1_ch_	idx	0	>=	0	All	TRUE
Buff_1_ch_	idx	1	>=	0	All	TRUE
Buff_1_ch_	idx	2	>=	0	All	TRUE
Buff_1_ch_	idx	3	>=	0	All	TRUE
Buff_1_ch_	idx	4	>=	0	All	TRUE
Buff_1_ch_	idx	5	>=	0	All	TRUE

Results -> Script variables (including array variables) with Type set to be "Max", "Min", "Mean" or "All"

	BLOCK	VARIABLE	REFERENCE VALUE	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAINT	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
63	Buff_1_ch_arr			{ 0_13.0 || 1_63.0 || 2_40.0 }	>=		0 { 13.0 || 63.0 || 40	All	TRUE
64	Buff_1_ch_arr			{ 0_63.0 || 1_40.0 }	>=		0 { 63.0 || 40.0 }	All	TRUE
65	Buff_1_ch_arr			{ 0_40.0 }	>=		0 { 40.0 }	All	TRUE
66	Pack_1_la	prioArr	{ 0x13 }	{ 0x13_6.0 }	>=		0 { 6.0 }	All	TRUE
67	Pack_1_la	prioArr	{ 0x13 || 0x10 }	{ 0x13_6.0 || 0x10_1.0 }	>=		0 { 6.0 || 1.0 }	All	TRUE
68	Pack_1_la	prioArr	{ 0x13 || 0x10 || 0x20	{ 0x13 6.0 || 0x10 1.0 || 0x20 8.0 }	>=		0 { 6.0 || 1.0 || 8.0 }	All	TRUE

The following image shows the input CSV file for different types of devices that can be used with the diagnostic block.

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE	REFERENCE VARIABLE
MC_DRAM	DRAM_00_Total_Requests	>	10	Max	
MC_DRAM	DRAM_10_Max_Queue_Usage	>	10	Max	
AXI_Top	Master_1_Read_Data_MBps	<	30	Min	
AXI_Top	Slave_1_BW_Utilization_Prct	>	1	Standard Deviation	
Cache_D1	Cache_A_Hit_Ratio	<	80	All	
Cache_D1	Cache_A_Hit_Ratio	>	75	Mean	
Processor	Context_Switch_Time_Pct_Min	>	1	Standard Deviation	
Processor	Context_Switch_Time_Pct_StDev	>	1	Mean	
Processor	Task_Delay_Max	>	1	Max	
Bus_1	Input_Buffer_Occupancy_in_Words	>	1	Max	
Bus_1	Input_Buffer_Occupancy_in_Words	>	1	Min	
Pack_1_lat_09	prioArr	>=	{0x13 = 5,0x15 = 6, default= 5}	All	id_arr
RR1	InThru	>=	{0=10000;default=1000}	All	
RR2	InThru	>=	1.00E+03	All	
RR2	throughputArr	>	3.00E+07	Mean	
Ingress1	Latency	>	2.00E-05	Mean	
Ingress1	Buffer_Occupancy	>	3	Max	
Ingress2	Latency	<	2.00E-05	Max	
Server_Resource_Allocation	Time_in_Block	>	1	Mean	
Server_Resource_Allocation	Buffer_Occupancy	>	1	Max	
Server_Resource_Allocation	Utilization	>	1	Min	

Hardware Devices

Script

Queue

Server

7.3.6 Reference Variable:

Used only for an array variable monitoring.
Example: Latency array variable only stores the latency value, but to correlate it with id array we need to specify the id array.

For an array, if we don't specify the Reference variable, then the array index will be used for correlation.

BLOCK	METRICS	CONSTRAI	CONSTRAINT VALUE	STATISTIC	REFERENCE VARIABLE
Pack_1_la	prioArr	>=	{0x13 = 5,0x15 = 6, default= 5}	All	id_arr
Buff_1_ch_	arr	>=	0	Min	
Buff_1_ch_	idx	>=	0	Mean	

Input File

BLOCK	VARIABLE	REFERENCE VALUE	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAINT	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
Pack_1_la	prioArr	{ 0x13 }	{ 0x13_6.0 }	>=	0	{ 6.0 }	All	TRUE
Pack_1_la	prioArr	{ 0x13 || 0x10 }	{ 0x13_6.0 || 0x10_1.0 }	>=	0	{ 6.0 || 1.0 }	All	TRUE
Pack_1_la	prioArr	{ 0x13 || 0x10 || 0x20	{ 0x13_6.0 || 0x10_1.0 || 0x20_8.0 }	>=	0	{ 6.0 || 1.0 || 8.(	All	TRUE
Pack_1_la	prioArr	{ 0x13 || 0x10 || 0x20	{ 0x13_6.0 || 0x10_1.0 || 0x20_8.0 || 0x14_1.0 }	>=	0	{ 6.0 || 1.0 || 8.(	All	TRUE

Output File

MATCHED CONSTRAINT column will list out the values which pass the threshold condition (*only for **array***).

Format:

{<array index (if reference variable column is empty) or reference array corresponding value>_<variable value which we are monitoring> || <>_<> || .....}

7.4 Examples for Resource Block

7.4.1 Metrics: Time_in_Block; Statistic Type: All

Input File:

BLOCK	METRICS	CONSTRAI	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Time_in_Block	>	2	All

Outputs the details of all the data packets which have Time_in_Block **> 2**

Output File:

GENERATED_TIME	BLOCK	SOURCE	ID	Buffer_Occupancy	Buffer_Size	Time_in_Block	Utilization_Mean
3.5	Smart_Res	Traffic	12	{2 4 0 1 0}	{30 30 30 30 30}	2.4	0
3.9	Smart_Res	Traffic	13	{1 3 0 1 2}	{30 30 30 30 30}	2.7	0
6.3	Smart_Res	Traffic	36	{0 8 0 2 1}	{30 30 30 30 30}	2.8	0
6.9	Smart_Res	Traffic	44	{0 9 0 3 2}	{30 30 30 30 30}	2.6	0
7.3	Smart_Res	Traffic	46	{0 8 0 5 4}	{30 30 30 30 30}	2.8	0

7.4.2 Metrics: Time_in_Block; Statistic Type: Max/Min/Mean

Input File:

BLOCK	METRICS	CONSTRAI	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Time_in_Block	>=	3.05	Mean

Outputs a single value - Mean/Max/Min, for the Time_in_Block of all the data packets and gives a Result - TRUE, if the value satisfies the given constraint ,else FALSE

Output File:

BLOCK	METRICS	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
Smart_Res	Time_in_Block	3.035140997830801	>=	3.05	Mean	FALSE

7.4.3 **Metrics:** Utilization**; Statistic Type: All**

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Utilization	>=	0	All

Outputs the details of all the data packets which have **Utilization >= 0**

Output File:

GENERATE	BLOCK	SOURCE	ID	Buffer_Oc	Buffer_Size	Time_in_Bl	Utilization_Mean
0	Smart_Res	Traffic	1	{0 0 0 0 0	{30 30 30	0	0
0.1	Smart_Res	Traffic	2	{0 0 0 0 0	{30 30 30	0	0
0.2	Smart_Res	Traffic	3	{0 0 0 0 0	{30 30 30	0	0
0.4	Smart_Res	Traffic	5	{0 1 0 0 0	{30 30 30	0	0
0.6	Smart_Res	Traffic	7	{0 2 0 0 0	{30 30 30	0	0
0.7	Smart_Res	Traffic	8	{0 2 0 0 0	{30 30 30	0	0

7.4.4 **Metrics:** Utilization; **Statistic Type: Max/Min/Mean**

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Utilization	>=	3.05	Max

Outputs a single value - Mean/Max/Min, for the **Utilizations** of all the data packets and gives a Result - TRUE, if the value satisfies the given constraint ,else FALSE

Output File:

BLOCK	METRICS	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
Smart_Res	Utilization	0	>=	3.05	Max	FALSE

7.4.5 **Metrics:** Buffer Occupancy; **Statistic Type: All**

Input File: Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Buffer_Occupancy	>=	3.05	All

Outputs the details of all the data packets which have
Buffer_Occupancy >= 3.05

In this case the Buffer_Occupancy is an array. So, we check if any element of the array has a value **>= 3.05**. If it does, we print the details of that packet.
Output File:

GENERATED_TIME	BLOCK	SOURCE	ID	Buffer_Occupancy	Buffer_Size	Time_in_Block	Utilization_Mean
1.3	Smart_Res	Traffic	14	{0 5 0 0 0}	{30 30 30 30 30}	0	0
1.5	Smart_Res	Traffic	16	{1 5 0 0 0}	{30 30 30 30 30}	0	0
1.6	Smart_Res	Traffic	17	{1 5 0 0 0}	{30 30 30 30 30}	0	0
1.6	Smart_Res	Traffic	4	{1 4 0 0 0}	{30 30 30 30 30}	1.3	0
1.8	Smart_Res	Traffic	18	{2 4 0 0 0}	{30 30 30 30 30}	0.1	0
1.9	Smart_Res	Traffic	20	{2 4 0 0 0}	{30 30 30 30 30}	0	0
2.1	Smart_Res	Traffic	22	{3 4 0 0 0}	{30 30 30 30 30}	0	0

7.4.6 **Metrics:** Buffer Occupancy; **Statistic Type: Max/Min/Mean**

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Buffer_Occupancy	>=	3.05	Max

Outputs a single value - Mean/Max/Min, for the **Buffer_Occupancies** of all the data packets and gives a Result - TRUE, if the value satisfies the given constraint, else FALSE

Output File:

BLOCK	METRICS	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC	RESULT
Smart_Res	Buffer_Occupancy	20	>=	3.05	Max	TRUE

7.4.7 **Metrics:** Time_in_Block, Utilization, Buffer Occupancy**; Statistic Type:** All

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Buffer_Occupancy	>	25	All
Smart_Res	Utilization	>=	0	All
Smart_Res	Time_in_Block	>	2	All

Outputs the details of all the data packets which have satisfy **anyone** of the given constraints
Output File:

GENERATED_TIME	BLOCK	SOURCE	ID	Buffer_Occupancy	Buffer_Size	Time_in_Block	Utilization_Mean
0	Smart_Res	Traffic	1	{0 0 0 0 0}	{30 30 30 30 30}	0	0
0.1	Smart_Res	Traffic	2	{0 0 0 0 0}	{30 30 30 30 30}	0	0
0.2	Smart_Res	Traffic	3	{0 0 0 0 0}	{30 30 30 30 30}	0	0
0.4	Smart_Res	Traffic	5	{0 1 0 0 0}	{30 30 30 30 30}	0	0
0.6	Smart_Res	Traffic	7	{0 2 0 0 0}	{30 30 30 30 30}	0	0
0.7	Smart_Res	Traffic	8	{0 2 0 0 0}	{30 30 30 30 30}	0	0
0.9	Smart_Res	Traffic	10	{1 2 0 0 0}	{30 30 30 30 30}	0	0

7.4.8 Metrics: Time_in_Block, Utilization, Buffer Occupancy, Statistic Type: Max/Min/Mean

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
Smart_Res	Buffer_Occupancy	>	3.05	Min
Smart_Res	Utilization	>=	3.05	Mean
Smart_Res	Time_in_Block	>	3.05	Max

Outputs a single value for each Stats_Name, for all the data packets and gives Result - TRUE, if the value satisfies the given constraint else, FALSE
Output File:

BLOCK	METRICS	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
Smart_Res	Buffer_Occupancy	0	>	3.05	Min	FALSE
Smart_Res	Utilization	0	>=	3.05	Mean	FALSE
Smart_Res	Time_in_Block	9.199999999999996	>	3.05	Max	TRUE

7.5 Examples for Script Block –

7.5.1 Metrics: Block variable, Data Type: Int/Double, Statistics Type: All, Reference Variable: -

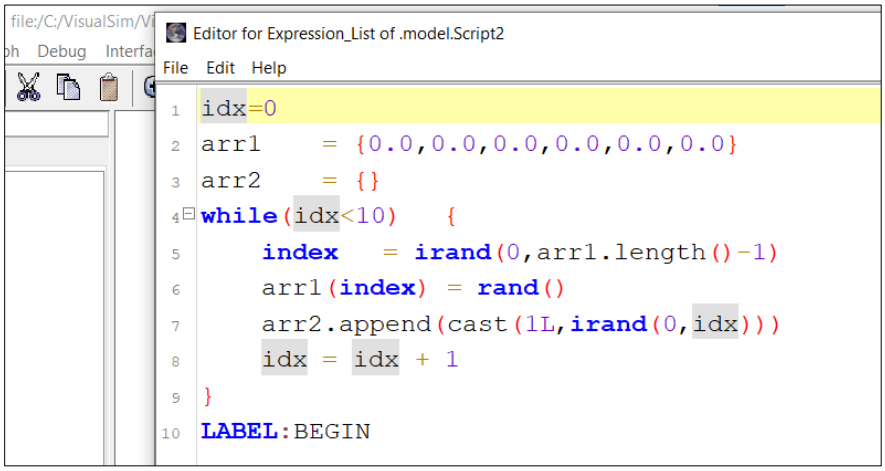
Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
S2_Test_C	idx	>=	0	All
S2_Test_C	index	>=	0	All

Outputs all the values stored by each **Stats_Name (idx and index,** in this case) that passes the given constraint, for the entire simulation.

So here we get all the values of **idx** which are >=0 and all the values of **index** which are >=0
Output File:

5	BLOCK	VARIABLE	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
12	S2_Test_C	index	4	>=	0	All	TRUE
13	S2_Test_C	index	0	>=	0	All	TRUE
14	S2_Test_C	index	1	>=	0	All	TRUE
15	S2_Test_C	index	3	>=	0	All	TRUE
16	S2_Test_C	idx	0	>=	0	All	TRUE
17	S2_Test_C	idx	1	>=	0	All	TRUE
18	S2_Test_C	idx	2	>=	0	All	TRUE



S2_Test_Case

7.5.2 Metrics: Block variable, Data Type: Array, Statistics Type: All, Reference Variable: -

Input File:

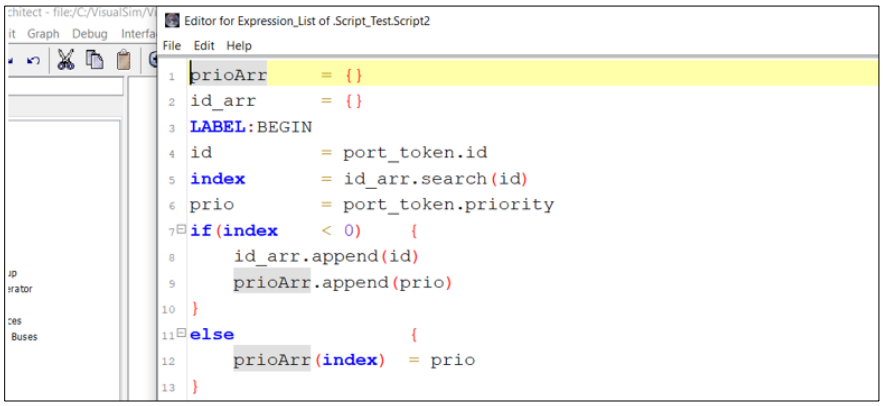
BLOCK	METRICS	CONSTRAI	CONSTRAI	STATISTIC
Pack_1_la	prioArr	>=	5	All

Outputs all the values stored by each **Stats_Name (prioArr,** in this case) if any one of the values present in the array passes the given constraint, for the entire simulation.

So here we get all the values of **prioArr** in which
any one element of the array is >= 5.

We get all the values(of the array) which passes the constraint in the **MATCHED CONSTRAINT** column with its *index* or *correlated* value as explained in the **Columns Description section** above.
Output File: Output File:

BLOCK	VARIABLE	REFERENCE VALUE	MATCHED CONSTRAIN	CONSTRAI	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
Pack_1_lat	prioArr		{ 0_6.0 }	>=	5	{ 6.0 }	All	TRUE
Pack_1_lat	prioArr		{ 0_6.0 }	>=	5	{ 6.0 || 1.0 }	All	TRUE
Pack_1_lat	prioArr		{ 0_6.0 || 2_8.0 }	>=	5	{ 6.0 || 1.0 || 8.0 }	All	TRUE
Pack_1_lat	prioArr		{ 0_6.0 || 2_8.0 }	>=	5	{ 6.0 || 1.0 || 8.0 |	All	TRUE
Pack_1_lat	prioArr		{ 0_6.0 || 1_6.0 || 2_	>=	5	{ 6.0 || 6.0 || 8.0 |	All	TRUE
Pack_1_lat	prioArr		{ 0_6.0 || 1_6.0 || 2_	>=	5	{ 6.0 || 6.0 || 8.0 |	All	TRUE
Pack_1_lat	prioArr		{ 0_6.0 || 1_6.0 || 2_	>=	5	{ 6.0 || 6.0 || 8.0 |	All	TRUE



Pack_1_lat_09

7.5.3 Metrics: Block variable, Data Type: Int/Double & Array, Statistics Type: All, Reference Variable: -

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
S2_Test_C	arr1	>=	0	All
S2_Test_C	idx	>=	0	All
S2_Test_C	index	>=	0	Min
S1_Test_C	arr	>=	0	Mean
S1_Test_C	idx	>=	0	Max

Outputs a single value - Avg/Max/Min, for each **Stats_Name** for the entire simulation and gives Result - TRUE if the value satisfies the given constraint, else FALSE

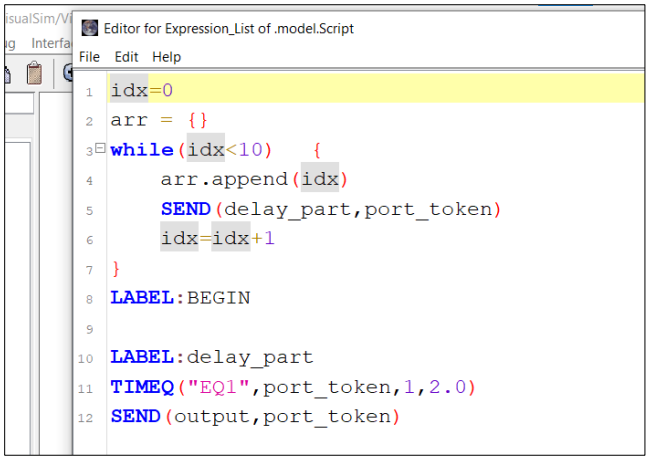
Output Files:

BLOCK	VARIABLE	REFERENC	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAI	CONSTRAI	MEASURED_VALUE	STATISTIC TYPE	RESULT
S2_Test_C	arr1		{ 0_0.0 || 1_0.0 || 2_0.0 || 3_0.0 || 4_0.0 ||	>=	0	{ 0.0 || 0.0 || 0.0 || 0.0 || 0.0 || 0.0 }	All	TRUE
S2_Test_C	arr1		{ 0_0.0 || 1_0.0 || 2_0.6513255814883855 ||	>=	0	{ 0.0 || 0.0 || 0.6513255814883855 || 0.0	All	TRUE
S2_Test_C	arr1		{ 0_0.3330171718873298 || 1_0.0 || 2_0.6513	>=	0	{ 0.3330171718873298 || 0.0 || 0.6513255	All	TRUE
S2_Test_C	arr1		{ 0_0.3330171718873298 || 1_0.613104986831	>=	0	{ 0.3330171718873298 || 0.61310498683185	All	TRUE
S2_Test_C	arr1		{ 0_0.3330171718873298 || 1_0.613104986831	>=	0	{ 0.3330171718873298 || 0.61310498683185	All	TRUE
S2_Test_C	arr1		{ 0_0.3330171718873298 || 1_0.613104986831	>=	0	{ 0.3330171718873298 || 0.61310498683185	All	TRUE

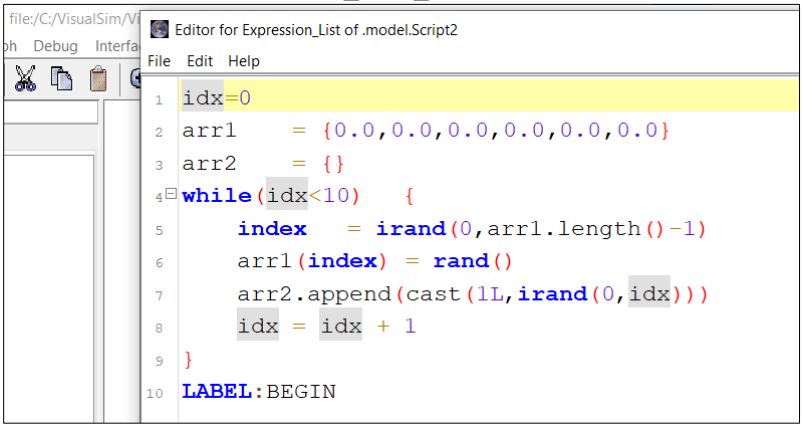
Array File

BLOCK	VARIABLE	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
S1_Test_C	idx	10	>=	0	Max	TRUE
S2_Test_C	index	0	>=	0	Min	TRUE
S2_Test_C	idx	0	>=	0	All	TRUE
S2_Test_C	idx	1	>=	0	All	TRUE
S2_Test_C	idx	2	>=	0	All	TRUE
S2_Test_C	idx	3	>=	0	All	TRUE
S2_Test_C	idx	4	>=	0	All	TRUE

Non-Array File



S1_Test_Case



S2_Test_Case

7.5.4 Metrics: Block variable, Data Type: Int/Double & Array, Statistics Type:All, Reference Variable: id_arr

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE	REFERENCE VARIABLE
Pack_1_la	prioArr	>=	0	All	id_arr
Buff_1_ch_	arr	>=	0	All	
Buff_1_ch_	idx	>=	0	All	

Outputs all the values stored by each **Stats_Name** for the entire simulation based on the respective method for **Int/Double/Array** data types, as explained above.

Here we have an additional **REFERENCE** column for **Arrays.** It gives the value stored by the corresponding **Reference** array at the same time.

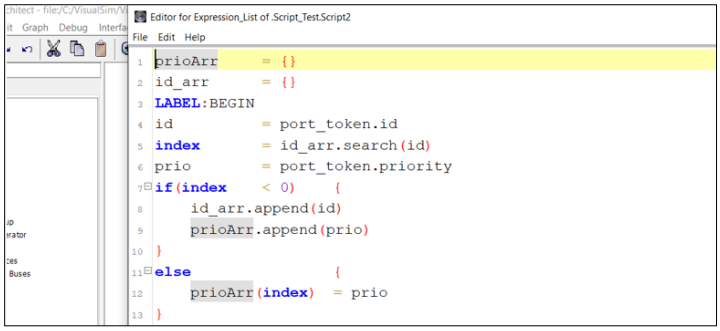
So, when **prioArr** stored {6.0}, **id_arr** stored {0x13} etc.
Output Files:

BLOCK	VARIABLE	REFERENC	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAINT	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
Pack_1_lat	prioArr	{ 0x13 }	{ 0x13_6.0 }	>=	0	{ 6.0 }	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_6.0 || 0x10_1.0 }	>=	0	{ 6.0 || 1.0 }	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_6.0 || 0x10_1.0 || 0x20_8.0 }	>=	0	{ 6.0 || 1.0 || 8.0 }	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_6.0 || 0x10_1.0 || 0x20_8.0 || 0x14_1.	>=	0	{ 6.0 || 1.0 || 8.0 || 1.0 }	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_6.0 || 0x10_6.0 || 0x20_8.0 || 0x14_1.	>=	0	{ 6.0 || 6.0 || 8.0 || 1.0 }	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_6.0 || 0x10_6.0 || 0x20_8.0 || 0x14_1.	>=	0	{ 6.0 || 6.0 || 8.0 || 1.0 || 4.0 }	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_6.0 || 0x10_6.0 || 0x20_8.0 || 0x14_1.	>=	0	{ 6.0 || 6.0 || 8.0 || 1.0 || 4.0 ||	All	TRUE
Pack_1_lat	prioArr	{ 0x13	|| { 0x13_5.0 || 0x10_6.0 || 0x20_8.0 || 0x14_1.	>=	0	{ 5.0 || 6.0 || 8.0 || 1.0 || 4.0 ||	All	TRUE

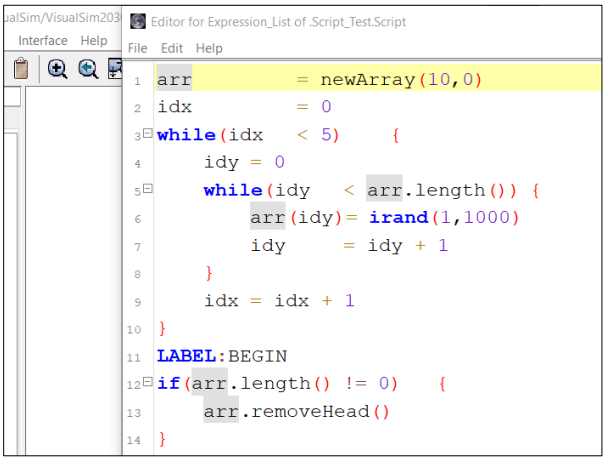
Array File

BLOCK	VARIABLE	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC TYPE	RESULT
Buff_1_ch_	idx	0	>=	0	All	TRUE
Buff_1_ch_	idx	1	>=	0	All	TRUE
Buff_1_ch_	idx	2	>=	0	All	TRUE
Buff_1_ch_	idx	3	>=	0	All	TRUE
Buff_1_ch_	idx	4	>=	0	All	TRUE
Buff_1_ch_	idx	5	>=	0	All	TRUE

Non - Array File



Pack_1_lat_09



Buff_1_ch_2

7.5.5 Metrics: Block variable, Data Type: Int/Double & Array, Statistics Type: Max,Min,Mean , Reference Variable: id_arr

Input File:

BLOCK	METRICS	CONSTRAI	CONSTRAI	STATISTIC	REFERENCE	VARIABL
Pack_1_la	prioArr	>=	0	Mean	id_arr	
Buff_1_ch_	arr	>=	0	Mean		
Buff_1_ch_	idx	>=	0	Mean		

Outputs a single value - Mean/Max/Min, for each **Stats_Name** for the entire simulation, along with the **CORRELATE** column and gives, Result - TRUE if the value satisfies the given constraint, else FALSE

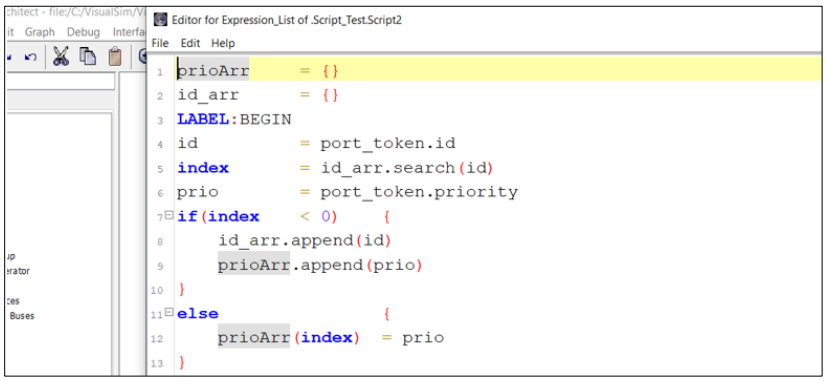
Output Files:

BLOCK	VARIABLE	REFERENC	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAINT	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
Buff_1_ch_	arr		{ 0_441.4 || 1_465.0 || 2_415.846153846153	>=	0	{ 441.4 || 465.0 || 415.846153846153	Mean	TRUE
Pack_1_la	prioArr	{ 0x13 ||	{ 0x13_4.571428571428571 || 0x10_4.6 || 0x	>=	0	{ 4.571428571428571 || 4.6 || 6.5 ||	Mean	TRUE

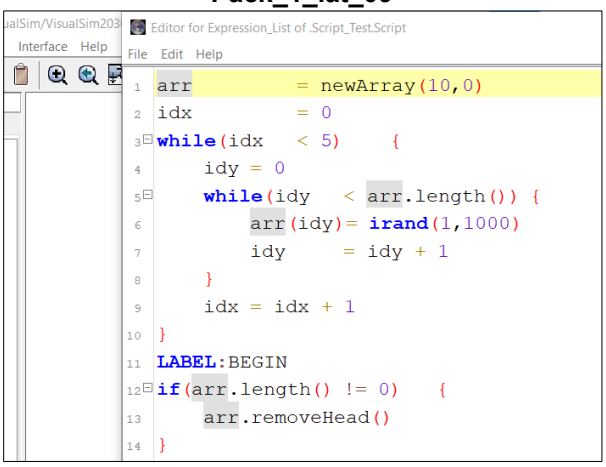
Array File

BLOCK	VARIABLE	MEASURE	CONSTRAI	CONSTRAI	STATISTIC	RESULT
Buff_1_ch_	idx	2.5	>=	0	Mean	TRUE

Non-Array File



Pack_1_lat_09



Buff_1_ch_2

7.5.6 Metrics: Block variable, Data Type: Array, Statistics Type: Max,Min,Mean , Reference Variable: id_arr

Input File:

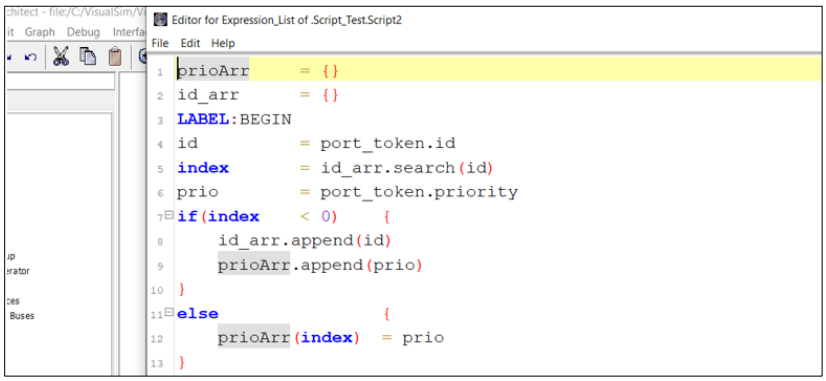
BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE	REFERENCE VARIABLE
Pack_1_lat	prioArr	>=	{0x13 = 2; 0x10 = 1; 0x20 = 2; 0x14 = 1;default = 10}	Mean	id_arr
Buff_1_ch_	arr	<	{0 = 500; 1 = 450; 2 = 500; 3 = 400; default = 600}	Mean	

Outputs a single value - Mean/Max/Min, for each **Stats_Type** for the entire simulation, along with the **REFERENCE** column and gives, Result - TRUE if the value satisfies the given constraint else FALSE

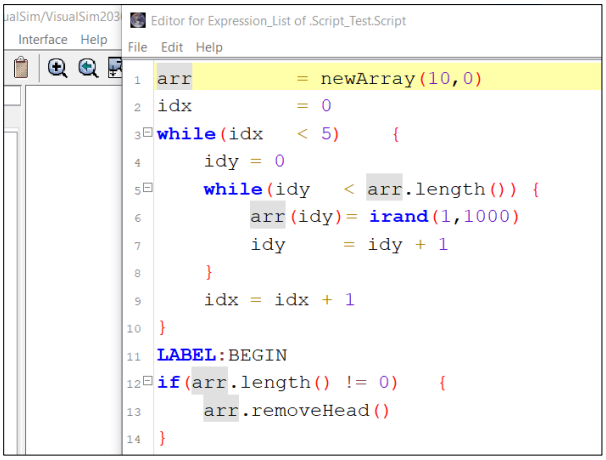
For the **Arrays,** you can enter the **value** in the **structured** form as explained in the **Column Description section.**

Output File:

BLOCK	VARIABLE	REFERENCE VALUE	MATCHED CONSTRAINT(<Reference>_<Value>)	CONSTRAI	CONSTRAINT_VALUE	MEASURED_VALUE	STATISTIC TYPE	RESULT
Buff_1_ch_	arr		{ 0_441.4 || 2_415.84615384615387 || 3_382.	<	{0 = 500; 1 = 450; 2 = 500; 3	{ 441.4 || 465.0 ||	Mean	TRUE
Pack_1_lat	prioArr	{ 0x13 || 0x10 || 0x	{ 0x13_4.571428571428571 || 0x10_4.6 || 0x2	>=	{0x13 = 2; 0x10 = 1; 0x20 =	{ 4.571428571428571	Mean	TRUE



Pack_1_lat_09



Buff_1_ch_2

7.6 Examples of Hardware blocks:

The hardware blocks can be used with diagnostic block by adding the requirements based on the statistics provided by each. The statistics of the blocks observed through the architecture setup are eligible to use with the diagnostics feature.

The Block column contains the block name of the device. The Metrics column contains the statistic name that can be observed through the architecture setup port. For example Hit ratio of Cache_D1 will be displayed as “Cache_D1_Cache_A_Hit_Ratio”, user must enter the stats name without the block name, such as “Cache_A_Hit_Ratio”.

The Constraint column determines the condition that user want to evaluate. The constraint value is the threshold value which is compared with the values obtained based on the stats type.

The Stats type define the type of comparison, for example Max defines the comparison will be made with constraint value and the maximum value of the stats that is obtained.

User can use any block variable of hardware devices such as the block variable in the script block. Check section 7.5 above for more details.

Input File:

BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE
MC_DRAM	DRAM_00_Total_Requests	>	10	Max
MC_DRAM	DRAM_10_Max_Queue_Usage	>	10	Max
MC_DRAM	DRAM_30_DRAM_Delay_Mean	>	2.50E-08	Max
MC_DRAM	DRAM_02_Total_MB_per_Second	<	30	Max
AXI_Top	Master_1_Read_Data_Bytes	>	300	Max
AXI_Top	Master_1_Read_Data_MBps	<	30	Min
AXI_Top	Slave_1_BW_Utilization_Prct	>	1	Standard Deviation
AXI_Top	Slave_1_Read_Data_Bytes	>	300	Mean
AXI_Top	Slave_1_Read_Data_MBps	<	30	Max
AXI_Top	Slave_1_Rd_Threshold_Usage	>	1	Min
AXI_Top	Slave_1_Rd_Transactions	>	1	Standard Deviation
Cache_D1	Cache_A_Hit_Ratio	<	80	All
Cache_D1	Cache_A_Hit_Ratio	>	75	Mean
Cache_D1	Cache_A_Miss_Ratio	>	1	Max
Cache_D1	Cache_A_Prefetch_Issued	>	1	Min

Once the simulation is over user can get the result folders according to the blocks used in the design. The detailed description about the folders and files are given in the next section (Output File).

The results folder will contain the detail for the constraints listed in the requirement file as shown below.

BLOCK	METRICS	MEASURED_VALUE	CONSTRAINT	CONSTRAINT_VALUE	STATISTIC	RESULT
MC_DRAM	DRAM_00_Total_Requests	56	>	10	Max	TRUE
MC_DRAM	DRAM_10_Max_Queue_Usage	4	>	10	Max	FALSE
MC_DRAM	DRAM_30_DRAM_Delay_Mean	1.6250607142857E-8	>	2.50E-08	Max	FALSE
MC_DRAM	DRAM_02_Total_MB_per_Second	35.85221055	<	30	Max	FALSE
AXI_Top	Master_1_Read_Data_Bytes	3584	>	300	Max	TRUE
AXI_Top	Master_1_Read_Data_MBps	35.84000036	<	30	Min	FALSE
AXI_Top	Slave_1_BW_Utilization_Prct	0	>	1	Standard D	FALSE
AXI_Top	Slave_1_Read_Data_Bytes	3584	>	300	Mean	TRUE

NOTE:
a) The contents of the CSV file specified are *CASE SENSITIVE* except the ***Type*** column.
b) The columns should be entered in the same order as shown above, in the Input.csv file.
c) The ***"Statistics Type"*** should be the 5th column in the input file with any one of the following values :

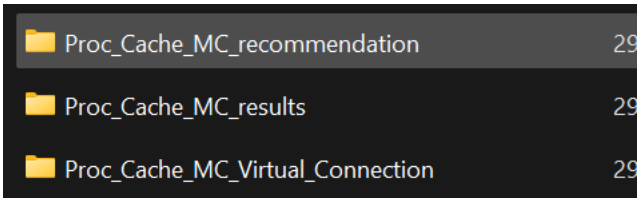
i) **All -** Outputs all the values for the *Stats_name* which pass the given constraint.
ii) **Mean -** Outputs a single *Average* value for all the values stored in *Stats_name*.
iii) **Min -** Outputs a single *Minimum* value for all the values stored in *Stats_name*.
iv) **Max -** Outputs a single *Maximum* value for all the values stored in *Stats_name*.

d) ***"Reference variable"*** is an **optional** column, should be added after the ***"Type"*** column.
e) The format shown in the last example for *Script Block* for comparing **array** values for the index/correlated value comparison needs to be followed strictly. Any deviation from the given format will lead to errors.

7.7 Output File

The following are the possible output files that can be generated by the diagnostic block
1)Recommendation
2) Result

3) Virtual Connection



The folders and files are generated according to the blocks used in the model. The recommendation and results folder will be generated for most of the block, but the Virtual connection folder will be generated only if there are blocks that support virtual packet transfer.

The recommendation folder will have EOS file, recommendation file, and Script Analysis file. The EOS fill provides the buffer usage of the resource blocks present in the model.

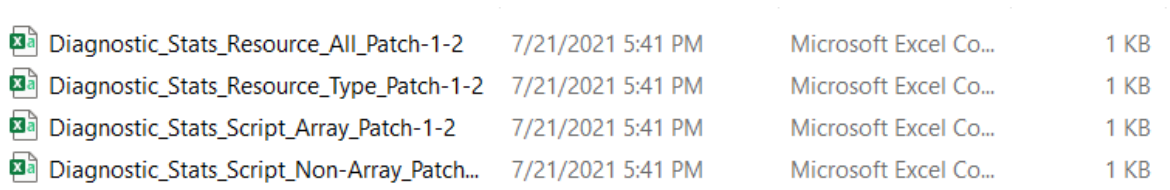
The recommendation file provides useful suggestions such as the buffer entries are not used in the simulation, or a specific resource is not sending anything at its output port. It captures any anomaly if present, inside the block and It will be useful for the designer to find any bottleneck in the design.
if any of the block name specified in the CSV file doesn't match the Block_Name parameter of particular resource block then no constraint evaluation will be performed for that block.

The Script analysis file provides you the list of parameters, variables in the scipts, the type of the variable such as global or local or block variable and also the usage of the variable like whether that variable is being used or not. Along with that it provides some consolidated stats such as the total number of variables in a script, number of if, for and while are executed in the script, total number of lines in the script, and number of statements executed and not executed.

The Virtual connection folder contains a file that list out the virtual connections that sent packets from one part of the device to another.

```
Block_Recommendation            0.0 ps

Queue Number 1, 2, 4, 5 of Smart_Resource is not being used at 0.0 ps
Queue Number 1, 2, 4, 5 of Smart_Resource is not sending out any transaction. Check the pop or the priority of the queue to make sure the logic is correct at 0.0 ps
Queue Number 1, 2, 5 of Smart_Resource is not being used at 100.0000000000 ms
Queue Number 1, 2, 5 of Smart_Resource is not sending out any transaction. Check the pop or the priority of the queue to make sure the logic is correct at 100.000000
Queue Number 2, 5 of Smart_Resource is not being used at 200.0000000000 ms
Queue Number 2, 5 of Smart_Resource is not sending out any transaction. Check the pop or the priority of the queue to make sure the logic is correct at 200.000000000
Queue Number 5 of Smart_Resource is not being used at 400.0000000000 ms
Queue Number 5 of Smart_Resource is not sending out any transaction. Check the pop or the priority of the queue to make sure the logic is correct at 400.0000000000 m
The data from Queue Number 1 is going out slower than it is entering in Smart_Resource at 15.0000000000000 sec
The data from Queue Number 1 is going out slower than it is entering in Smart_Resource at 18.0000000000000 sec
The data from Queue Number 1, 2, 3, 4 is going out slower than it is entering in Smart_Resource at 21.0000000000000 sec
The data from Queue Number 1, 2, 3, 4, 5 is going out slower than it is entering in Smart_Resource at 24.0000000000000 sec
```

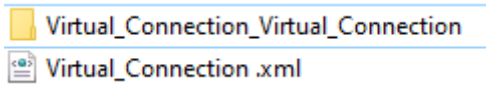
Result file will contain the statistics from the resource block & script blocks (according to the given constraints in the Input file) :



i) **Diagnostic_Stats_Resource_All_<Model Name>** - Contains statistics for the Resource Blocks with 'All' in the *"Type"* column.
ii) **Diagnostic_Stats_Resource_Type_<Model Name>** - Contains statistics for the Resource Blocks with 'Max / Min / Avg' in the *"Type"* column.
iii) **Diagnostic_Stats_Script_Array_<Model Name>** - Contains statistics for the Script Block's variables of 'Array' data type.
iv) **Diagnostic_Stats_Script_Non_Array_<Model Name>** - Contains statistics for the Script Block's of 'Int / Double' data type.

7.8 Virtual Connection in Diagnostic Block

- *How to use?*
  - Drag and Drop Diagnostic block from Library.
  - Run the model.
  - Save the model to generate Output file.
  - Output file demo:



  - The .txt file will contain the list of block sending data from *Source* to *Destination*

```
Packet sent from Source .Virtual_Connection_.Script to Destination .Virtual_Connection_.Script2 ( Value : Processor_Script )
Packet sent from Source .Virtual_Connection_.ExpressionList to Destination .Virtual_Connection_.Script ( Value : MyMachine )
Packet sent from Source .Virtual_Connection_.ExpressionList3 to Destination .Virtual_Connection_.Script2 ( Value : Processor_Script )
Packet sent from Source .Virtual_Connection_.ExpressionList3 to Destination .Virtual_Connection_.Script ( Value : MyMachine )
```

- *Supported block*
  - IN
  - OUT
  - MUX
  - DEMUX
  - Script
  - Expression List

Chapter 3 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 **Queue** blocks, **Server** 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.

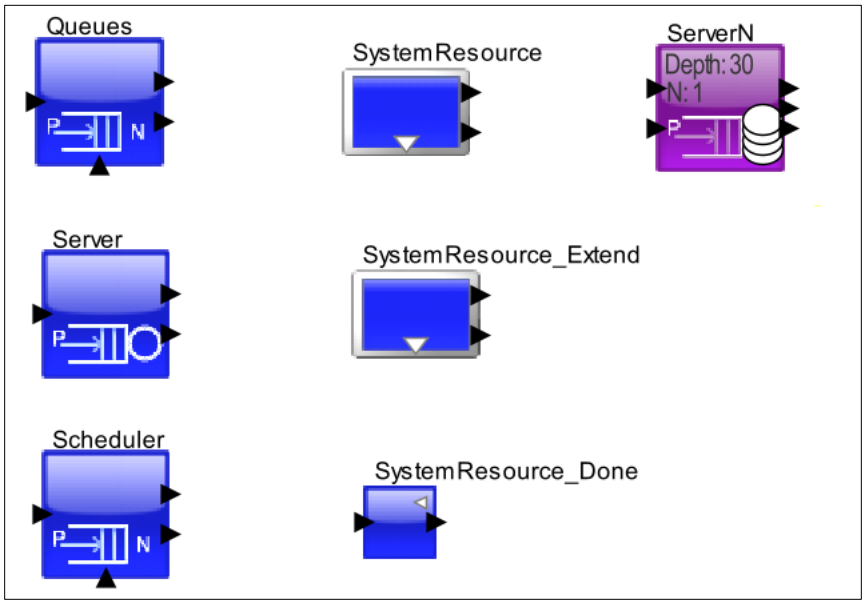


Figure 35 VisualSim Resource Library

Queues (a.k.a Smart_Resource)

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.

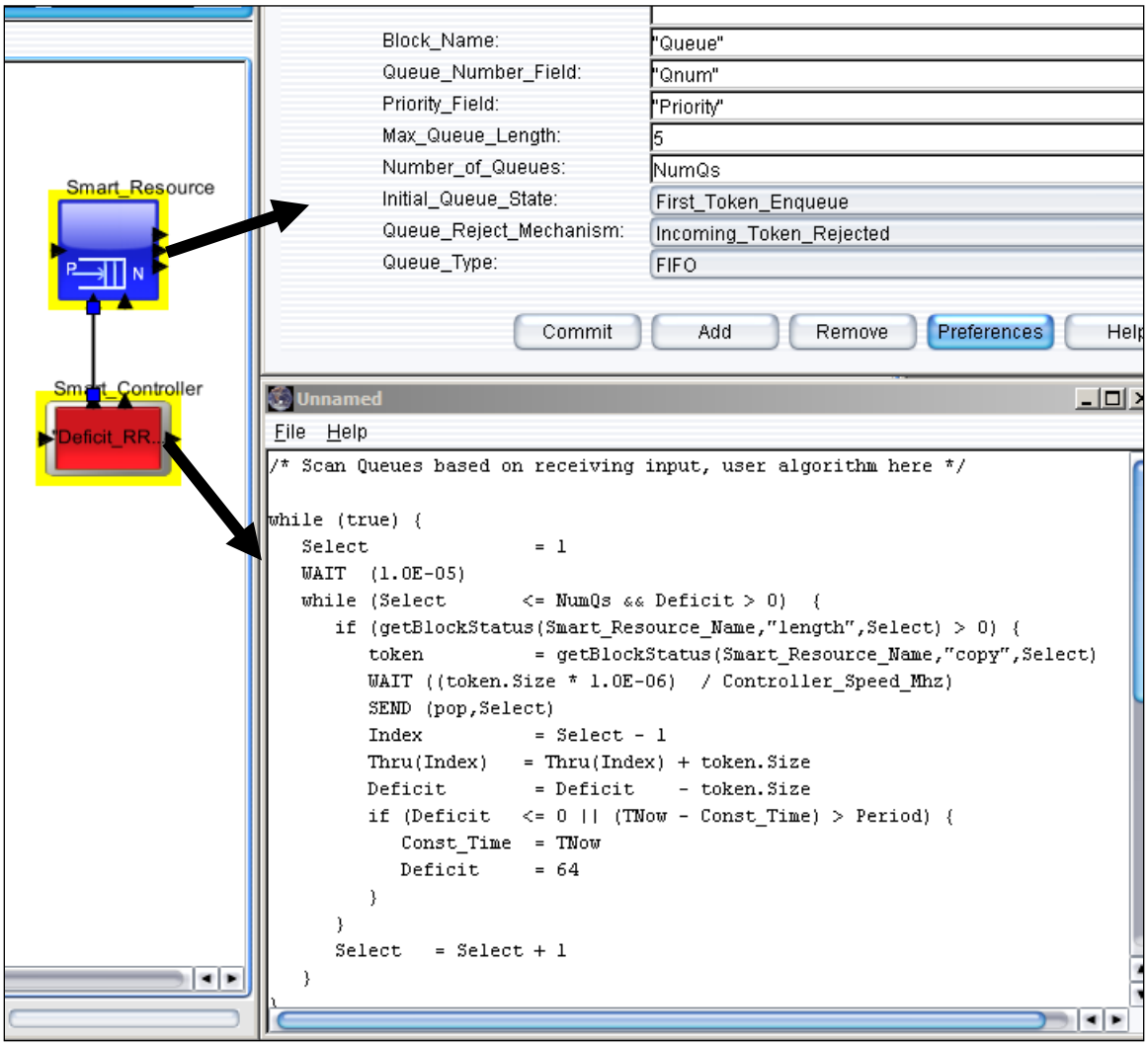


Figure 36 Controller Library blocks

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.

ServerN

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 this block, view the Block documentation by right click on the block and select Documentation->Get Documentation. Try out the demo model listed in the document to clearly understand the functionality. The required inputs must all exist before the Data Structure will enter the queue.

Scheduler

This special block has inbuilt scheduling and queue management algorithm with queue structure. The incoming packets are allocated to the appropriate queue by referring the queue number field parmater. When there is a pop from the external trigger source the scheduler selects the queue according to the scheduling algorithm and other necessary configurations. It supports the following scheduling algorithms:

1. FCFS
2. FIFO
3. LIFO
4. Round Robin
5. Weighted Round Robin
6. Weighted Fair Queueing
7. Strict Priority
8. Deficit Round Robin
9. Round Robin Priority

It supports the following queue management algorithms:

1. Tail Drop / Incoming Token Rejected
2. Random Drop on Full
3. Drop Front on Full
4. Random Early Detection
5. Lowest Priority Token Rejected

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 task can be assigned to the system resource or System Resource Extend by one of the 3 mappers available in VisualSim.

1. Mapper
2. Software Mapper
3. Dynamic Mapper

Any one of these mapper can be used to allocate the target resource for the incoming request, But the Software mapper and Dynamic mapper has special ability than the normal mapper.

Software Mapper can enable Mutual Exclusion on a task which is assigned to a shared resource that can process multiple task from different mappers. When a particular task is enabled with Mutual Exclusion this will prevent all other tasks from preempting this task.

Dynamic mapper can perform fixed or dynamic allocation of resources based on the configuration. The important parameters are Database_lookup, Task_Name, Target_Resource and Database_Expression. Other parameters can be configured as regular mapper. If Database_lookup is “None” then user have to put direct value or field names to allocate the resource, which will be a fixed mapping to a target resource.
If Database reference is configured, then user can have 3 types of mapping

1. Fixed mapping – user can still use the filed name or parameter in Target_Resource parameter.
2. Dynamic mapping through database Task_Name – the Task Name must be a field and it must be listed in the database, the Target_Resource must be the destination column name.
3. Dynamic mapping through Regex – User can define the regex condition in Database_Expression parameter and it will be linked to Target_Resource parameter.

For more details about each mapper block please refer the block documentation and the demo model listed in them.

The System Resource blocks, namely **SystemResource_Extend** 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 **SystemResourcce** Block can be arranged as a hierarchical set of schedulers. The **SystemResource_Extend** 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** Block. A **Mapper** Block can contain values in the parameter fields that effect scheduler operation, or the names of data structure fields entering the **Mapper** 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 a **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** Block in the model can address any scheduler in the model. Multiple **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 a

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 a **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 than 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.

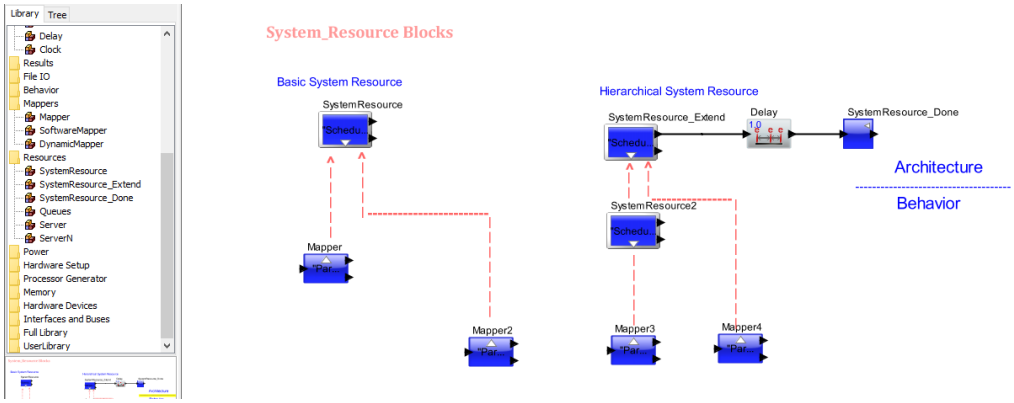


Figure 37 VisualSim Scheduler Library

Key Difference between SystemResource and SystemResource_Extend

Features	SystemResource	SystemResource_Extend
Preemption	Yes (Pre-emption is not allowed if there are more than two schedulers in a hierarchy or the SystemResource_Extend is extended using the “output_port” port.)	No
Hierarchical	Yes	No
Extended Task Processing	No	Yes
Non-blocking scheduling	No	Yes
	Can reference another scheduler	Cannot reference and execute in the next-level scheduler

Caveat: A SystemResource can support preemption only when there are zero or one SystemResource above this one.

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-tranmissions 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).

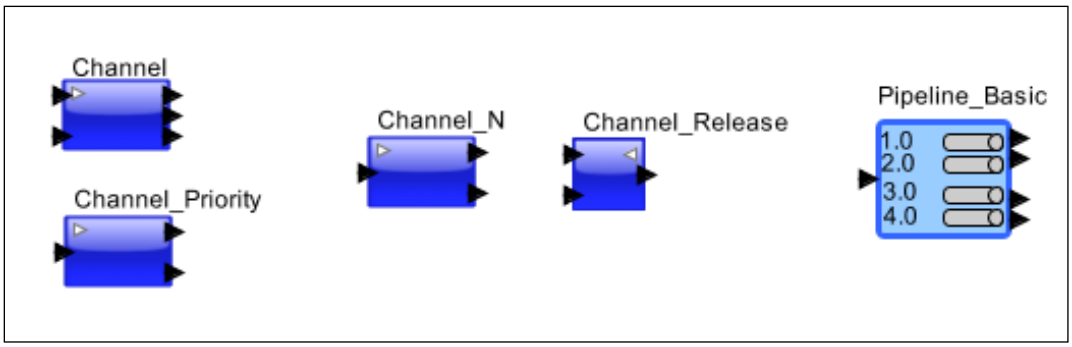


Figure 38 VisualSim Channel and Pipeline block

Passive or Quantity-Shared Resource Blocks

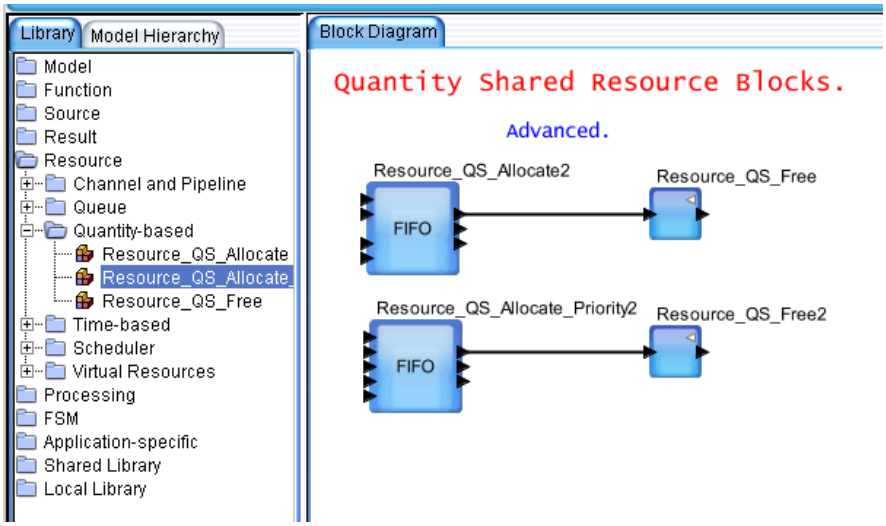


Figure 39 VisualSim Quantity-Shared Resource Library

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.

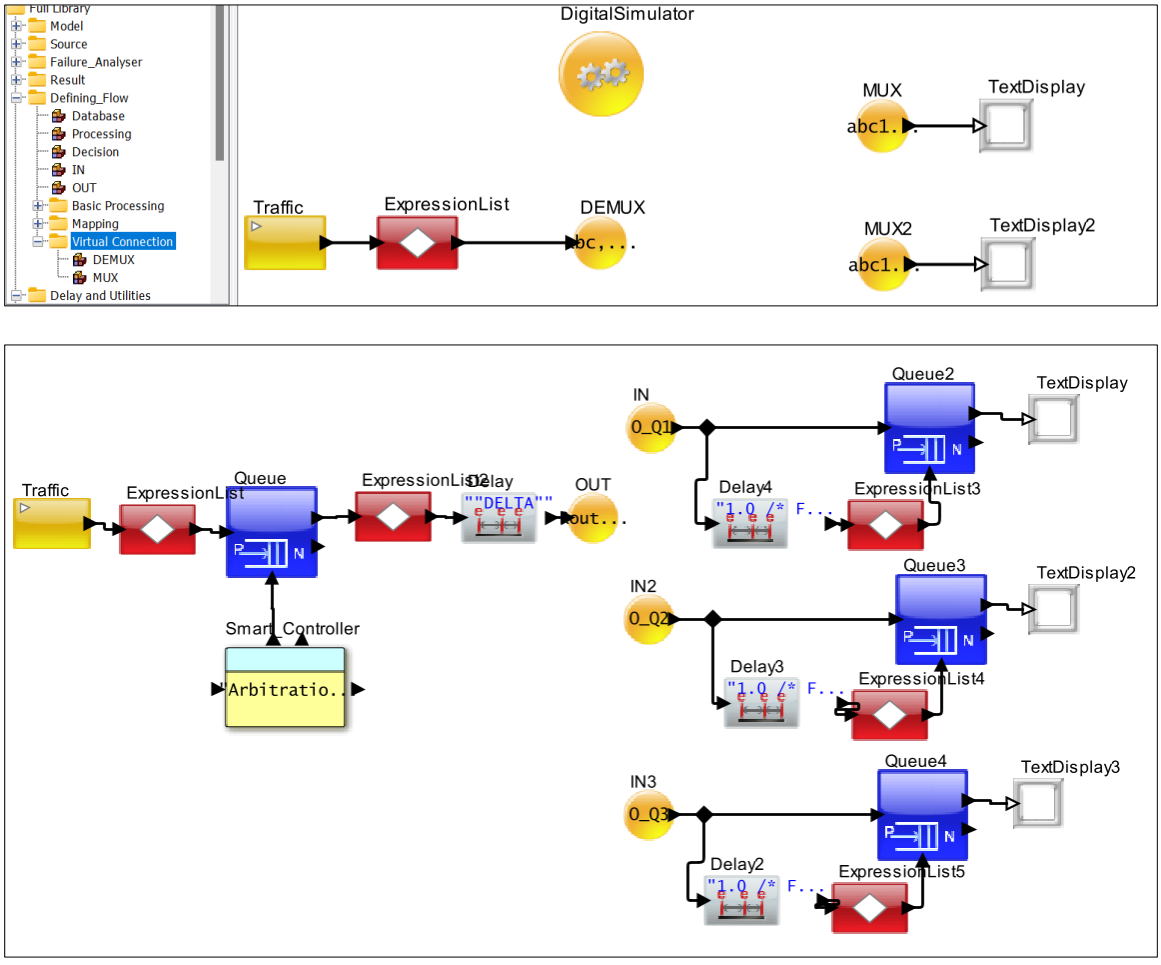


Figure 40 VisualSim Virtual Connection blocks (Virtual Connections Library)

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.

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.

Source Blocks

Source

User can generate traffic using a standard distribution, sequences, trace files, clocks, and custom distributions.

- **Clock** – Generate all possible clock inputs.
- **Event** – Generate the input according to the Event set.
- **File Import** – Input can be generated according to the file input.
- **Traffic** – Generate input as per the specified DS and time distribution.

Generating

The **Source** blocks generate the data structures that trigger the model to execute and flows through the model. There are five mechanisms for generating Data Structures in VisualSim.

- In case of using a standard distribution for the time interval between Data Structures, use Traffic.
- If a custom distribution is required or the Data Structure is generated as a function of another activity, or triggered during the flow, use the TriggeredTraffic.
- If the distribution is a specific sequence or the generation rate is a function of time and probability, use the Transaction_Sequence block.
- If a file contains the sequence from an existing system, such as a trace or Network Sniffer or variable activity, then use the Traffic_Reader or File Reader blocks to generate the Data Structure. In this case, the columns of the trace file are the fields and the values in each line form the new Data Structure. The File Reader loads the entire file into computer memory before generating values while the Traffic_Reader reads one line at a time from the file.
- If a custom traffic with pause time is needed the Custom_Traffic block can be used. Number of Transactions defined in the parmeter will be generated with the Time Interval parameter. Once the total transactions are generated then it will pause for a time mentioned in Time_Pause parameter.
- If a software trace from a cycle accurate model or external simulator needs to be assigned for a hybrid processor then Trace_Mapper can be used. It will add the necessary fields to the each line read from the trace and send it to the processor, once the current request is completed it will read the next line add the fields and send it to the processor until the end of the trace.

Math Functions and Distribution

This library section contains the mathematical operations and distributions as library blocks where the user can provide inputs in the required format and the expected result can be obtained as the output. It includes array, Boolean, logical, distribution, transform and string operations.

Array Libraries

1. **ArrayAppend:**
   Library that appends two Arrays together. This library has a single input multiport, and a single output port. The types on the input and the output port must both be the same array type. During each input, this block reads up to one ArrayToken from each channel of the input port and creates an ArrayToken of the same type on the output port. If no token is available on a particular channel, then there will be no contribution to the output.

2. **ArrayAverage**
   Compute the average of the elements in an array. This block reads an array from the *input* port and sends the average of its elements to the *output* port. The output data type is at least the type of the elements of the input array.

3. **ArrayElement**
   Extract an element from an array. This block reads an array from the *input* port and sends one of its elements to the *output* port. The element that is extracted is determined by the *index* parameter (or port). It is required that 0 <= *index* < *N*, where *N* is the length of the input array, or an exception will be thrown.

4. **ArraySetElement**
   This block requires an array on the input port. The block replaces the value at a designated location in the array with the new value. The designated location is provided at the index port. The new value is povided on the "element" input ports. The resulting array is send out on the output port. It is required that 0 <= index < N, where N is the length of the input array, or an exception will be thrown.

5. **ArrayExtract**
   Extract a subarray from an array. This block reads an array from the *input* port and sends a subarray to the *output* port, possibly padded with zeros. The segment of the input array starting at *sourcePosition* with length *extractLength* is copied to the output array, starting at *destinationPosition*. The total length of the output array is *outputArrayLength*. Any of its entries that are not supplied by the input have value zero (of the same type as the entries in the input array). With the default values of the parameters, only the first element of the input array is copied to the output array, which has length one.

6. **ArrayLength**
   Output the length of an array. This block reads an ArrayToken from the *input* port and sends its length to the *output* port.

7. **ArrayLevelCrossing**
   Search an array from the specified starting index and report the index of the first item in the array that is below or above the specified threshold. If there is no such item, then -1 is returned. The threshold can be absolute or relative to the value at the starting index. If it is relative, it can be given on a linear scale or in decibels. If the threshold is relative and we are looking for values above the threshold, then values that are above the value at the starting index by more than the threshold are reported. If the threshold is relative and we are looking for values below the threshold, then values that are below the value at the starting index by more than the threshold are reported.

8. **ArrayMaximum**
   Extract the maximum element from an array. This block reads an array from the *input* port and sends the largest of its elements to the *output* port. The index of the largest element is sent to the *index* output port. If there is more than one entry in the array with the maximum value, then the index of the first such entry is what is produced.

9. **ArrayMinimum**
   Extract the minimum element from an array. This block reads an array from the *input* port and sends the largest of its elements to the *output* port. The index of the smallest element (closest to minus infinity) is sent to the *index* output port. If there is more than one entry in the array with the minimum value, then the index of the first such entry is what is produced.

10. **ArrayPeakSearch**
    This block outputs the indices and values of peaks in an input array. The dip and squelch parameters control the sensitivity to noise. These are given either as absolute numbers or as relative numbers. If they are absolute numbers, then a peak is detected if a rise above dip is detected before the peak and a dip below dip is detected after the peak. If they are given as relative numbers, then a peak is detected when a rise by a factor dip above the most recently seen minimum (if there has been one) is seen before the peak, and if a dip by a factor dip relative to the peak is seen after the peak. Relative numbers can be either linear (a fraction) or in decibels. This is determined by the value of the scale parameter. For example, if dip is given as 2.0 and scale has value "relative linear", then a dip must drop to half of a local peak value to be considered a dip. If squelch is given as 10.0 and scale has value "relative linear", then any peaks that lie below 1/10 of the global peak are ignored.

11. **ArraySort**
    Sort the elements of an input array. This block reads an array from the *input* port and sends a sorted array to the *output* port. The input array is required to contain either strings or non-complex scalars, or a type error will occur. The output array will have the same type as the input and can be sorted in either ascending or descending order. Duplicate entries can be optionally removed.

Boolean Libraries:

1. **BooleanMultiplexor**
   This actor consumes exactly one token from each input port, and sends one of the tokens from either trueInput or falseInput to the output. The token sent to the output is determined by the select input, which must be a boolean value. Because tokens are immutable, the same Token is sent to the output, rather than a copy. The trueInput and falseInput port may receive Tokens of any type.

2. **BooleanSelect**
   In an iteration, if an input token is available at the *control* input, that token is read, and its value is noted. Its value specifies the input port that should be read next. If the *control* input is true, then if an input token is available on the *trueInput* port, then it is is read and sent to the output. Likewise with a false input and the *falseInput* port. The token sent to the output is determined by the *control* input, which must be a boolean value. Because tokens are immutable, the same Token is sent to the output, rather than a copy. The *trueInput* and *falseInput* port may receive Tokens of any type.

3. **BooleanSwitch**
   In an iteration, if an input token is available at the *control* input, that token is read, and its

value is noted. Its value specifies the input port that should be read next. If the *control* input is true, then if an input token is available on the *input* port, then it is is read and sent to the *trueOutput*. Likewise with a false input and the *falseOutput* port. Because tokens are immutable, the same Token is sent to the output, rather than a copy. The *input* port may receive Tokens of any type. If no token has ever been received on the *control* port, then *falseOutput is assumed to be the one to receive data.*

4. **CountTrues**
   It reads the given number of input booleans and output the number that are true

Counter Libraries:

1. **Counter_Basic**
   This block is a counter that is incremented or decremented based on the input value. The outputs are determined by the parameter settings.

   This block has four input ports. The incr_add_input port increments, or adds integer values to the counter state. The decr_sub_input port decrements, or subtracts integer values from the counter state. The reset_set_input port resets, or sets a new counter state. The test_input port always generates a value on the counter_output port.

   The equals_output generates an integer if the counter state equals the parameter 'Equals_Count".

   The increment, decrement, reset functions are controlled by the "Input_Port_Mode". The "Input_Port_Mode" can be "Increment" which is equivalent to "Increment_Decrement by a value of 1", the default setting, or can be value of "ADD" which is equivalent to "Add_Subtract the value in the appropriate input port".

   The "Counter_Mode" can keep count without a reset ("Ignore_Final_Count"), or in circular fashion ("Final_Count_to_Initial_Count").

   The output can generate counts on each change of the internal state ("Normal_Counter_Update"), or only generate output tokens to the counter_output with the "Output_Mode" setting "Via_Test_Port."

2. **Counter_Loop_K**
   This is a counter that increments or decrements until it reaches a the 'final value'. There are two models of operation for this block. The Do settings get the Final Value from the 'init' port. The Parameter Valid (Above) gets the final value from the Parameter. The increment or decrement of any value.

   The 'init' port is required to initialize the block for all mode of operations, to reset the counter and to restart the loop count. The ready_input is the trigger to modify the counter value. For every input on the ready_input port, the current value of the counter is placed on 'counter_output' port. When the loop count reaches the final value, this value is placed on the 'counter_output' and the 'done_output' port. In the case of DO(0,N-1), the final value will be one value less than the initial value set.
   The block has three modes of operation:
   1. "Do (1 , N)": In this mode, the value on the 'init' is the final value. The initial value is 1 and the increment value is the Increment_Value. The 'init' is required to start the loop and to reset it for every new loop. The counter value is incremented by the Increment_Value when a token arrives on the ready_input. When the loop is

completed, the 'counter_output' and done_output receive the last value. This can be fed back to the 'init' to restart the loop.

2. "Do (0 , N-1)": In this mode, one minus the value on the 'init' is the final value. The initial value is 0 and the increment value is the Increment_Value. The 'init' is required to start the loop and to reset it for every new loop. The counter value is incremented by the Increment_Value when a token arrives on the ready_input. When the loop is completed, the 'counter_output' and the done_output receives the **final value which is N-1**. This can be fed back to the 'init' to restart the loop.
3. Above Parameters Valid: This uses the initial, final and increment values in the block. The 'init' is required to start the loop and to reset it for every new loop. The counter value is incremented by the value arriving on the ready_input. When the loop is completed, the'counter_output' and the done_output receive the last value. This token can be fed back to the 'init' to restart the loop. The Final Value remains unchanged.

3. **Counter_Loop_Port**
   This block contains a counter that is incremented or decremented, whenever a value is received on the ready_input port. The "ready_input" triggers the loop increment (or decrement) and outputs the loop counter value to the "counter_output", unless the "final_input" value is exceeded in either a positive, or negative direction. Once the "final_input" value is exceeded, then the "done_output" is activated. The initial counter, final counter and increment (or decrement) counter values are sent by initializing this block with values on the three input ports: *initial_input*, *final_input*, *increment_input*. When done_output is activated, the internal counter is reset to the initial value.

Distribution Libraries:

1. **Bernoulli**
   Produce a random sequence of booleans. The output is of type BooleanToken. The values that are generated are independent and identically distributed, where the probability of *true* is given by the parameter *trueProbability*. The seed can be specified as a parameter to control the sequence that is generated.

2. **Beta**
   Produce a random sequence with a Beta distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

3. **Binomial**
   Produce a random sequence with a Binomial distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

4. **BreitWigner**
   Breit-Wigner is a also know as the Lorentz distribution. The Breit-Wigner distribution is a more generate form of the Cauchy distribution. Produce a random sequence with a BreitWigner distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In

addition, the seed can be specified as a parameter to control the sequence that is generated

5. **ColtChiSquare**
   The Chi Square Distribution is a special case of the Gamma distribution. Produce a random sequence with a ChiSquare distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the freedom and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

6. **DiscreteRandomSource**
   It produces tokens with a given probability mass function. The probability mass function is a parameter, *pmf*, of this actor. The *pmf* must be an array that contains entries that are all between 0.0 and 1.1, and sum to 1.0. By default, *pmf* is initialized to {0.5, 0.5}. Output values are selected at random from the *values* parameter, which contains an ArrayToken. This array must have the same length as *pmf*. Thus the *i*-th token in *values* has probability *pmf*[*i*]. The output port has the same type as the elements of the *values* array. The default *values* are {0, 1}, which are integers.

7. **Exponential**
   Produce a random sequence with a Exponential distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the lambda and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

8. **ExponentialPower**
   Produce a random sequence with a ExponentialPower distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the tau and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

9. **Gamma**
   Produce a random sequence with a Gamma distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

10. **Gaussian**
    Produce a random sequence with a Gaussian distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

11. **HyperGeometric**
    Produce a random sequence with a HyperGeometric distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

12. **Logarithmic**
    Produce a random sequence with a Logarithmic distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the p and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated

13. **NegativeBinomial**
    Produce a random sequence with a NegativeBinomial distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

14. **Normal**
    Produce a random sequence with a Normal distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

15. Poisson
    Produce a random sequence with a Poisson distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

16. **PoissonSlow**
    Produce a random sequence with a PoissonSlow distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

17. **Rician**
    Produce a random sequence with a Rician distribution. A Rician random variable is defined as follows: Let Z = sqrt(X^2 + Y^2), where X and Y are statistically independent Gaussian random variables with means given by parameters *xMean* and *yMean* respectively, and common variance given by parameter *standardDeviation*.
    The default values of *xMean* and *yMean* are both set to be zero, in which the distribution is also called a Rayleigh distribution. Hence, the actor is by default a Rayleigh random generator.
    On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the means and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

18. **StudentT**
    Produce a random sequence with a StudentT distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the freedom and the standard

deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

19. **Uniform**
    Produce a random sequence with a uniform distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with bounds defined by the parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

20. **VonMises**
    Produce a random sequence with a VonMises distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the freedom and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

21. **Zeta**
    Produce a random sequence with a Zeta distribution. On each iteration, a new random number is produced. The output port is of type DoubleToken. The values that are generated are independent and identically distributed with the mean and the standard deviation given by parameters. In addition, the seed can be specified as a parameter to control the sequence that is generated.

Logic Libraries:

1. **AND**
   Used to combine model inputs, passes three inputs to a single output, based on last input to arrive (passes data structure to output). All three inputs must be present to create AND condition.

2. **Comparator**
   Compare two double-valued inputs, and output the boolean result of the comparison. The exact comparison performed is given by the *comparison* attribute, which can take any of the following values:
   a. **>**: *left* > *right*
   b. **>=**: *left* >= *right*
   c. **<**: *left* < *right*
   d. **<=**: *left* <= *right*
   e. **==**: *left* == *right*

   The default is ">". The input ports are named *left* and *right* to indicate which side of the comparison operator their value appears on.

   The *tolerance* parameter, which defaults to zero, defines an error tolerance. That is, the actor may produce true even if the specified test is not exactly satisfied, but rather is almost satisfied, within the specified tolerance.

3. **Compare_K**
   A compare_output by constant K, which routes input tokens to the true_output port, if the comparison is true, else the input token is sent to the false_output. This actor has one input port, the *input* port for incoming data tokens. In addition, the value of the comparison is sent to the compare_output port, represented as '1' equals true, '0' equals false.

4. **Compare_Port**
   A compare_output by two inputs, which routes input tokens to the true_output port, if the comparison is true, else the input token is sent to the false_output. This actor has two input ports, In addition, the value of the comparison is sent to the compare_output port, represented as '1' equals true, '0' equals false.

5. **EIO**
   This block is being deprecated. Use the Connect_EIO or the Pipeline blocks. The Execute in Order (EIO) block can be used to order the internal execution. A copy of the Data Structure arriving on the input port is first placed on 'output' port, then on 'output2' and then on 'output3'. The output tokens or data structures are simply copies of the input that are placed on each port in order.j

6. **Equals**
   A logical equals operator. This operator has one input multiport and one output port that is not a multiport. It will consume at most one token from each input channel, and compare the tokens using the isEqualTo() method of the Token class. If all observed input tokens are equal, then the output will be a true-valued boolean token. If there is not at least one token on the input channels, then no output is produced. The type of the input port is undeclared and will be resolved by the type resolution mechanism. Note that all input channels must resolve to the same type. The type of the output port is boolean.

7. **IsPresent**
   On each firing, output true if the input is present and false otherwise. The type of the output port is boolean, and the input is general. The width of the input is expected to match the width of the output. Note that the utility of this actor varies by domain. In PN, for example, the input is always present (by definition). In SDF, it is expected to be always present, but may not be. In DE, the actor is only triggered if one of the input channels has data. The actor is probably most useful in synchronous simulators like SR and Giotto.

8. **LogicalNot**
   This actor implements a logical NOT operator. It has one input and one output port, neither of which is a multiport, and both of which have type boolean. A BooleanToken that arrives in the *input* will be negated and sent on the *output*. If no input token is available, then no output is produced.

9. **LogicFunction**
   Produce an output token on each firing with a value that is equal to the specified logic operator of the input(s). The functions are:
   a. **and**: The logical and operator. This is the default function for this actor.
   b. **or**: The logical or operator.
   c. **xor**: The logical xor operator.
   d. **nand**: The logical nand operator. Equivalent to the negation of *and*.
   e. **nor**: The logical nor operator. Equivalent to the negation of *or*.
   f. **xnor**: The logical xnor operator. Equivalent to the negation of *xor*.

NOTE: All operators have a single input port, which is a multiport, and a single output port, which is not a multiport. All ports have type boolean.
This actor does not require that each input channel have a token upon firing. As long as one channel contains a token, output will be produced. If no input tokens are available at all, then no output is produced. At most one token is consumed on each input channel.

Math and Trig Libraries:

1. **AbsoluteValue**
   Produce an output token on each firing with a value that is equal to the absolute value of the input. The input can have any scalar type. If the input type is not Complex, the output has the same type as the input. If the input type is Complex, the output type is Double, in which case, the output value is the magnitude of the input complex.

2. **AddSubtract**
   This adder has two input ports, both of which are multiports, and one output port, which is not. The types on the ports are undeclared and will be resolved by the type resolution mechanism. Data that arrives on the input port named *plus* will be added, and data that arrives on the input port named *minus* will be subtracted. Any token type supporting addition and subtraction can be used. In most simulators, either input port can be left unconnected. Thus, to get a simple adder (with no subtractor), just leave the *minus* input unconnected.
   The *plus* input port will typically resolve to the least upper bound of the types presented to it. Thus, for example, if one input channel comes from a source of type BooleanToken and another comes from a source of type IntToken, the resolved type will be StringToken, and addition will be that implemented in StringToken (which concatenates strings). Notice that StringToken does not support subtraction, so if any inputs are presented to the *minus* port, an exception will be thrown at run time.
   Currently, the type system is quite liberal about the resolved types it will permit at the inputs. In particular, it may permit the *plus* and *minus* inputs to resolve to types that cannot in fact be subtracted. In these cases, a run-time error will occur. In the future, we hope that the type system will intercept such errors before run time.
   This actor does not require that each input channel have a token upon firing. It will add or subtract available tokens at the inputs and ignore the channels that do not have tokens. It consumes at most one input token from each port. If no input tokens are available at all, then no output is produced.

3. **Accumulator**
   Output the initial value plus the sum of all the inputs since the last time a true token was received at the reset port. One output is produced each time the actor is fired. The inputs and outputs can be any token type that supports addition. The output type is constrained to be the greater than or equal to the input type and the initial value type.

4. **Average**
   Output the average of the inputs after the last time a true token is received at the reset port. One output is produced each time the actor is fired. The inputs and outputs can be any token type that supports addition and division by an integer. The output type is constrained to be the same as the input type.

5. **Const**
   Produce a constant value on the output port everytime the input is triggered. The value of the output is what is contained by the *value* parameter, which by default is an Integer with value 1. The type of the output is that of *value* parameter. This block can accept the full range of the Expression language as value of the *value* parameter.
   The input can be any value that triggers the *value* parameter to be computed and placed on the output port. The output is of type General.

6. **StringConst**
   Produce a constant output of type string. This is similar to the base class Const, which can also produce a string output, but this has the added convenience that the string can be specified without the enclosing double quotes. Moreover, the string can include

references to parameters within scope using the $name syntax. The value of the output is that of the token contained by the *value* parameter, which by default is an empty string.

7. **Differential**
   Output the current input minus the previous input, or if there has been no previous input, the current input itself

8. **DotProduct**
   Compute the dot product of two arrays or matrices. This actor has two input ports, from which it receives two ArrayTokens or two Matrix Tokens. The elements of the ArrayTokens or MatrixTokens must be of type ScalarToken. The output is the dot product of the two arrays or matrices.

9. **Limiter**
   Produce an output token on each firing with a value that is equal to the input if the input lies between the *bottom* and *top* parameters. Otherwise, if the input is greater than *top*, output *top*. If the input is less than *bottom*, output *bottom*. This actor operates on doubles only.

10. **MathFunction**
    Produce an output token on each firing with a value that is equal to the specified math function of the input. The input and output types are DoubleToken. The functions are a subset of those in the java.lang.Math class. They are:
    a. **exp**: The exponential function. This is the default function for this actor If the argument is NaN, then the result is NaN.
    b. **log**: The natural logarithm function. If the argument is NaN, then the result is NaN.
    c. **modulo**: The modulo after division. If the second operand is zero, then the result is NaN.
    d. **sign**: If the argument is greater than 0, return 1.0, if it is less than 0, return -1.0, otherwise return 0.0.
    e. **square**: The square function If the argument is NaN, then the result is NaN.
    f. **sqrt**: The square root function. If the argument is NaN, then the result is NaN.
    NOTES: 1. Some functions like exp, log, square, and sqrt act on a single operand only. Other functions like modulo act on two operands. The actor acquires a second input when the function is changed to modulo, and loses the input when the function is changed back. 2. There is an alternative to using the MathFunction.modulo() method If you want to use the IEEE remainder standard, use the Remainder actor.

11. **Math_K**
    A compare_output by constant K, which routes input tokens to the output port, if the comparison is true, else the input token is sent to the false_output. This actor has one input port, the *input* port for incoming data tokens. In addition, the value of the comparison is sent to the compare_output port, represented as '1' equals true, '0' equals false.

12. **Math_Port**
    A compare_output by two inputs, which routes input tokens to the output port, if the comparison is true, else the input token is sent to the false_output. This actor has two input ports, In addition, the value of the comparison is sent to the compare_output port, represented as '1' equals true, '0' equals false.

13. **Maximum**

Read at most one token from each input channel and broadcast the one with the greatest value to the *maximumValue* output. In addition, broadcast the channel number of the maximum on the *channelNumber* output port. Either output port may be left unconnected if you do not need its results (this is why these are multiports). This actor works with any scalar token. For ComplexToken, the output is the one with the maximum magnitude. The input port is a multiport.

14. **Max_by_Port**
    Maximum of input port tokens. This actor has two input ports, *a_input*, *b_input*, and one output port for the maximum data token.

15. **Minimum**
    Read at most one token from each input channel and broadcast the one with the least value to the *minimumValue* output. In addition, broadcast the channel number of the minimum on the *channelNumber* output port. Either output port may be left unconnected if you do not need its results (this is why these are multiports). This actor works with any scalar token. For ComplexToken, the output is the one with the minimum magnitude. The input port is a multiport.

16. **Min_by_Port**
    Minimum of input port tokens. This actor has two input ports, *a_input*, *b_input*, and one output port for the minimum data token.

17. **MultiplyDivide**
    This adder has two input ports, both of which are multiports, and one output port, which is not. The types on the ports are undeclared and will be resolved by the type resolution mechanism. Data that arrives on the input port named *multiply* will be multiplied, and data that arrives on the input port named *divide* will be divided. Any token type supporting multiplication and division can be used. In most simulators, either input port can be left unconnected. Thus, to get a simple multiplier (with no division), just leave the *divide* input unconnected.
    Currently, the type system is quite liberal about the resolved types it will permit at the inputs. In particular, it may permit the *multiply* and *divide* inputs to resolve to types that cannot in fact be multiplied or divided. In these cases, a run-time error will occur. In the future, we hope that the type system will intercept such errors before run time.
    This actor does not require that each input channel have a token upon firing. It will multiply or divide available tokens at the inputs and ignore the channels that do not have tokens. It consumes at most one input token from each port. If no input tokens are available on the *multiply* inputs, then a numerator of one is assumed for the division operations. The "one" is obtained by calling the one() method of the first token seen at the *divide* input. If no input tokens are available at all, then no output is produced.

18. **Quantizer**
    Produce an output token on each firing with a value that is a quantized version of the input. The input and output types are both double.
    The *levels* parameter contains an array of doubles specifying the quantization levels. The elements must be in an increasing order, or an exception will be thrown. The default value of *levels* is {-1.0, 1.0}.
    Suppose u is the input, and *levels* = {a, b, c}, where $a < b < c$, then the output of the actor will be:
    $y = a$, for $u <= (b+a)/2$;
    $y = b$, for $(b+a)/2 < u <= (c+b)/2$;
    $y = c$, for $u > (c+b)/2$;

Thus, for the default *levels*, the output is (almost) the signum function of the input, or +1.0 if the input is positive, and -1.0 otherwise. This is almost the signum function because it outputs -1.0 if the input is zero.
This actor does not require that the quantization intervals be equal, i.e. we allow that (*c*-*b*) != (*b*-*a*).

19. **Remainder**
    Compute the remainder after dividing the input by the divisor. The input and output data types are both double. This is implemented using the IEEEremainder() method of the java Math class, which computes the remainder as prescribed by the IEEE 754 standard. The method documentation states: "The remainder value is mathematically equal to f1 - f2 ? n, where n is the mathematical integer closest to the exact mathematical value of the quotient f1/f2, and if two mathematical integers are equally close to f1/f2, then n is the integer that is even. If the remainder is zero, its sign is the same as the sign of the first argument. Special cases:
    a. If either argument is NaN, or the first argument is infinite, or the second argument is positive zero or negative zero, then the result is NaN.
    b. If the first argument is finite and the second argument is infinite, then the result is the same as the first argument."

    Note: The divisor parameter is available as an input port in the MathFunction.Modulo() method. If you need to change the divisor during run-time, the MathFunction actor may be the a better choice.

20. **Scale**
    Produce an output token on each firing with a value that is equal to a scaled version of the input. The actor is polymorphic in that it can support any token type that supports multiplication by the *factor* parameter. If the input type is a scalar, the output type is constrained to be at least as general as both the input and the *factor* parameter; if the input is an array, the output is also an array with the elements scaled. The input can be an array of array, in which case the elements of the inner most array will be scaled. For data types where multiplication is not commutative (such as matrices), whether the factor is multiplied on the left is controlled by the *scaleOnLeft* parameter. Setting the parameter to true means that the factor is multiplied on the left, and the input on the right. Otherwise, the factor is multiplied on the right.

21. **TrigFunction**
    Produce an output token on each firing with a value that is equal to the specified trigonometric function of the input. The input and output types are DoubleToken. The functions are exactly those in the java.lang.Math class. They are:
    a. **acos**: The arc cosine of an angle, in the range from 0.0 through pi. If the argument is NaN or its absolute value is greater than 1, then the result is NaN.
    b. **asin**: The arc sine of an angle, in the range of -pi/2 through pi/2. If the argument is NaN or its absolute value is greater than 1, then the result is NaN. If the argument is positive zero, then the result is positive zero; if the argument is negative zero, then the result is negative zero.
    c. **atan**: The arc tangent of an angle, in the range of -pi/2 through pi/2. If the argument is NaN, then the result is NaN. If the argument is positive zero, then the result is positive zero; if the argument is negative zero, then the result is negative zero.
    d. **cos**: The trigonometric cosine of an angle. If the argument is NaN or an infinity, then the result is NaN.

e. **sin**: The trigonometric sine of an angle. If the argument is NaN or an infinity, then the result is NaN.
f. **tan**: The trigonometric tangent of an angle. If the argument is NaN or an infinity, then the result is NaN. If the argument is positive zero, then the result is positive zero; if the argument is negative zero, then the result is negative zero

22. UnaryMathFunction

Produce an output token on each firing with a value that is equal to the specified math function of the input. The input and output types are DoubleToken. The functions are a subset of those in the java.lang.Math class. They are:

1. **exp**: The exponential function. This is the default function for this actor If the argument is NaN, then the result is NaN.
2. **log**: The natural logarithm function. If the argument is NaN, then the result is NaN.
3. **sign**: If the argument is greater than 0, return 1.0, if it is less than 0, return -1.0, otherwise return 0.0.
4. **square**: The square function If the argument is NaN, then the result is NaN.
5. **sqrt**: The square root function. If the argument is NaN, then the result is NaN.

Parser Libraries:

1. Parse_File

This block converts a file containing strings at the location specified by 'File_Path' plus 'File_Name' into Strings. This block parses an existing text file by looking for spaces to delineate the String Tokens. The **Use Model** is to bring in an external trace file that can be used with 'Model_List' or 'Model_List_DS' to create an internal list. This list can be executed very fast in a simulation. The 'Parse_String' Block is similar to 'Parse_File' except that the 'Parse_String' block does not open a file. The 'Parse_String' accepts a string that was extracted using the 'File_to_String' Block.

When Parse_File receives an input in the open_file port, it opens the file named in the parameters and converst the content in the file to a string. When the parse_file port is activated, the string is parsed to match the list of keywords provided in the keyword parameter field. When a keyword is matched, the keyword followed by the string tokens are placed on the string_output. The keyword is sent to both the keyword port and string port, when matched. An End_Of_File marker is generated on the End-of-File output port once the entire file has been parsed. A trigger is required in the parse_file input for the next string_token to be matched and placed on the output ports

The trigger can be any value. The strings in the file must be separated by space to indicate the next string.

2. Parse_Keyword

This block converts an input string token, to one of the three outputs, based on the parameter keywords, and incoming keyword. If a keyword arrives, then all subsequent StringTokens will be sent to the same keyword output. The keyword will not be sent out of the block and acts as switch for incoming StringTokens. No output is generated till the first keyword arrives.

The **Use Model** is to detect sub-sequences from the 'Parse_File' or 'Parse_String' Blocks and to redirect seqences of Data Structures from a single file, or StringToken, into Model_List or Model_List_DS Blocks.

For example a file containing the following would be parsed as in the parathenses:
Header text .... ; name1 type1 value1 (';' is keyword 1, values directed to

'keyword_1_out') : name2 type2 value2 (':' is keyword 2, values directed to 'keyword_2_out') ; name3 type3 value3 (';' is keyword 1, again, values directed to 'keyword_1_out')

3. **Parse_String**
   This actor parses an incoming string at the 'string_in' port into Java String Tokens. Spaces are used to delineate the Java String Tokens.

   The **Use Model** is primarily bring in an incoming string, that can then be used with other blocks to create an internal 'Model_List' or 'Model_List_DS' that can be executed very fast in a simulation. The 'Parse_File' Block is similar to 'Parse_String' except it opens a file. One most likely will need to use a 'File_to_String' block to obtain the text file.

   When Parse_String receives an input in the string_in port, it converst the string into a Java String Token. When the parse_string port is activated, the string is parsed to match the list of keywords provided in the keyword parameter field. When a keyword is matched, the keyword followed by the string tokens are placed on the string_output. The keyword is sent to both the keyword port and string port, when matched. An End_Of_File marker is generated on the End-of-File output port once the entire file has been parsed. A trigger is required in the parse_file input for the next string_token to be matched and placed on the output ports
   The trigger can be any value. The strings in the file must be separated by space to indicate the next string.

Transform Libraries:

Basic:

1. **AnythingToDouble**
   Convert an integer-array into a string. Uses only the low order byte from each integer. NOTE: Assumes an 8-bit character set. The output is a string assembled from these bytes. This actor is designed to facilitate use of the SerialComm serial communication actor which uses the same kind of integer array format as this actor. Datagram actors can use this format as well.

2. **BooleanToAnything**
   This actor converts a boolean input token into any data type. A *true* at the input results in an output with value given by the *trueValue* parameter. A *false* at the input results in an output with value given by the *falseValue* parameter.

3. **BitsToInt**
   This actor converts a sequence of BooleanTokens into a single IntToken. The number of Boolean tokens is specified by the *numberOfBits* parameter and should be a positive integer not larger than 32. Let *k* denotes the value of the *numberOfBits* parameter. The output integer is ranged from -2^k to $2^k - 1$. The first boolean token received indicates the sign of the integer. If it is "false", the output integer is a non-negative number. If it is "true", the output integer is a negative number. The least significant bit is the last boolean token received.

4. **IntToBits**
   This actor converts an IntToken into a sequence of Boolean tokens. The number of Boolean tokens is specified by the *numberOfBits* parameter. It should be a positive integer not bigger than 32. The most significant bit (the sign bit) is the first boolean token send out. It is "false" if the input integer is non-negative, otherwise it is "true". The least

significant bit is the last boolean token send out. Let *k* denotes the value of the *numberOfBits* parameter. An exception is thrown if the input integer is smaller than -2^k or greater $2^k - 1$.

5. **DoubleToFix**
   This actor converts a DoubleToken to a FixToken with a specified precision. Note that this conversion is lossy, in that the output is an approximation of the input. The approximation can be constructed using a variety of rounding and overflow strategies.

   The precision of the output is given by the *precision* parameter, which is an integer matrix of the form [*m*, *n*], where the total number of bits in the output is *m*, of which *n* are integer bits. The default precision is [16, 2], which means that an output has 16 bits, of which 2 bits represent the integer part.

   The rounding strategy is defined by the *rounding* parameter and defaults to *nearest* (or *half_floor*), selecting the nearest representable value. The floor value nearer to minus infinity is used for values half way between representable values. Other strategies such as *truncate* are described under VisualSim.math.Rounding.

   The overflow strategy is defined by the *overflow* parameter and defaults to *saturate* (or *clip*). Out of range values are saturated to the nearest representable value. Other strategies such as *modulo* are described under VisualSim.math.Overflow.

6. **FixToDouble**
   This actor converts a FixToken into a DoubleToken. This conversion is explicitly provided because there is no lossless conversion between these two types in the type lattice, and thus the type system does not do this conversion automatically. Although the conversion might be lossless, a FixToken can, in principle, have arbitrary precision, and thus can exceed the capabilities of a double.

7. **FixToFix**
   This actor converts a FixToken into another FixToken with a specified precision. Note that this conversion may be lossy, in that the output may be an approximation of the input. The approximation can be constructed using a variety of rounding and overflow strategies,

   The precision of the output is given by the *precision* parameter, which is an integer matrix of the form [*m*, *n*], where the total number of bits in the output is *m*, of which *n* are integer bits. The default precision is [16, 2], which means that an output has 16 bits, of which 2 bits represent the integer part.

   The rounding strategy is defined by the *rounding* parameter and defaults to *nearest* (or *half_floor*), selecting the nearest representable value. The floor value nearer to minus infinity is used for values half way between representable values. Other strategies such as *truncate* are described under VisualSim.math.Rounding.

   The overflow strategy is defined by the *overflow* parameter and defaults to *saturate* (or *clip*). Out of range values are saturated to the nearest representable value. Other strategies such as *modulo* are described under VisualSim.math.Overflow.

8. **Int_Long_to_Double**
   This actor converts an integer or long type input number to a double output token. The type of output token is double.

9. **InUnitsOf**
   An actor that converts input tokens to specified units by dividing the input by the value of the *units* parameter. This actor is designed to be used with a *unit system*, which must be included in the model (note that some VisualSim applications do not include unit systems).
   The units are specified by the *units* parameter, which contains a DoubleToken with units. The input tokens and the token in the *unit* parameter must have the same unit category. Otherwise, an exception will be thrown in the fire() method. Unit categories include the ones defined in the MoML file, such as length, time, mass, and the composite ones formed through the base categories, such as length/time (speed), and length * length (area). The output token is a DoubleToken without units.

10. **Round**
    Produce an output token on each firing with a value that is equal to the specified rounded value of the input. The input type is DoubleToken. The output type is IntToken. The functions are a subset of those in the java.lang.Math class. They are:
    a. **ceil**: Round towards positive infinity.
    b. **floor**: Round towards negative infinity.
    c. **round**: Round towards nearest integer.
    d. **truncate**: Round towards zero.
    If the argument is NaN, then the result is NaN. The default is "round".

11. **TokenToExpression**
    This actor reads a string expression from the input port and outputs the token resulting from the evaluation. The type of the output port defaults to general, meaning that the only output will be a pure event. In order to usefully use this class, the type of the output port must be set to the type of the expression that is expected.

Complex:

1. **ComplexToPolar**
   Convert a complex token to polar coordinates, which are represented by two double tokens (magnitude and angle). The output angle is in radians.

2. **PolarToComplex**
   This actor reads two double tokens (magnitude and angle) and outputs a single complex token. The output is a complex token representation of the coordinates given at the inputs in polar form. The angle input is assumed to be in radians. If either input is NaN or infinity, then the output is NaN or infinity.

3. **ComplexToCartesian**
   Read a complex token and output double tokens that represent the real and imaginary parts to two different output ports.

4. **CartesianToComplex**
   Convert a Cartesian pair (represented as two double tokens) to a single complex token. At each firing of the actor, it will consume exactly one token from each of the two input ports and produce a complex token on the output port. The x input becomes the real output and the y input becomes the imaginary output. If either input port is empty, nothing is produced.

5. **PolarToCartesian**
   This actor reads two double tokens (magnitude and angle) and outputs two new double tokens (x and y). The outputs are a Cartesian representation of the pair given at the

inputs in polar form. The angle input is assumed to be in radians. If either input is NaN or infinity, then the outputs are NaN or infinity.

6. **CartesianToPolar**
   Convert a Cartesian pair, which is represented by two double tokens (x and y), to a polar form, which is also represented by two double tokens (magnitude and angle). The angle is in radians.

String:

1. **Number_to_String**
   This actor converts an input number, the *input*, including integer, long, or double to a string output, the output token. The type of output token is string.

2. **String_to_Number**
   This actor converts an input string, the *input*, to one of four output, the output, type tokens, including integer, double, or string. The type of output token is selected by the user via menu parameter.

3. **ImageToString**
   This actor reads an ObjectToken that is a java.awt.Image from the input and writes information about the image to the output as a StringToken.

4. **Replace_String**
   This actor converts an input string, the *input*, by the parameter settings to a string output, the output token. The type of output token is string.

Array:

1. **IntArrayToString**
   Convert an integer-array into a string. Uses only the low order byte from each integer. NOTE: Assumes an 8-bit character set. The output is a string assembled from these bytes. This actor is designed to facilitate use of the SerialComm serial communication actor which uses the same kind of integer array format as this actor. Datagram actors can use this format as well.

2. **StringToIntArray**
   Convert a string to an integer-array. The output is an array of integers constructed by placing one byte (i.e. one character) of the string into the least significant byte of each integer. Typically, this byte is the ASCII code of the character.

3. **StringToUnsignedByteArray**
   Convert a string to an array of unsigned byte. The conversion is performed using the default character set, returned by the system property "file.encoding".

4. **UnsignedByteArrayToString**
   Convert an array of bytes into a string. The conversion is performed using the default character set, returned by the system property "file.encoding". The output is a string assembled from these bytes.

5. **ArrayToElements**
   On each firing, this actor reads an ArrayToken from the input port and send out each element token to each channel of the output port. If the width of the output port (say, n) is less than the number of elements in the array (say m), then the first n elements in the

array will be sent, and the remaining tokens are discarded. If n is greater than m, then the last n-m channels of the output port will never send tokens out.

6. **ArrayToSequence**
   This actor reads an array at the input and writes the array elements as a sequence to the output. The parameter *arrayLength* can be used to specify the length of arrays that the actor will accept. If the *enforceArrayLength* parameter true, then if an input array does not match *arrayLength*, the fire() method will throw an exception. This feature is important in simulators, such as SDF, that do static scheduling based on production and consumption rates. For other simulators, such as DE and PN, the *enforceArrayLength* parameter can be set to false, in which case the *arrayLength* parameter will be ignored.

7. **MatrixToSequence**
   This actor unbundles a matrix into a sequence of output tokens. On each firing, it writes the elements of the array to the output as a sequence of output tokens. and writes one output matrix token with the specified number of rows and columns. If the *enforceMatrixSize* parameter true, then if an input matrix does not match *rows* and *columns*, then the fire() method will throw an exception. This feature is important in simulators, such as SDF, that do static scheduling based on production and consumption rates. For other simulators, such as DE and PN, the *enforceMatrixSize* parameter can be set to false, in which case the *rows* and *columns* parameters will be ignored. This actor is polymorphic. It can accept any matrix input and the output will have the type of the elements of the matrix.

8. **ElementsToArray**
   On each firing, this actor reads exactly one token from each channel of the input port and assembles the tokens into an ArrayToken. The ArrayToken is sent to the output port. If there is no input token at any channel of the input port, the prefire() will return false.

   It can accept inputs of any type, as long as the type does not change, and will produce an array with elements of the corresponding type.

9. **SequenceToArray**
   This actor bundles a specified number of input tokens into a single array. The number of tokens to be bundled is specified by the *arrayLength* parameter.
   This actor is polymorphic. It can accept inputs of any type, as long as the type does not change, and will produce an array with elements of the corresponding type.

10. **SequenceToMatrix**
    This actor bundles a specified number of input tokens into a matrix. On each firing, it reads *rows* times *columns* input tokens and writes one output matrix token with the specified number of rows and columns. This actor is polymorphic. It can accept inputs of any scalar type that has a corresponding matrix type.

11. **DoubleToMatrix**
    This actor converts a sequence of input tokens to a matrix. The actor reads *rows* times *columns* inputs and inserts their values into a double matrix with *rows* rows and *columns* columns. The first row is filled first, then the second row, then the third, etc.

12. **MatrixToDouble**
    This actor converts a matrix input sequence of output tokens. The input must be a DoubleMatrixToken and the output will be a sequence of instances of DoubleToken. The number of outputs produced on each firing is the number of elements in the matrix. This is assumed to equal the product *rows* times *columns*, although this is not enforced unless

the actor is under the control of an instance of SDFDirector. The SDF director requires this information to construct its schedule. The first row is produced first, then the second row, then the third, etc.

JSON:

1. **TokenToJSON**
   This block converts any VisualSim data type to a JSON token.

2. **JSONToToken**
   This block converts a JSON token to a data structure or anay other VisualSim data type depending on the content of the JSON.

String Operation Libraries:

1. **StringCompare**
   Compare two strings specified either as inputs or parameters. The output is either true or false, depending on whether the comparison function is satisfied. The comparison functions are:
   - **equals**: Output true if the strings are equal (Default).
   - **startsWith**: Output true if *firstString* starts with *secondString*.
   - **endsWith**: Output true if *firstString* ends with *secondString*.
   - **contains**: Output true if *firstString* contains *secondString*.

   The strings to be compared will be taken from the inputs if they are available, and otherwise will be taken from the corresponding parameters.

2. **StringFunction**
   Produce the output string generated by applying a user-specified string function on a provided input string. The available string functions are a case sensitive subset of the java.lang.String functions.
   - **trim**: Remove leading and trailing whitespace from a string.
   - **toUpperCase**: Convert all letters in a string to uppercase.
   - **toLowerCase**: Convert all letters in a string to lowercase.

3. **StringIndexOf**
   Output the index of a *searchFor* string contained in a given *inText*. The search begins at the index given by *startIndex* and ends at either the end or the beginning of the string, depending on whether *searchForwards* is true or false, respectively. If the string is not found, then the output is -1. If the string is found, then the output is the index of the start of the string, where index 0 refers to the first character in the string.

4. **StringLength**
   Output the length of a string provided at the input.

5. **StringMatches**
   Pattern match a string to a regular expression and output a true if it matches and a false if it does not.

6. **StringReplace**
   On each firing, look for instances of the pattern specified by *pattern* in *stringToEdit* and replace them with the string given by *replacement*. If *replaceAll* is true, then replace all instances that match *pattern*. Otherwise, replace only the first instance that matches. If

there is no match, then the output is the string provided by *stringToEdit*, unchanged. The *pattern* is given by a regular expression.

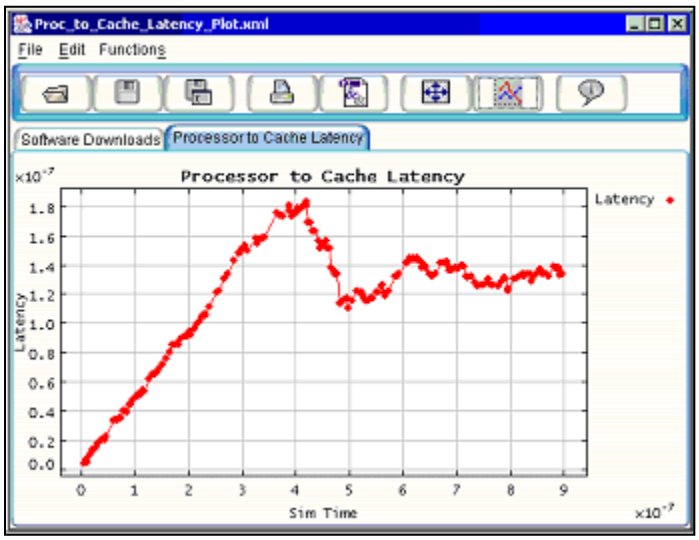
7. **StringSubstring:**
   Output a substring of the string provided at the input. The position of the substring within the input string is determined by the *start* and *stop* port parameters. Following Java convention, the character at *start* is included, but the character at *stop* is not. If the *stop* is less than *start*, then the substring starts at *start* and extends to the end of the string. The default values for *start* and *stop* are both 0; this results in an empty string at the output.

Result- Analysis and Plotting

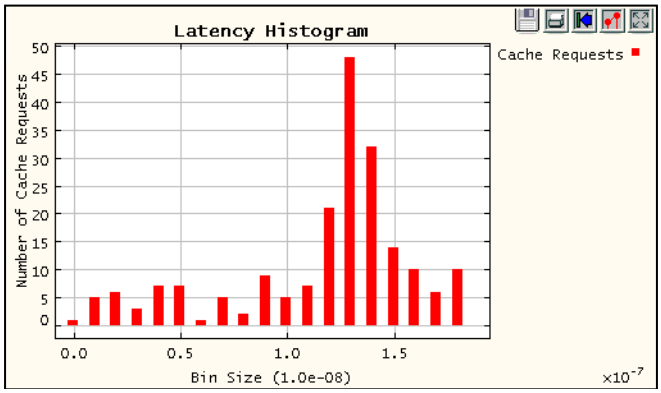
Post-Processing is sometimes viewed as a “post” modeling activity. You can use the VisualSim Post Processor to graphically display and analyze performance data collected from the simulation. The Post Processor can organize results into a variety of x-y graphs, bar charts, and scatter plots, and displays them in either graphical or tabular format. The Post Processor can also perform a variety of statistical functions as a mean, min, max, standard deviation, and confidence intervals.
This section focuses on post-processing in terms of what VisualSim can provide in terms of system latency, system throughput, and system utilization. These are the three most common performance modeling statistics to obtain and plot in the post-processor. VisualSim can provide post-processing capability on other system metrics. The approach is similar to the three common metrics.

System Latency – System latency can be calculated quite easily, because the TIME field in the composite Data Structure contains the time that the Data Structure was created. If one subtracts TIME (creation time, a data structure Field) from the current simulation time (TNow), then the difference is the time spent in the system. One might use one of the optional composite Data Structure header fields (DELTA) to record the transaction, or packet latency (DS expression):

```
System_Latency = Sim_Time – Time_Created  [input.DELTA = Tnow – input.TIME]
```



Post Processing Latency



Post Processing Histogram

System Throughput – System throughput involves accumulating data being sent through a digital model, and at specified time intervals, calculating the Throughput in Bytes per second, Kbytes per second, Mbytes per second, or Gbytes per second. You need to accumulate the data passing through the system in a variable, or statistical block, and then calculate the system throughput by dividing the amount of data by the time interval in the appropriate units.

```
System_Throughput = System_Data_Accumulated / System_Sample_Time
```

System Utilization -- System utilization is related to system throughput in that is the proportion of system throughput to system capacity:

```
System_Utilization = (System_Throughput  *  100.0) / System_Capacity
```

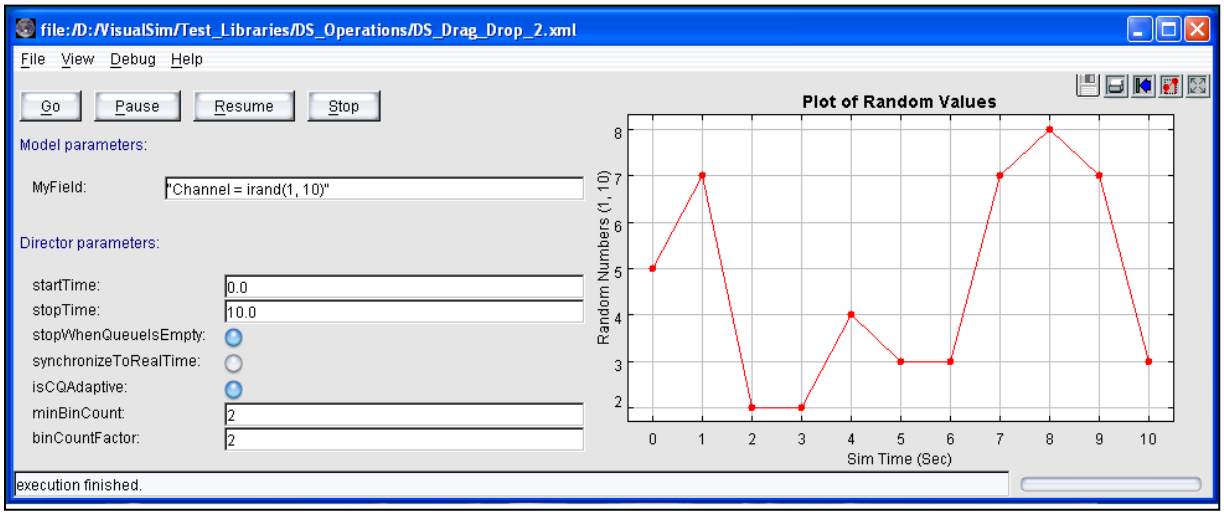
In the case of time-related activities, system utilization may simply be the time the resource is busy divided by the total time:

```
System_Utilization = (System_Time_Busy  * 100.0) / System_Sample_Time
```

System Throughput and *System Utilization* can be plotted similar to System Latency, using a **TimeDataPlotter** or **HistogramPlotter** block, because they are both single valued metrics. There are variations of the TimeDataPlotter including XYPlotter, DS_TimeDataPlotter, and DS_XYPlotter. The prefix DS_ allows the user to control the visual effects and attributes of the plot dataset.

TimeDataPlotter Block Output

The following figure shows the TimeDataPlotter Block output in Simulation Cockpit.



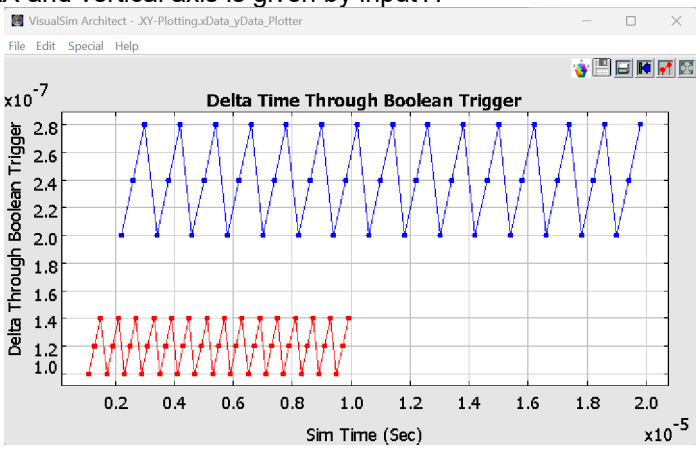
Model Plot

The TimeDataPlotter block accepts a single double input token (Y Axis) and appends the current simulation time (X Axis) to plot each X,Y value. There are other plotting blocks available in the **Result** Library.

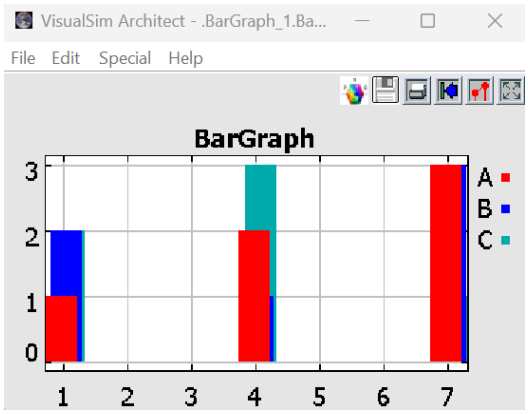
Other than the Time DataPlotter (XY plotter) and histogram ploitter mentioned above, user can utilize different plotters in VisualSim to analyze their model for better understanding of its working.

1. xData_yData_Plotter
2. BarGraph
3. DS_xTime_yData_Plotter
4. DS_xData_yData_Plotter
5. ArrayPlotter
6. CandleStick plotter

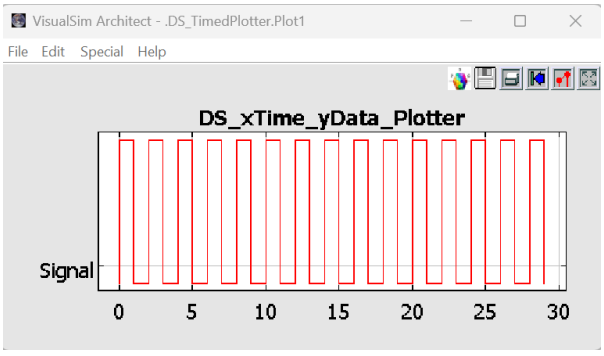
xData_yData_Plotter plots data at inputX and inputY on an instance. When plotted, the first dataset of inputX and the first dataset of inputY are together considered the first signal, then the second channel of inputX and the second channel of inputY are considered the second signal, and so on. This requires that inputX and inputY to arrive simultaneously. The block assumes that there is at least one value available for each dataset. The horizontal axis is given by the value of the input from inputX and vertical axis is given by inputY.



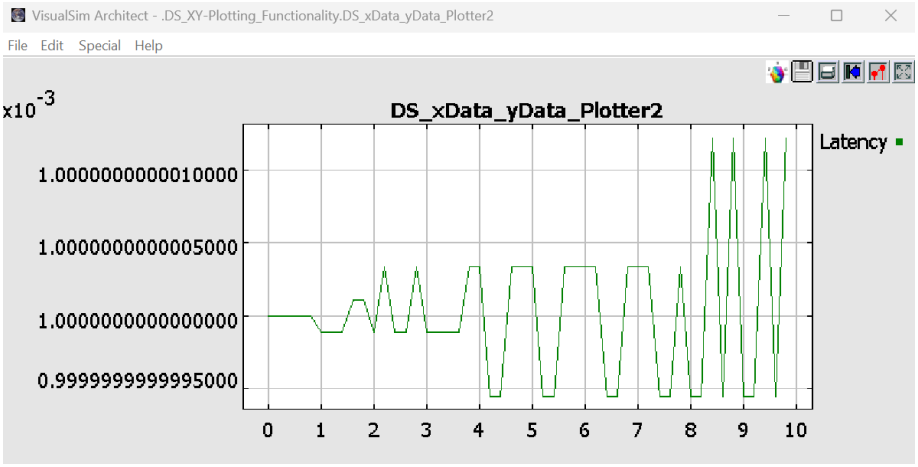
Bar Graph plots the incoming data on the Y-Axis against the current simulation time on the X-axis. Every wire connected to this block input is considered a separate dataset and plotted separately. Multiple datasets can be created in associating each stream as an instance/channel. To create a independent channel/instance, each incoming data wire can be attached to a relation or the DS_to_Instance block can be placed prior to this block with all the data wires connected to it. Data at the input, which can consist of any number of channels, are plotted on this instance. Each input channel is plotted as a separate data set. Each input token is an array of doubles.



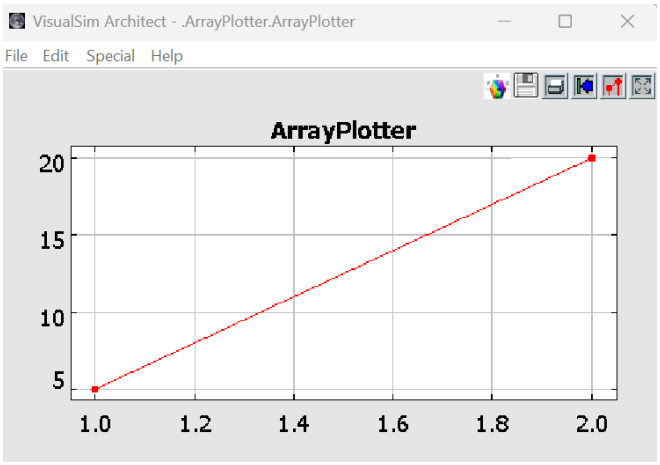
DS_xTime_yData_Plotter generates a timing diagram for a sequence of data structures placed on the input port. The plot will display high and low value for each dataset or signal. The block will only display a sequence of 0 or 0.0 and 1 or 1.0 values for the Y-axis. Each dataset is uniquely distinguished by the offset parameter value. The offset is the height from the 0.0 of the Y-axis. For a Offset of 4, the 0 value is at 4 and the 1 is at value 5. The Simulation Time is on the X-axis and the Datasets are on the Y-axis. The Field_Trace_Name is the legend on the Y-axis for the particular dataset and will be denoted at the Offset value + 0.2.



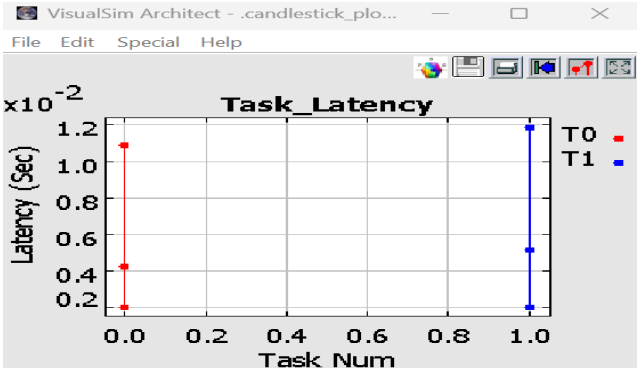
DS_xData_yData_Plotter displays points on the Plot using values from a data structure arriving on the input port, fields of the Data Structure. This plot displays a maximum of 29 individual datasets from 0 to 28. The parameters of this block match the fields (or RegEx) of the incoming Data Structure to determine the coordinates, color and trace identifier (Dataset). All DataSets arrive on the same connection to the input port of this block and are differentiated by the color name or integer value. The legend for each dataset can be retrieved from the legend parameter (highest priority and overrides everything else), Field_Trace_Name (next priority) or nothing, if both of them are empty. The legend is displayed on the right side of the plot display. The X- and Y- values can be any RegEx function.



The **Array plotter** can be used to generate live plots, where the display updates on every input. If there are 5 results to be displayed with a time interval between each result, the display will be updated entirely for each input received. At the end of simulation, the final data entered to the display will be retained.
The input to the plot must be an array format with the port type array. The x and y axis values can be represented as an array, such as {x,y}.



Candle Stick plotter can be realized using script block, where the max, min and mean of the samples are plotted in a candle stick format.



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_Arbiter, Bus_Interface, Cache, DRAM, DeviceInterface (a.k.a I_O), DMA, Bridge, Swith and Crossbar blocks.

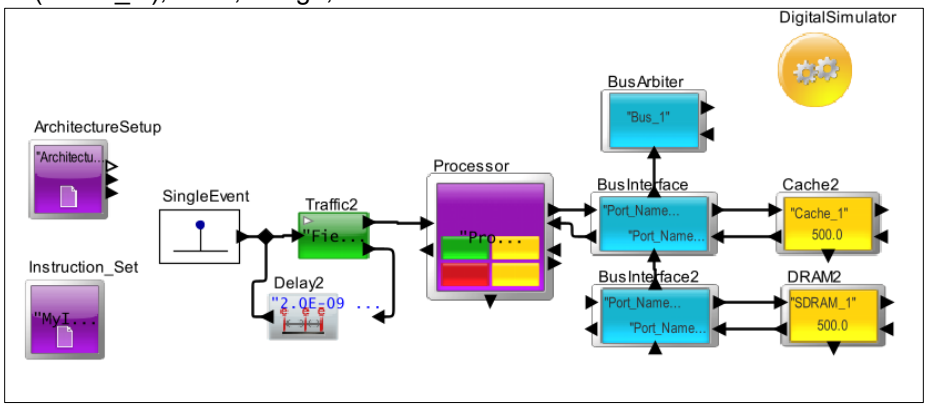
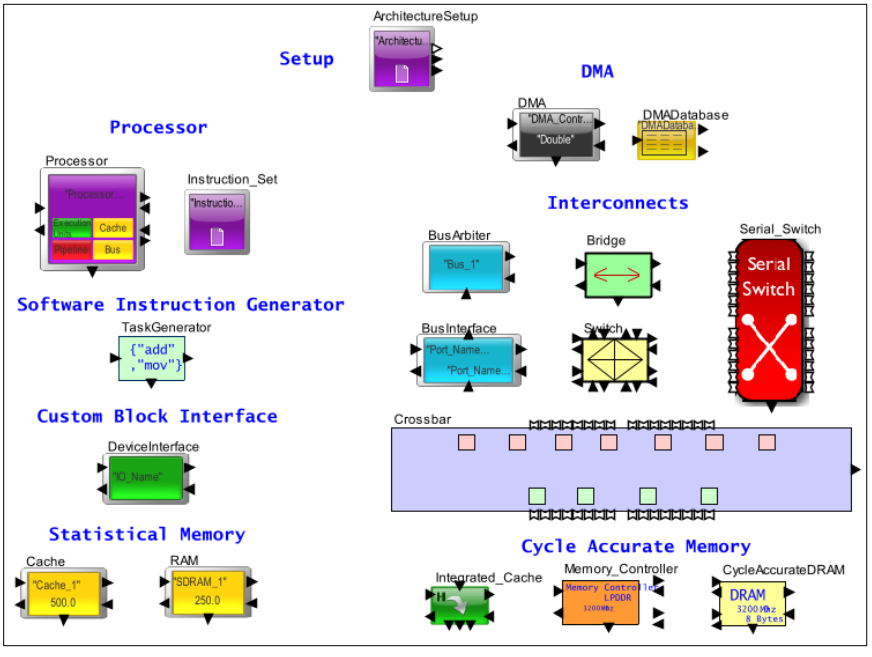
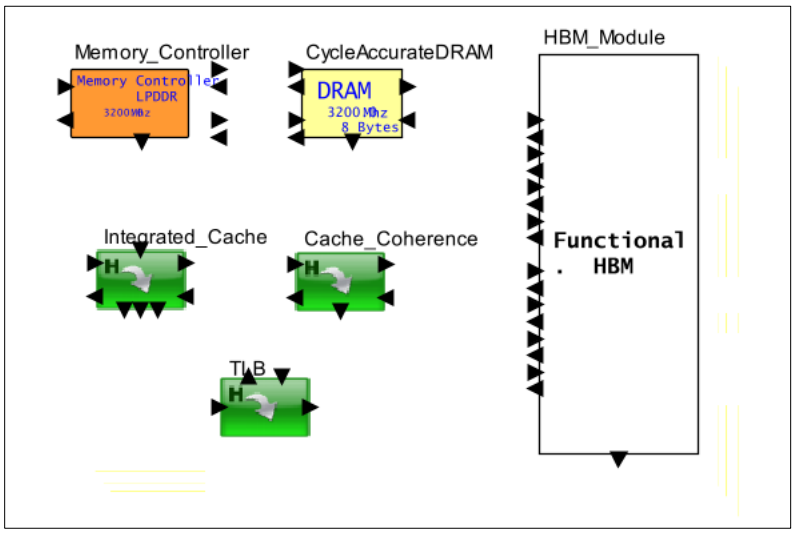


Figure 41 Model containing key Hardware Architecture Library Blocks



Hardware modeling components



Memory Libraries external to the processor

Processor	**Cache**
○ Pipeline ○ External calls from pipeline for advanced operation creation ○ Registers ○ 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 ○ Outstanding memory request ○ Branch Instruction ○ Branch misprediction flush ○ Statistics - Throughput, utilization, latency ○ Processor width	○ Speed ○ Size ○ Hit-ratio ○ Pre-fetch ○ Buffer ○ Line width ○ Cache width **DRAM** ○ Speed ○ Size ○ Buffer ○ Line width ○ Memory width ○ Access Time - Read, Refresh, Write, Erase, Read/Write, ○ Banks ○ Refresh ○ Controllers - SDR, DDR, DDR-2, DDR-3, DDR-4, DDR-5 QDR, RDR, custom
Bus Arbiter and Bus Interface	**Instruction Set**
○ Control message ○ Data message ○ Speed ○ Buffer	○ List of instructions by Execution Units ○ Cycle time by instructions ○ Associate instructions by groups

○ Word width ○ Switch ○ Arbitration **DeviceInterface (a.k.a I_O)** ○ External In/Out Port ○ Bridge Functionality ○ External Routing	○ Setup uniform distribution for instruction cycle time ○ Load store instruction data width definition. **Architecture Setup** ○ 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 done 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.

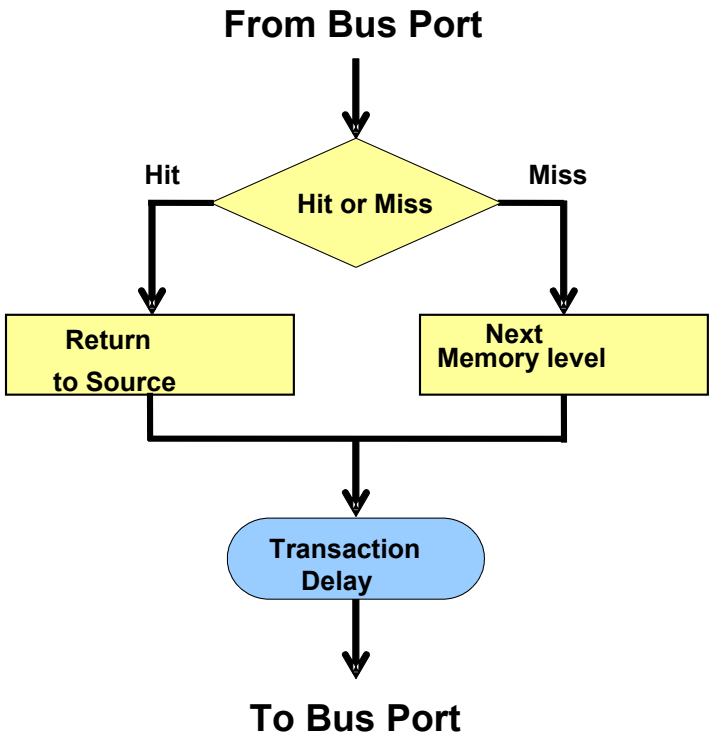


Figure 42 Cache Flow Diagram

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 and a miss read request is sent to the next level memory.

A Cache 'Read' request is processed and the data is returned to the source. If the instruction is Cache 'Write' and the A_Task_Flag is false then it is processed and the token is just dropped after a transaction delay, if A_Task_Flag is true then the response for the write is sent to source.

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). This can be observed by using a Timing Diagram Block and configure the cache name to the corresponding cache block name.

Statistics

Statistics collected are Cache utilization, throughput, transaction delay time and Hit Ratio. 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

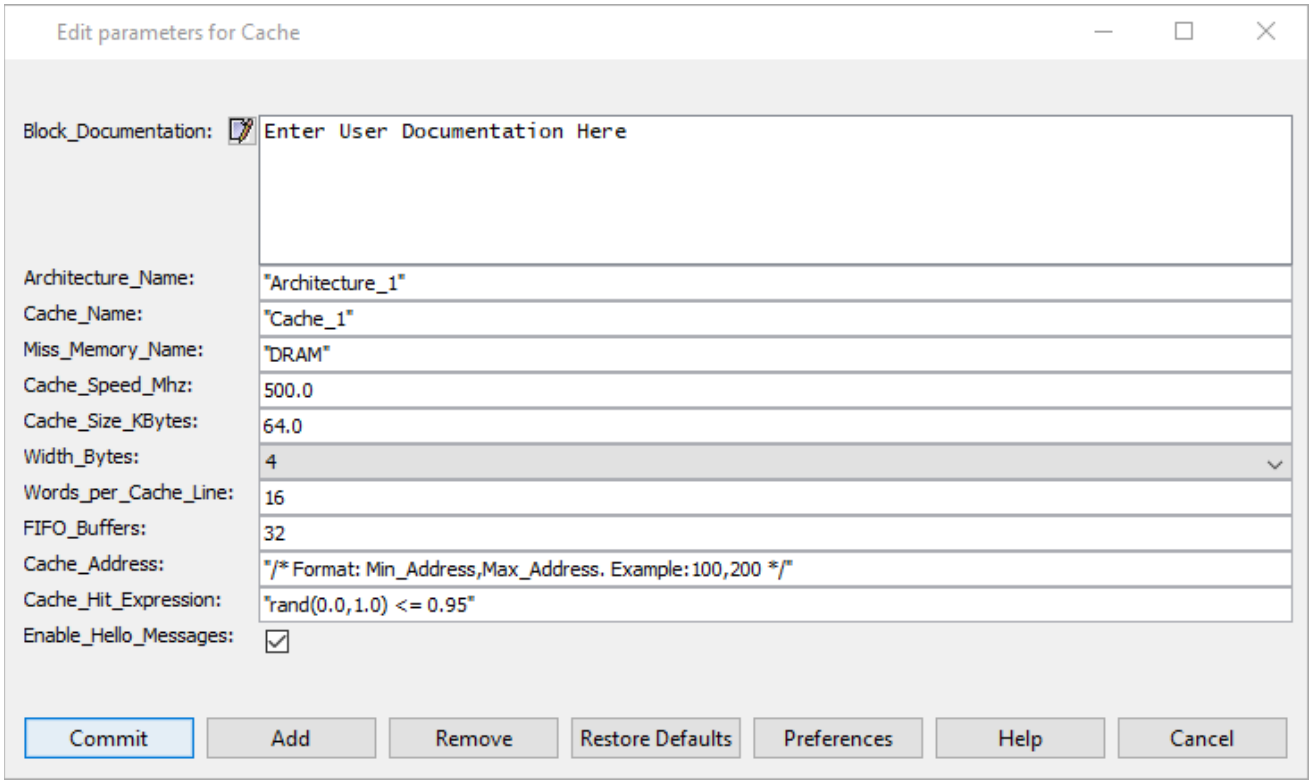


Figure 43 Cache Parameter Window

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 / 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 RAM (SDR)
- Double Data Rate (DDR, DDR-2, DDR-3, DDR-4 and DDR-5)
- Quad Data Rate (QDR) SRAM
- RAMBUS DRAM

Functionality

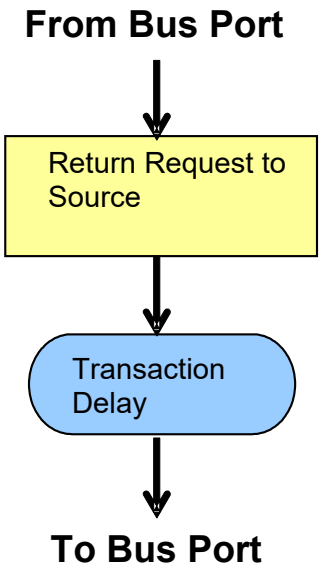


Figure 44 Memory Flow Diagram

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

To observe the plot please add Timing Diagram block from Hardware setup and configure the DRAM name . 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

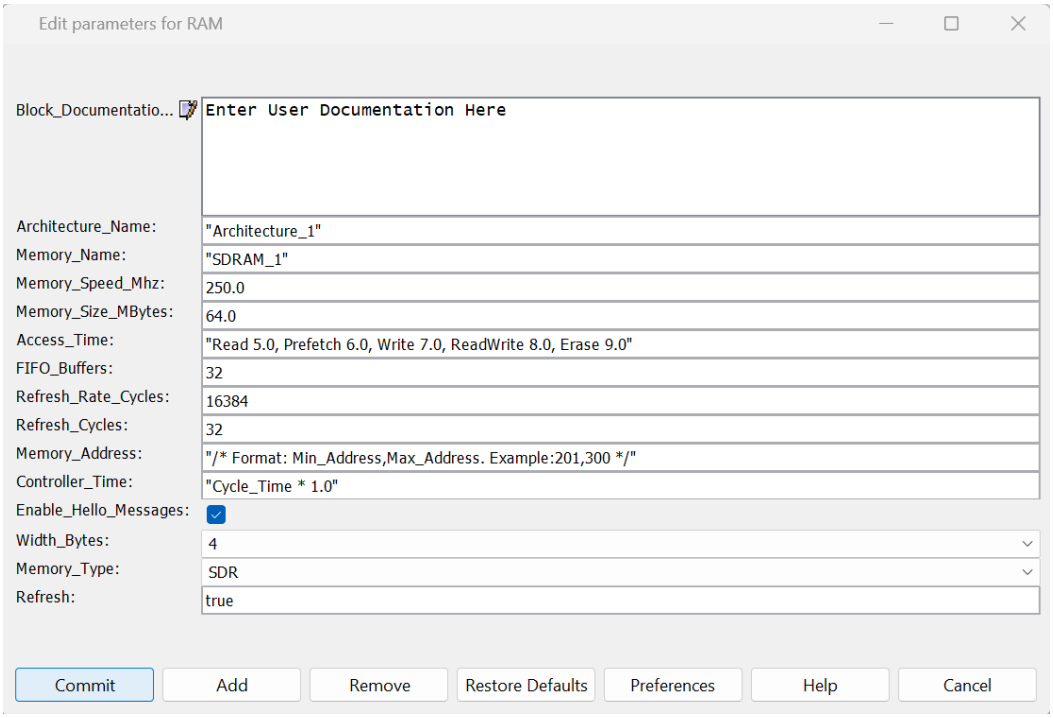


Figure 45 Memory Configuration Window

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 / 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.

Refresh
This enables or disables the refresh functionality in the DRAM.

Processor Block

Description

The VisualSim processor block approaches the capabilities of an Instruction Set Simulator (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 ISS 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, optional instruction reorder array, and internal routing with external bus, DMA, or memory.

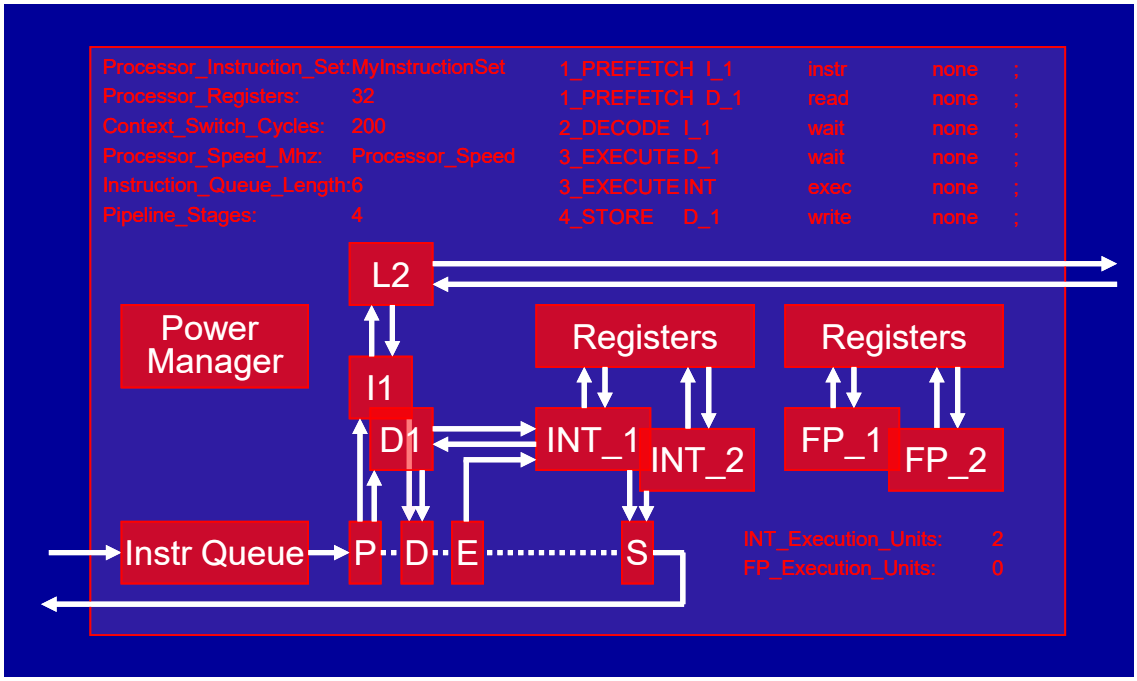


Figure 46 Processor Block

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. The following image shows the processor configuration with necessary parameters, it can also have some optional parameters which are explained in the Configuration of Parameter section below.

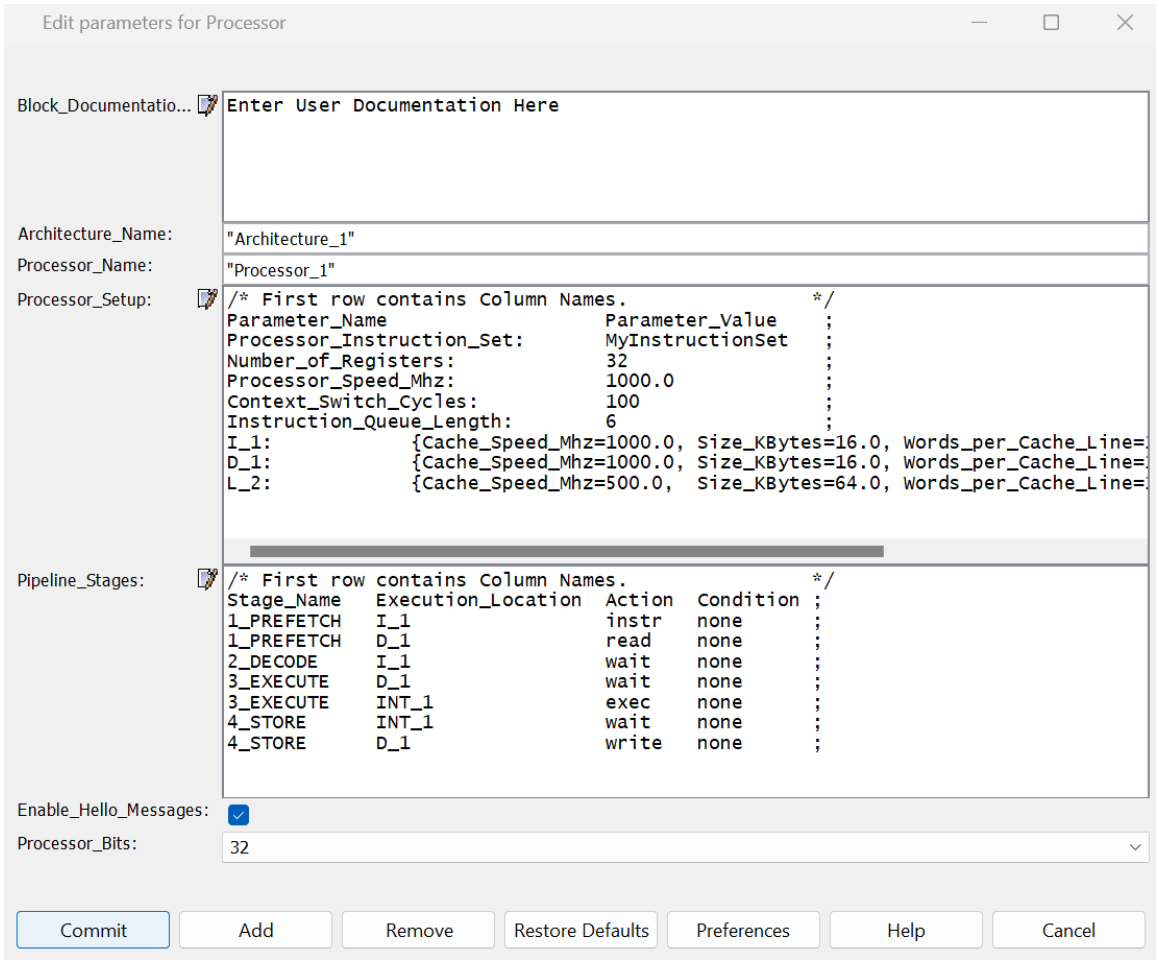


Figure 47 Processor Block Configuration Parameters

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 128 Bits

The Processor_Name must be unique with the Architecture_Name, which can be thought of as the processor-platform domain. The Processor_Setup and Pipeline_Stages will be described in more detail later. The Processor_Bits represent the execution width of the Processor Block.

Processor_DS

The Processor_DS was created as the best data structure to send through the Processor, as this is the same data structure used by other Architectural Library busses and memories. The following are specific data structure fields to be set prior to entering the Processor Block.

{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        = "Processor_1",
A_First_Word         = true,
A_Hop                = "Processor_1",
A_IDX                = 0,
A_Instruction        = {"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "ADD",
"ADD", "ADD", "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             = "Src",
A_Status             = "Status",
A_Task_Flag          = false,
A_Task_ID            = 1L,
A_Task_Name          = "Name",
A_Time               = 0.0,
A_Variables          = 16}
```

A_Source may represent a port on the bus, or an RTOS name where the task or thread originated.

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 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 Bus Interface 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 Bus Interface. 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.

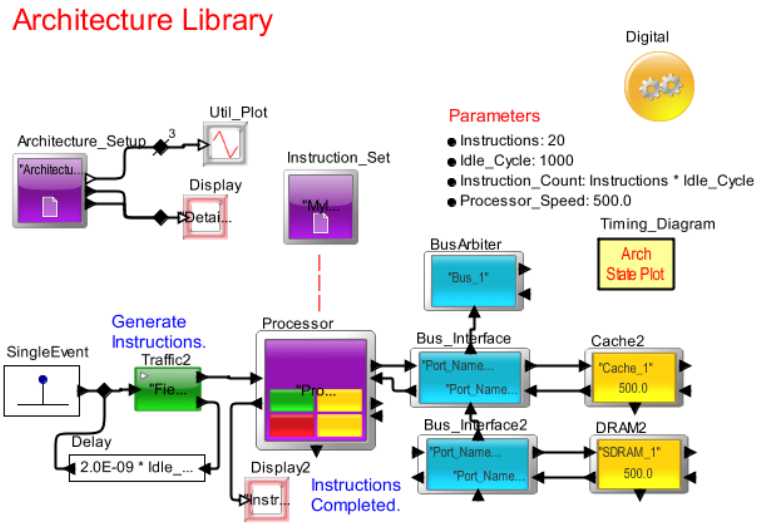


Figure 48 Typical Processor Model

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                  ;
ROB_Size:                       160                ;
I_1:  {Cache_Speed_Mhz=cache_speed, Size_KBytes=16.0, Words_per_Cache_Access=2,
Hit_Ratio=0.9, Words_per_Cache_Line=16, Cache_Miss_Name=SDRAM_1,
Outstanding_Req_Count = 1}
D_1:  {Cache_Speed_Mhz=cache_speed, Size_KBytes=16.0, Words_per_Cache_Access=2,
, Hit_Ratio=0.9, Words_per_Cache_Line=16, Cache_Miss_Name=SDRAM_1,
Outstanding_Req_Count = 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 (Optional)
Number of instructions per cycle in the pipeline execution stage, optional integer parameter. This is an optional parameter.

ROB_Size (Optional)
Defines the size of Re-Order Buffer for out of order execution configuration.

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:

```
I_1  {Cache_Speed_Mhz=500.0, Size_KBytes=64.0,  Words_per_Cache_Access=1,
Words_per_Cache_Line=16, Hit_Ratio=0.9, Cache_Miss_Name=L_2,
Outstanding_Req_Count = 1}
```

D_1 {Cache_Speed_Mhz=500.0, Size_KBytes=64.0, Words_per_Cache_Access=1, Words_per_Cache_Line=16, Hit_Ratio=0.9, Cache_Miss_Name=L_2, Outstanding_Req_Count = 1, Cache_Type = Load_Store}
L_2 {Cache_Speed_Mhz=500.0, Size_KBytes=64.0,
   Words_per_Cache_Access=1,Words_per_Cache_Line=16, Hit_Ratio=0.9, Cache_Miss_Name=Cache_1, Outstanding_Req_Count = 1}

The Processor_Name (Optional) 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 (Optional) 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.
Hit_Ratio defines the hit miss percentage for that 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.
Outstanding_Req_Count (Optional) defines the number of outstanding requet can be sent to the miss memory.
The Cache_Type (Optional), is used to define the D cache or data cache that is connected to the processor. In case like the cache has different name this will help the processor to differentiate between I cache and D cache.

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.

The above mentioned structure works for the internal cache configuration. The user can also connect the cycle accurate caches externally for L1, L2 and L3. To enable this the user have to follow this structure:

External_I_Cache   : {}
External_D_Cache : {}

The External prefix make sure the processor read or writes it from the external cache. The name following the External_ provides the cache name, in this the I Cache name is “I_Cache” and the D Cache name is “D_Cache”. The user must update the pipeline stages with this cache name for proper functionality.
The parameters of the cache are configured externally, but the user can add a parameter related to the processor, in this case the I Cache does not need any parameter. The D cache can have Outstanding_Req_Count as a parameter and it can be put in the {}. Eg:

External_D_Cache : {Outstanding_Req_Count =1}

Enabling Data Access

Processor requires additional details to enable data access for load store instructions. To achieve this, users have to define the memory access instructions and its access size in the linked instruction set block as explained below and update the pipeline stages where the D_Cache access supposed happen as “none” and its action as “exec”.

Pipeline update :

```
1_PREFETCH   I_Cache          instr   none     ;
2_DECODE     I_Cache          wait    none     ;
3_EXECUTE    PPC              exec    none     ;
4_STORE      PPC              wait    none     ;
4_STORE      D_Cache          write   none     ;
```

In this case, the STORE stage performs write operation to D_Cache. To enable the data access to D_Cache please make sure the following format is used.

```
1_PREFETCH   I_Cache          instr   none     ;
2_DECODE     I_Cache          wait    none     ;
3_EXECUTE    PPC              exec    none     ;
4_STORE      PPC              wait    none     ;
4_STORE      none             exec    none     ;
```

Instruction set Update: (detailed information on instruction set is given in the “Instruction Set” section below)
The instruction set block or instruction set file must contain the section “size_config” along with the instructions listed for each execution unit. The example format is given below.

```
begin size_config                                    ;
 Read                  3              64        LDR ;
 Write                 3              64        STR ;
end size_config                                      ;
```

The section must contain begin and end like any other section in the instruction set. The Read and Write defines the load and store type of the instruction. The second column defines the pipeline stage in which the load store instructions will be executed. The third column defines the size of the instruction in bits. The Fourth column defines the corresponding instruction or the range of instruction as a reference from load store execution section.

The alternate format is

```
Mnew Ra Rb Rc Rd Re Rf Rg Rh ; /* Label */
ARM INTG ALU FPNEOSIMD LDSTR ;

INTG            INT_1 INT_2  ; /* Integer_add */
ALU             INT_3 INT_4  ; /*ALU Branch*/
FPNEOSIMD   FP_1       ; /*Floating point/NEON/ASIMD instructions*/
LDSTR          INT_5       ; /*Load/Store instructions*/

begin size_config                                          ;
 Read              6                64        INT_5[1:163]    ;
 Write             6                128       INT_5[164:500]  ;
end size_config                                            ;
```

In this case the INT_5 is the load store execution unit and the instructions listed in that from entry 1 to 163 are 64 bit memory access instructions, where the entry from 163 to 500 are 128 bit memory access instructions. And all the load store instructions are executed at 6th stage of the pipeline.

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  none              exec    none      ;
2_DECODE    I_1               wait    none      ;
3_EXECUTE   none              exec    none      ;
3_EXECUTE   INT_1             exec    none      ;
4_STORE     INT_1             wait    none      ;
4_STORE     none              exec    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
“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 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    none                  exec           none    ;
2_DECODE      I_1                   wait           none    ;
3_EXECUTE     none                  exec           none    ;
3_EXECUTE     INT                   exec           none    ;
4_STORE       none                  exec           none    ;
```

A processor with a longer 13 stage pipeline:

```
Stage_Name    Execution_Location      Action   Condition ;
1_FETCH1      I_1             instr   none     ; /* Fetch          */
2_FETCH2      none            exec    none     ; /* Fetch          */
3_DATA0       I_1             wait    none     ; /* Data           */
4_DATA1       none            exec    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      none            exec    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 be 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_EXEC0. 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.

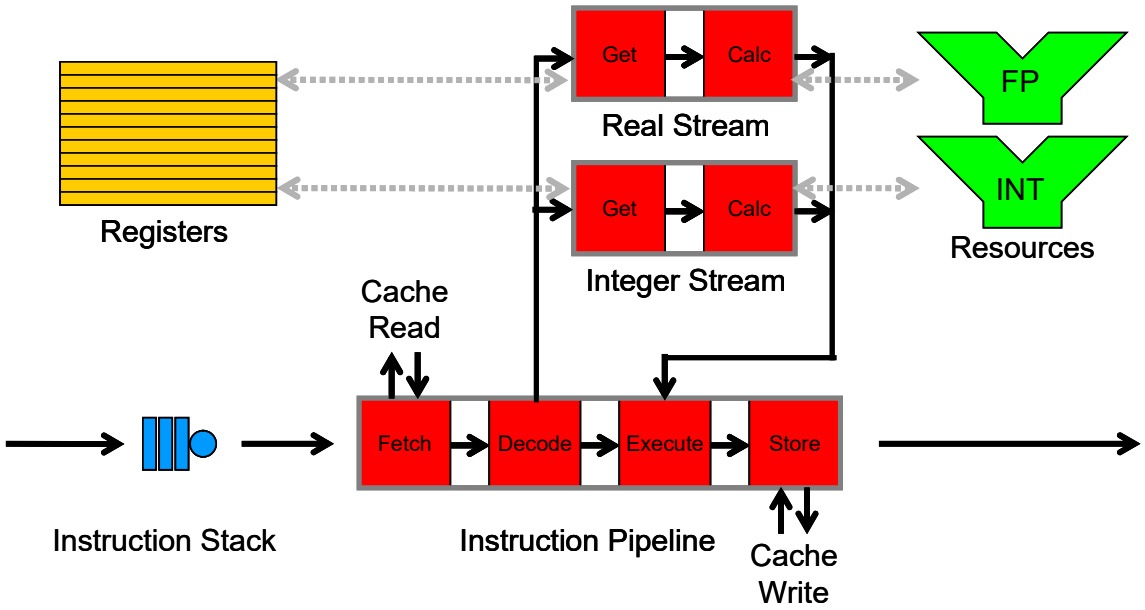


Figure 49 Statistical Processor Block with Pipeline

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

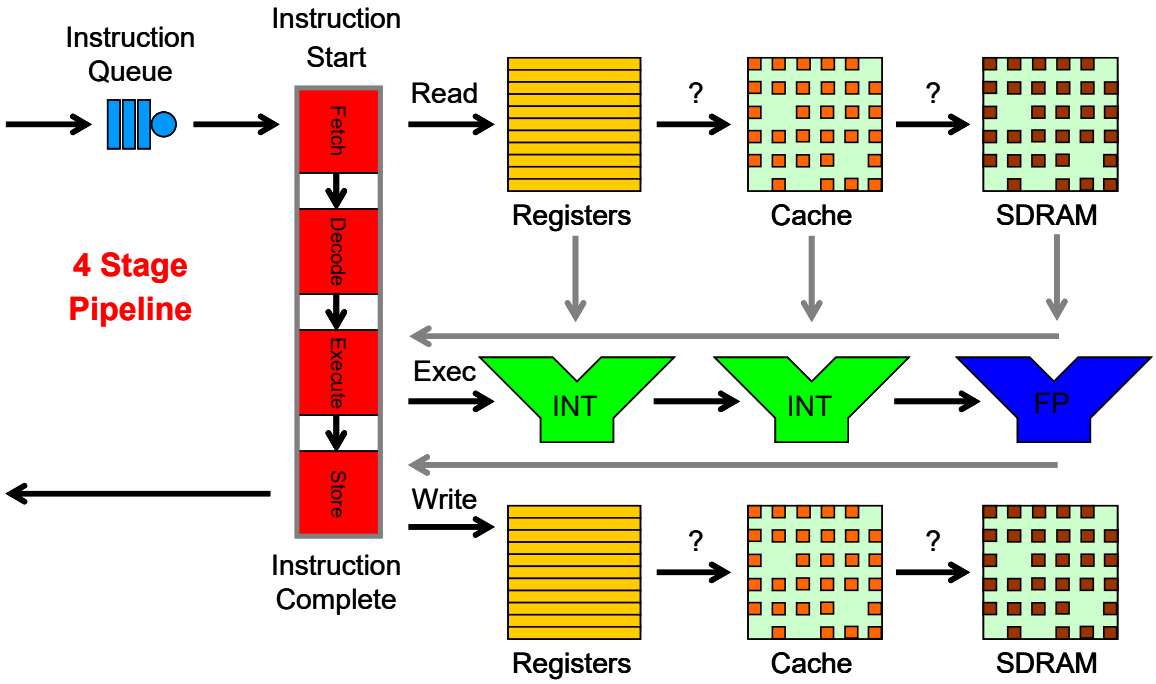


Figure 50 Statistical Processor Block Instruction flow

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:

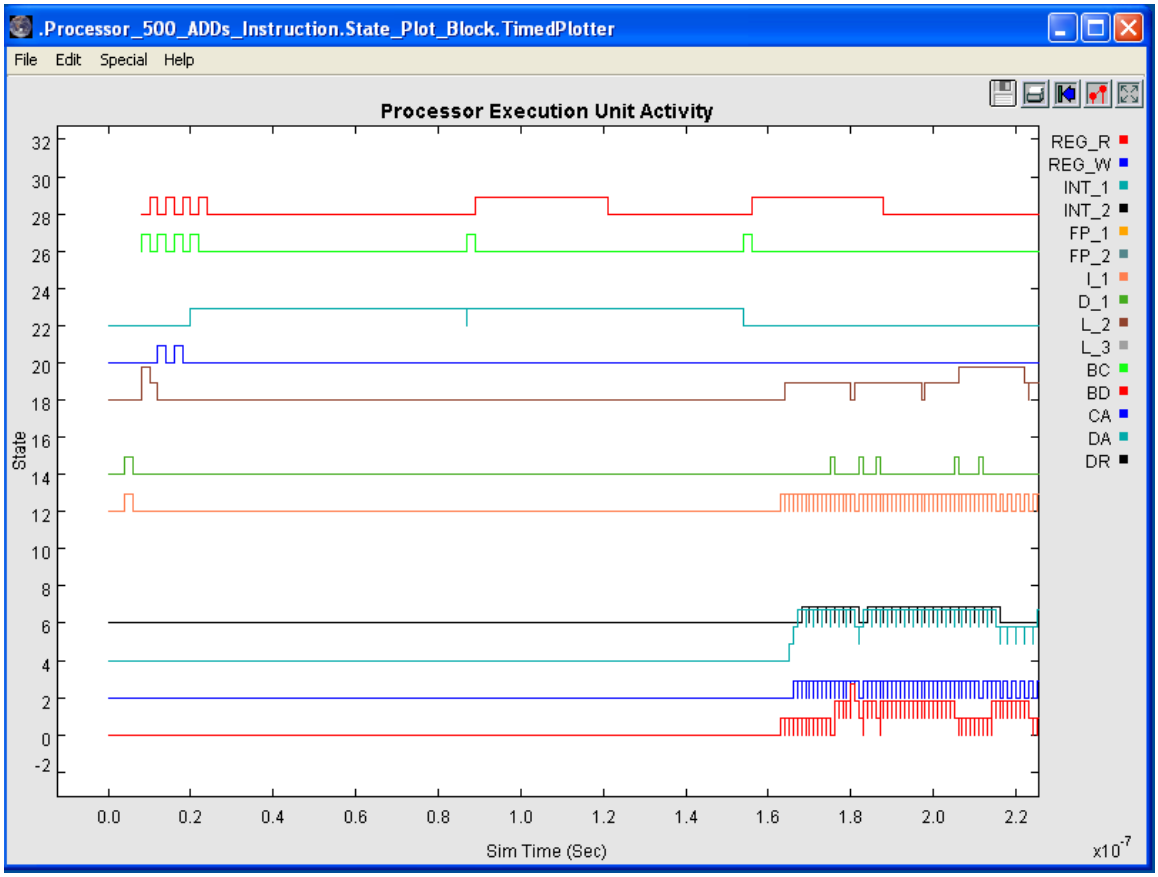


Figure 51 Statistical Processor Block Startup Cache Prefetch

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 the 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 size_config          ;
 Read   3     32   l_s     ;
 Write  3     32   l_sc    ;
end size_config            ;

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 size_config section defines the load store instructions width in bits. In this example the l_s instruction will read or write 32 bit data in the memory. The integer value followed by the Read/Write determines the pipeline stage in which the load store execution happens.

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               ;
```

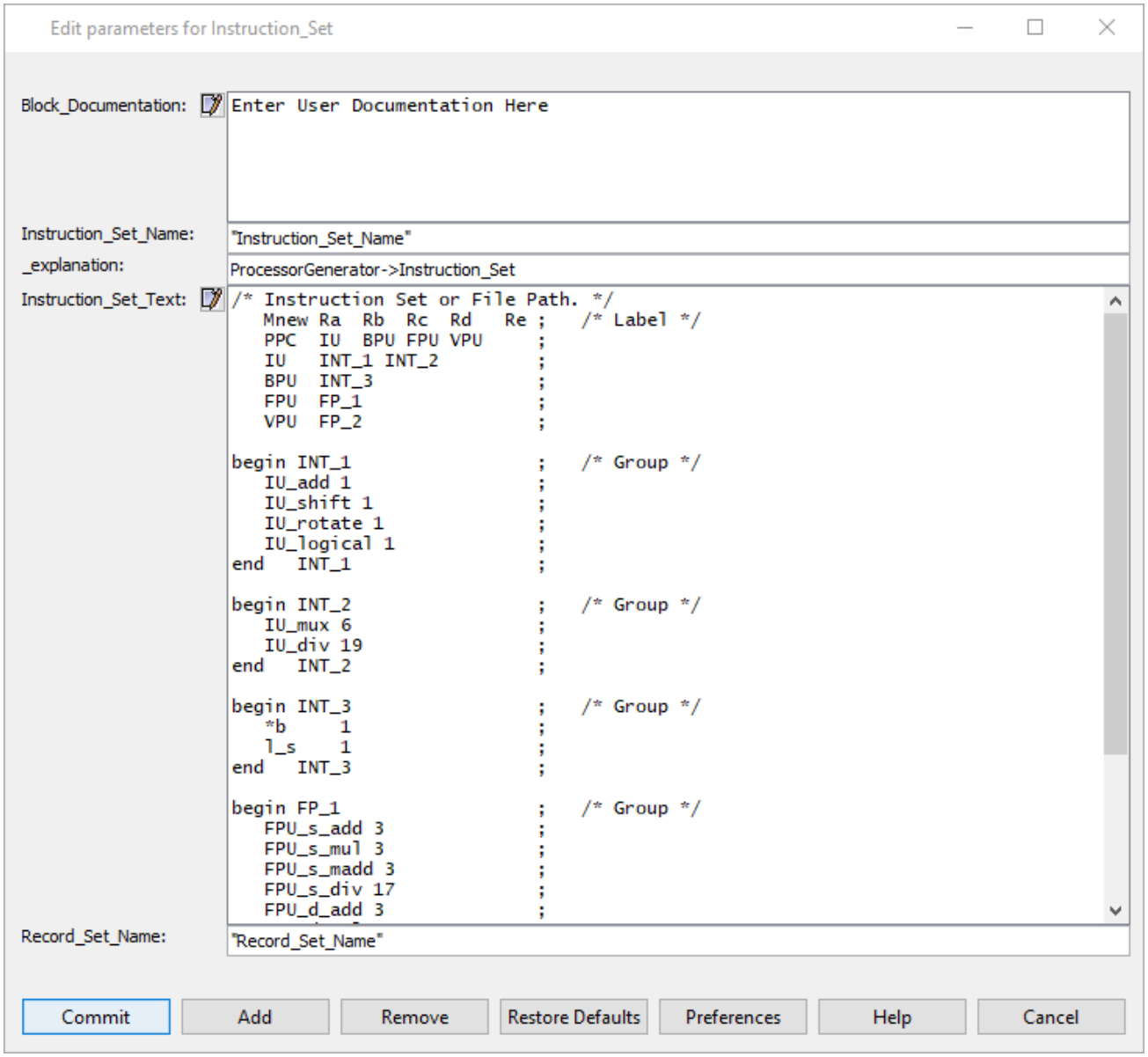


Figure 52 Instruction_Set Parameter Settings

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.

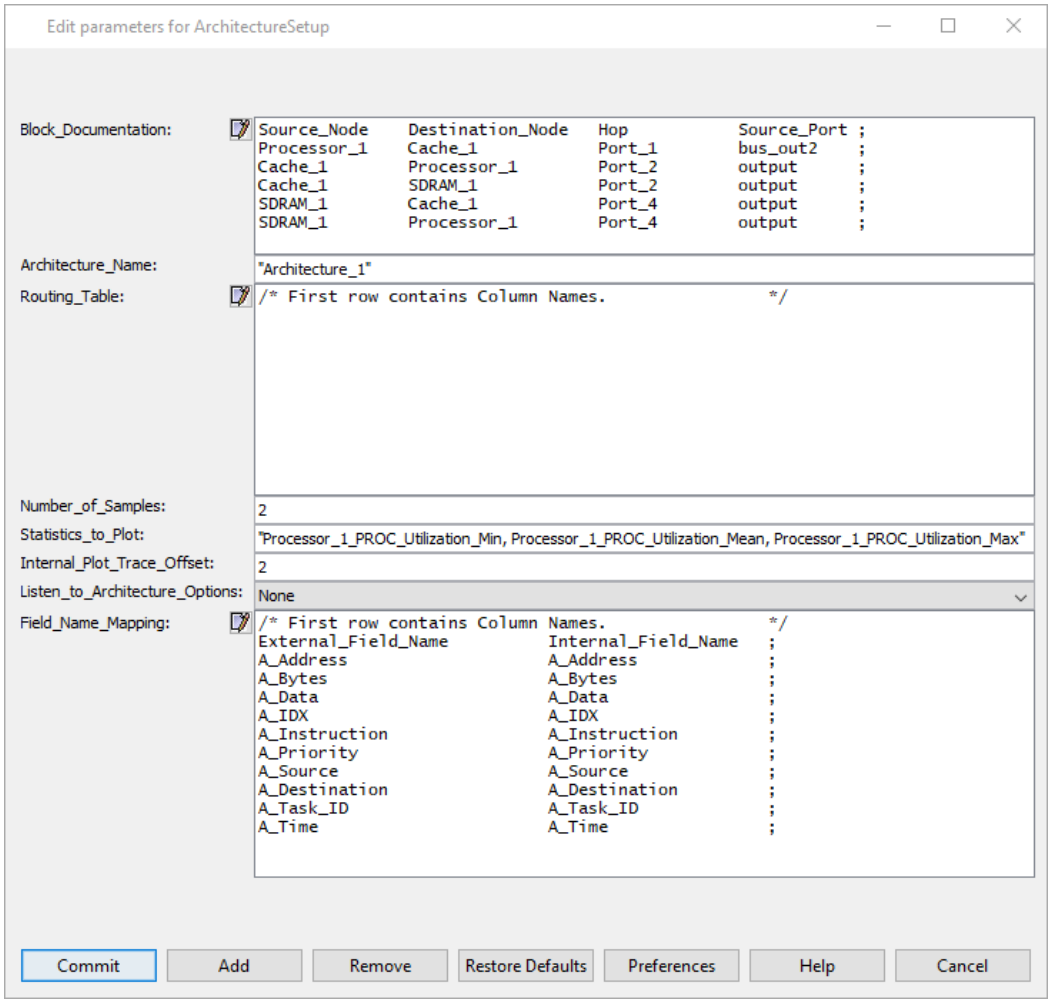
Instruction_Set_Text

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.



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 Interrrupt         */
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       ; /* Procesor 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.

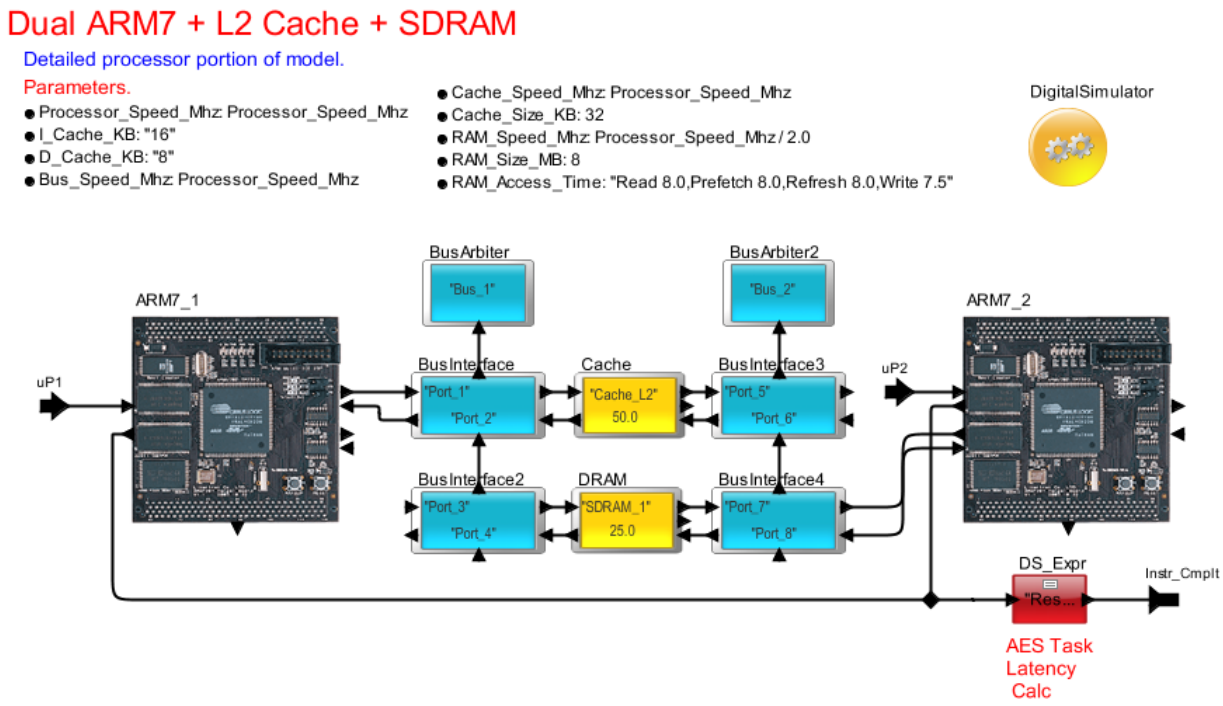


Figure 53 Dual Processor, Dual-Port Cache, Dual-Port SDRAM Routing

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:

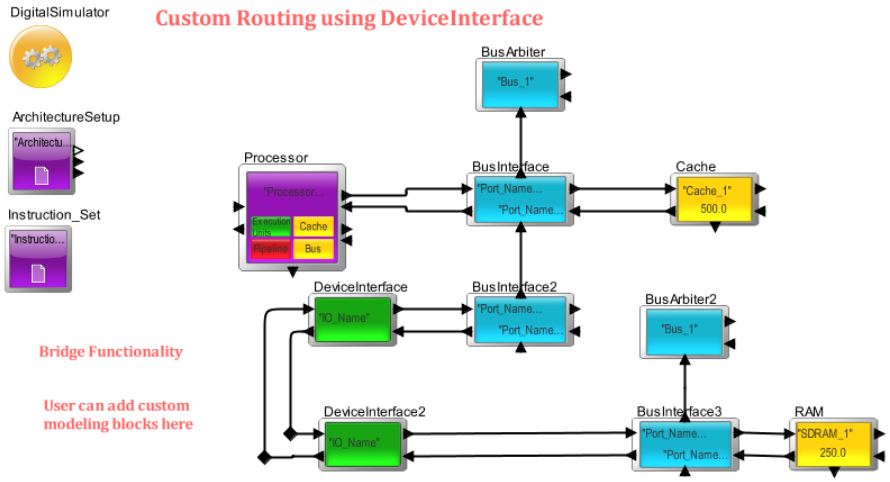


Figure 54 Custom Routing with I_O Block

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 prefetch I_1 internal (L_2) cache at Cycle: 4"
"Processor_1 context switch prefetch I_1 to bus (L_2) cache at Cycle: 4"
"Processor_1 context switch prefetch D_1 internal (L_2) cache at Cycle: 4"
"Processor_1 context switch prefetch D_1 to bus (L_2) cache at Cycle: 4"
"Processor_1 context switch at Cycle: 5"
"Processor_1 context switch at Cycle: 6"
"Processor_1 context switch at Cycle: 7"
"Processor_1 context switch at Cycle: 8"
"Processor_1 context switch at Cycle: 9"
"Processor_1 context switch at Cycle: 10"
"Processor_1 context switch at Cycle: 11"
"Processor_1 context switch at Cycle: 12"
"Processor_1 context switch at Cycle: 13"
"Processor_1 context switch at Cycle: 14"
"Processor_1 context switch at Cycle: 15"
"Processor_1 context switch at Cycle: 16"
"Processor_1 context switch at Cycle: 17"
"Processor_1 context switch at Cycle: 18"
"Processor_1 context switch at Cycle: 19"
"Processor_1 context switch at Cycle: 20"
"Processor_1 context switch at Cycle: 21"
"Processor_1 context switch at Cycle: 22"
"Processor_1 context switch at Cycle: 23"
"Processor_1 context switch at Cycle: 24"
"Processor_1 context switch at Cycle: 25"
"Processor_1 context switch at Cycle: 26"
"Processor_1 context switch at Cycle: 27"
"Processor_1 context switch at Cycle: 28"
"Processor_1 context switch at Cycle: 29"
"Processor_1 context switch at Cycle: 30"
"Processor_1 context switch at Cycle: 31"
"Processor_1 context switch at Cycle: 32"
"Processor_1 context switch at Cycle: 33"
"Processor_1 context switch at Cycle: 34"
"Processor_1 context switch at Cycle: 35"
"Processor_1 context switch at Cycle: 36"
"Processor_1 context switch at Cycle: 37"
"Processor_1 context switch at Cycle: 38"
"Processor_1 context switch at Cycle: 39"
"Processor_1 context switch at Cycle: 40"
"Processor_1 context switch at Cycle: 41"
"Processor_1 context switch at Cycle: 42"
"Processor_1 context switch at Cycle: 43"
"Processor_1 context switch at Cycle: 44"
"Processor_1 context switch at Cycle: 45"
"Processor_1 context switch at Cycle: 46"
"Processor_1 context switch at Cycle: 47"
"Processor_1 context switch at Cycle: 48"
"Processor_1 context switch at Cycle: 49"
"Processor_1 context switch at Cycle: 50"
"Processor_1 context switch at Cycle: 51"
"Processor_1 context switch at Cycle: 52"
"Processor_1 context switch at Cycle: 53"
"Processor_1 context switch at Cycle: 54"
"Processor_1 context switch at Cycle: 55"
"Processor_1 context switch at Cycle: 56"
"Processor_1 context switch at Cycle: 57"
"Processor_1 context switch at Cycle: 58"
"Processor_1 context switch at Cycle: 59"
"Processor_1 context switch at Cycle: 60"
"Processor_1 Bus In:
{A_Address                    = 0,
A_Branch                     = false,
A_Bytes                      = 64,
A_Bytes_Remaining            = 4,
A_Bytes_Sent                 = 64,
A_Command                    = \"Read\",
A_Data                       = \"MyData\",
A_Destination                = \"Processor_1\",
A_Hop                        = \"Processor_1\",
A_IDX                        = 0,
A_Instruction                = {\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
```

```
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\", \"ADD\",
\"ADD\", \"ADD\", \"ADD\", \"ADD\"}
,
A_Interrupt                  = false,
A_Pipeline                   = {0, 0, 0, 0, 0, 0}
,
A_Prefetch                   = true,
A_Priority                   = 0,
A_Proc_Return                = -1,
A_Return                     = -1,
A_Source                     = \"SDRAM_1\",
A_Status                     = \"Port_3_\",
A_TIME                       = 8.0E-9,
A_Task_ID                    = 731470001L,
A_Task_Name                  = \"I_1\",
A_Time                       = 8.7E-8,
A_Variables                  = 16,
BLOCK                        = \"uEngine\",
DELTA                        = 2.0E-9,
DS_NAME                      = \"Processor_DS\",
ID                           = 0,
INDEX                        = 0,
TIME                         = 2.0E-9}"
"Processor_1 External Cache Line Prefetch (I_1731470001) complete ................ at
Cycle: 60"
"Processor_1 context switch at Cycle: 60"
"Processor_1 context switch at Cycle: 61"
"Processor_1 context switch at Cycle: 61"
"Processor_1 context switch at Cycle: 62"
"Processor_1 context switch at Cycle: 62"
"Processor_1 context switch at Cycle: 63"
"Processor_1 context switch at Cycle: 63"
"Processor_1 context switch at Cycle: 64"
"Processor_1 context switch at Cycle: 64"
```

```
"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:

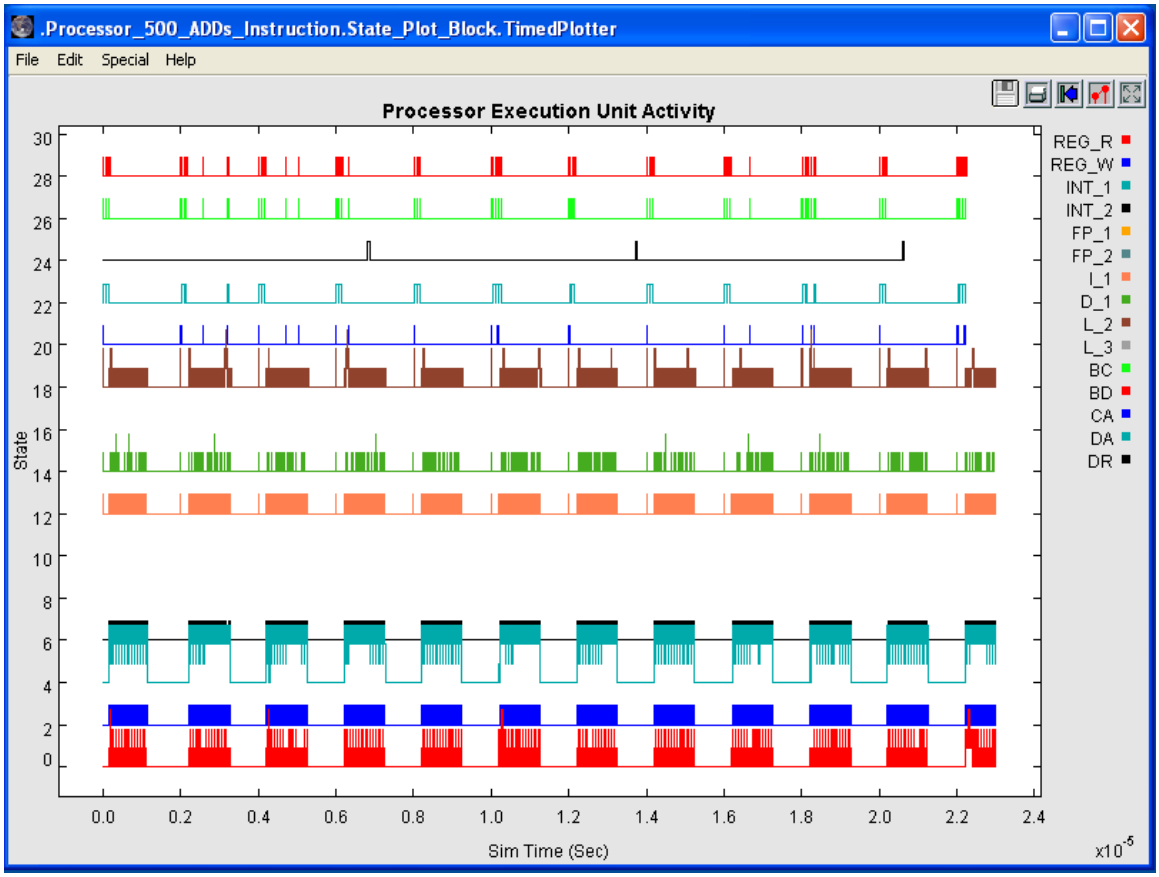


Figure 55 Architecture_State Plot Output

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, I_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 '*' chracter 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 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:

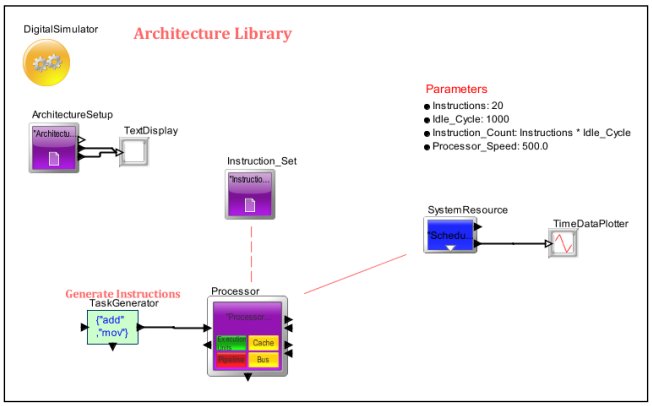


Figure 56 Processor to Scheduler_SW for instruction execution

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.

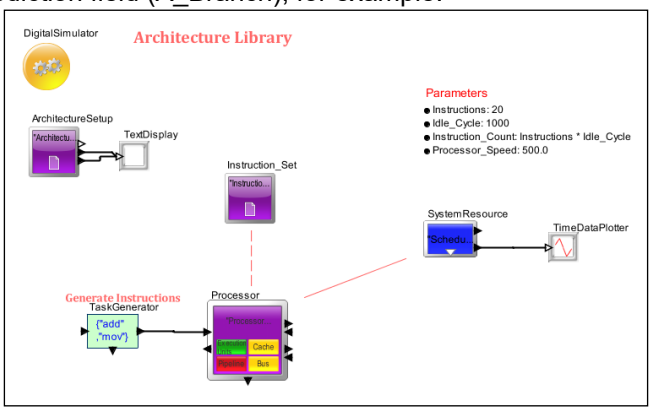


Figure 57 Processor to TaskGenerator for pipeline, instruction execution

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.

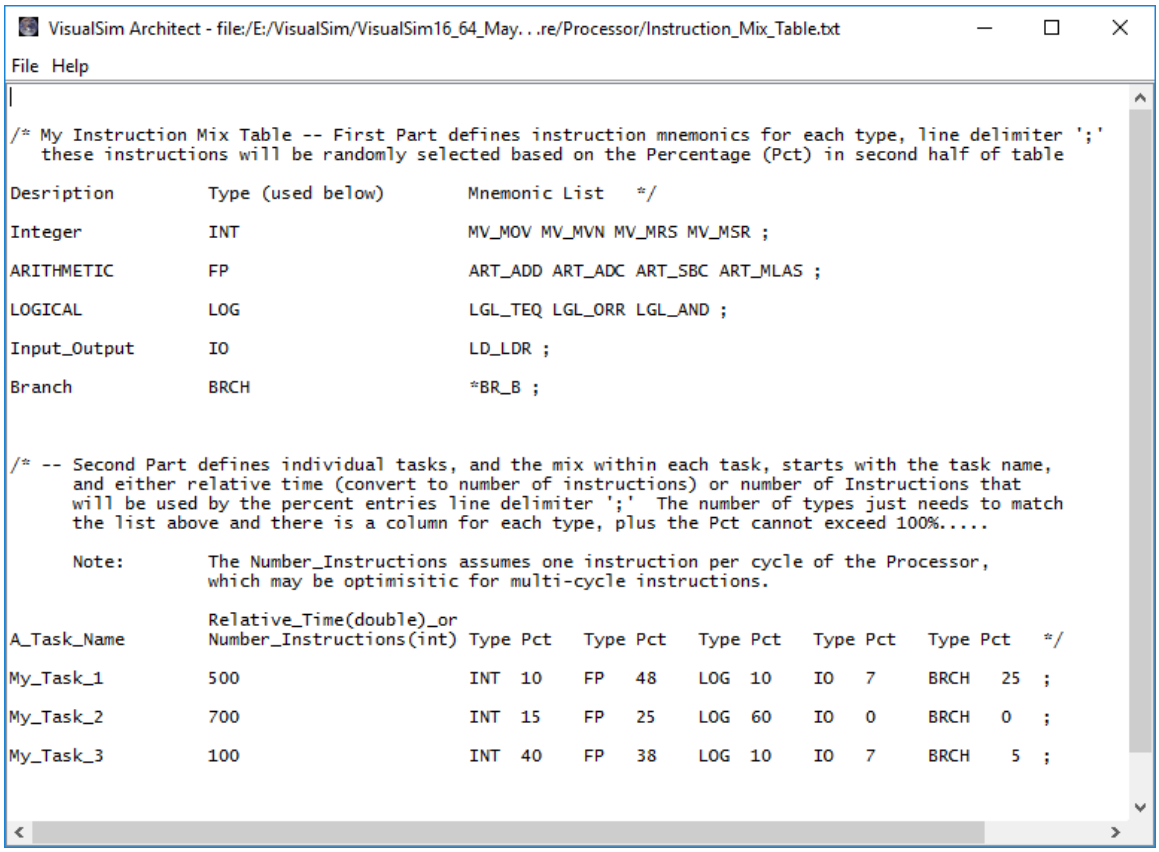
The instructions are placed in the A_Instruction field of the Processor_DS template Data Structure and sent on the output port.

The generated instruction assumes that each instruction takes exactly one cycle to excecute 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



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 mutlitple 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 he 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. 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.

Memory modeling Toolkit

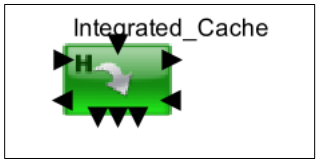
Cache

Block Description:

The Integrated cache library is a configurable cache block that supports address based hit or miss detection. The following features are supported by the integrated cache library:

1. Maintains state for each cache line.
2. Support Cache write policies.
3. Support cache replacement policies.
4. Support Next line prefetching.
5. Support cache Inclusion algorithm.
6. Support Associative mapping cache line placement.
7. Support Power modeling.
8. Support debugging.

This block can be configured as stochastic cache for high level designs.



User can use it is a standalone cache block and can be used in any SoC or complex architecture.

Block Operations:

Hit Miss:
The cycle accurate version of the cache use the A_Address field to determine the cache hit miss operation. The cache maintains the state of each cache line with the Tag, on cache request these are checked against the A_Address value.

For stochastic version it is uses the block parameter Hit_Ratio defined by the user.

Non-Blocking Cache access:
The cache uses Total_Oustanding parameter to define the size of MSHR register. On a cache miss the status of the request is maintained in the MSHR queue and the next request will be processed. The cache can be designed as blocking cache by sending only one outstanding request from the master device.

Associativity:
The current cache supports N way associaticity and Fully associative cache configuration. This make sure the cache lines are allocated in the appropriate set and the replacement of the cache lines restricted to that set.

Write Policies:
The cache support write back and write through policy and by default it uses write allocate method. With write back the cache lines are marked as dirty when a modification is made. It will not be sent to the next level memory unless it is being replaced by a cache miss. With Write through the cache line will be updated to next level memory whenever there is a modification in the cache line. It makes the current cache and next level memory consistent.

Replacement:
The cache will replace a cache line based on the associativity and the replacement policy. The selection of blocks for the replacement algorithm is determined by associativity and the selecting a particular block in that will be determined by the replacement algorithm.

Prefetching:
The cache by default uses next line prefetching and it uses a confidence counter to determine the number of hits must be made before issuing a prefetch, it uses the the parameter "N_Hits_Before_Prefetch". It can be disabled by setting the value to 0. User can also issue multiple prefetch, user can add "N_Lines_Per_Prefetch" to the block and defines the number of cache lines must be prefetched once the confidence counter is satisfied.

Cache Inclusion:
Cache support multiple inclusion policies and it maintains the inclusion of the cache with lower level cache, for example the inclusion policy is maintained by the L2 for L1 cache.

Inclusive policy maintains that all the cache line will be maintained in the L2 when it is loaded in L1. If the line is removed in L2 it will also be removed in L1.

Exclusive policy allows the cache line to be present in only one of the cache, for example the cache loaded in L1 will not be maintained in L2 or other caches.

NINE policy allows caches to maintain the cache line in any level of cache. The cache line in L1 can be present in L2 or it can be removed if it is required, it will not affect the L1 cache.

Block Ports:

to_cache, fm_cache: receive request and send response to the master device.
to_next_cache, fm_next_cache*:* Send memory access request and receive response from the slave device.
Overall_Throughput*:* Provides the instantaneous thoughput during the simulation.
Latency: latency of each packet processed by the cache.
Debug: Debug messages from cache.
VIPT_Addr_Check: Receive TLB response in VIPT mode.

Parameters:

Cache_Name
This will define the name of the cache block. User has to enter the unique name to avoid overlap with other blocks. Eg: "Cache_1"

Cache_Speed_Mhz
Speed of the cache in Mhz. It determines the clock cycle and internal timing of the Cache block. User can analyze the output time using this speed value. Eg: 500.0

Cache_Width_Bytes
This is the maximum width (in terms of byte) that this cache block can process in a single clock cycle. User can use this parameter for analyzing the output and its internal operation.Eg: 4

Cache_Size_Bytes
The overall cache memory size is determined by this parameter. Eg:4096

Block_Size_Bytes
Block size will help the cache to subdivide the memory into set of blocks. Cache will request for the miss based on this block size. Eg: 64

Cache_Type
Defines the Type of cache
"I-Cache" /* "I-Cache", "D-Cache", “I_Cache_No_Prefetch”,”D_Cache_No_Prefetch” */

Stochastic_or_Address_Based
This will determine the mode of operation of this cache block, user can choose Address based for cycle accurate mode of operation. To emulate the cache operation with less accurate user can use the stochastic mode.
Address_Based /*Stochastic,Address_Based*/

Hit Ratio
This will be used in stochastic mode of operation. Cache will use this information to determine the hit or miss of the input Data request. This will not be used in Address mode (Cycle Accurate) of operation. Eg: 0.8

Loop_Ratio
This will be used in stochastic mode for emulating the looping in instruction fetch. This will not be used in Address mode (Cycle Accurate) of operation. Eg: 0.2

Overhead_Cycles
User can include the overhead cycle for each request based on the requirement. An integer will make the overhead cycle common for all read an writes. To differentiate read and write set it as array {Read,Write}, eg: {1,3}

Input_Flow_Control
Input flow control will be enabled with this parameter. Master device should maintain the field “Event_Name” . otherwise the block will throw an error.

Req_Buffer_Size
Input request buffer queue size.

Output_Flow_Control
User can enable Flow control at the slave side by setting this parameter.

Total_Outstanding
This defines the size of MSHR registers for non blocking operation. This size includes both misses and prefetches.

N_Way_Associativity
Determines the Associativity usage and the number of ways in the associativity. if the value is 1 then fully associativity is used. if it greater than 1 the N way associativity will be used in the model.

Cache_Replacement_Policies
Cache blocks will be replaced based on these algorithms.
Pseudo-LRU /*Pseudo-LRU,Least_Recently_Used,Random*/

Cache_Write_Policy
User can choose the Write policy for this cache block for the entire simulation.

Inclusion_Policy
Determines the inclusion policy of the current cache with respect to its lower level cache.
NINE /* Mandatory for L1 cache */, Inclusive, Exclusive,
Incl_with_I_Cache_and_Excl_with_D_Cache /* Applicable only for L2 cache */ ,
Excl_with_I_Cache_and_Incl_with_D_Cache /* Applicable only for L2 cache */ ,
Incl_with_I_Cache_and_NINE_with_D_Cache /* Applicable only for L2 cache */ ,
Excl_with_I_Cache_and_NINE_with_D_Cache /* Applicable only for L2 cache */ ,
NINE_with_I_Cache_and_Incl_with_D_Cache /* Applicable only for L2 cache */ ,
NINE_with_I_Cache_and_Excl_with_D_Cache /* Applicable only for L2 cache */

Miss_Memory_Name
User has to enter the next level memory in this field. This will help this cache block to send the request in case of a miss. If the name is wrong or there is no next level memory with this name then the buses will throw an error as “Destination not found”.

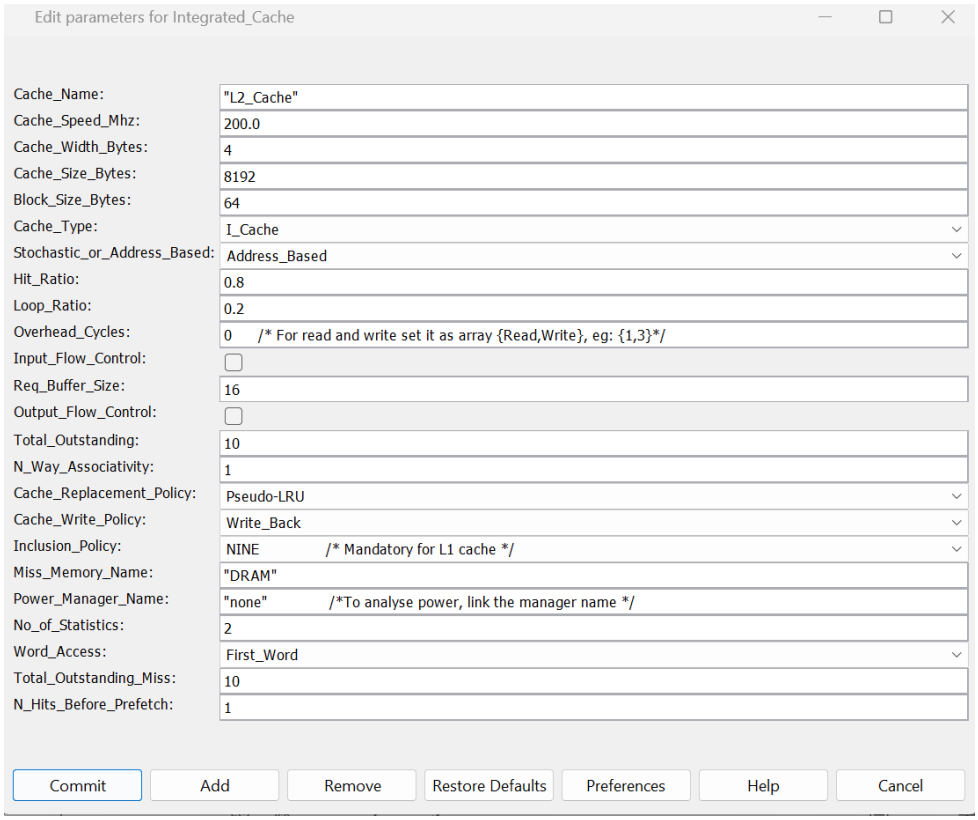
Power_Manager_Name
This parameter helps the user to observe the power consumption of this block along with the other blocks in the architecture. User has to enter the power table name in this field to get the power analysis.

No_of_Statistics
This determines the number of samples of statistics to be sent to the architecture setup block

Word_Access
This determines the word access method in cache when the data is accessed after a cache miss fill.
First Word: reads the first word and send the response, the delay for the remaining words are performed internally.
Critical_Word_First_line_Fill: loads the data from next level memory by reading the critical word first. Once the critical word is written to the memory, send it out to the requesting device.
Last_Word: Write all the words to the cache on cache miss, then send the response only when the last word read is complete.

Total_Outstanding_Miss
This determines the MSHR registers reserved for the cache misses on the total Outstanding count.

N_Hits_Before_Prefetch
This enables are disables the prefetch operation in cache. By default the cache uses next line prefetching. By setting this parameter to 0 disables the prefetch. The values greater than 0 enables the prefetch and the value defines the number of hits should be achieved before issuing a prefetch.



Hidden parameters:

User can add these parameters by selecting Add button in the cache configuration window.

N_Lines_Per_Prefetch (Integer)
This parameter will be used if the prefetch is enabled. It determines the number of lines will be prefetched once the number of hits are satisfied.

VIPT_Mode (bool)
The VIPT mode can be enabled using this parameter. In this case the user have to fork the input and connect one port to the cache input and another port to the TLB bloick. The response of the TLB is connected to the top port of the cache (VIPT_Addr_Check)

Enable_Data (bool)
This enables the actual data storage in the cache. The data storage relies on the address, use this option in Cycle accurate models for proper functionality

Debug (bool)
This will enable the debug message to sent out in Debug port.

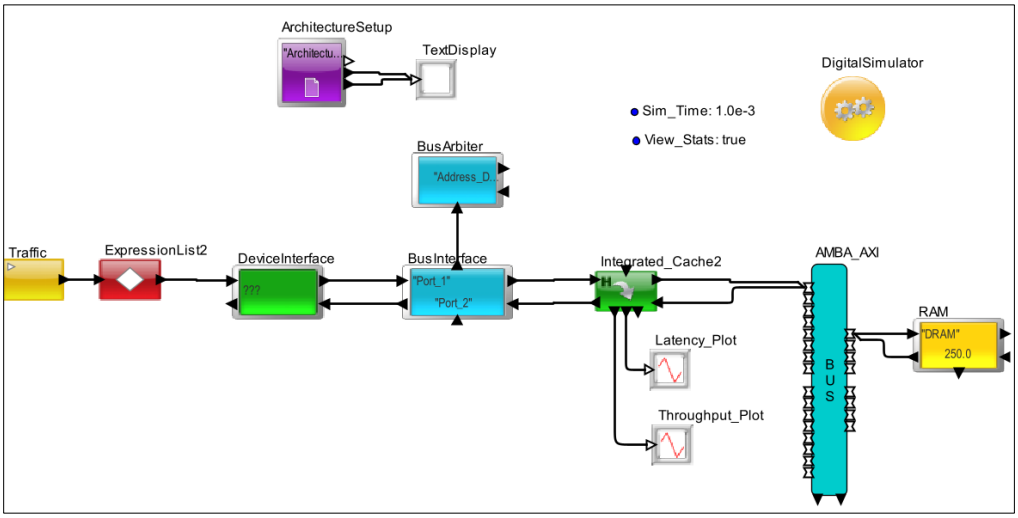
Script_Trace_Slot (Array)
Script_Trace_Enable (bool)
The combination of these two parameters enable the script tracing in the cache block. The Script_Trace_Enable enable or disable this feature. The Script_Trace_Slot defines the start and end time of the tracing in array format. Eg: {1.0e-6,12e-6}

Cache_Miss_Trace (bool)
This enables the cache to generate signals when the miss request is completed in this cache. The master device can use this event signal to define their operations. Event format: (A_Source+"_"+Cache_Name+"_Miss")

Enable_Hello_Messages(bool)
By default the cache will send the help message out. User can disable that by adding this parameter and set to false.

Architecture_Name (String)
The cache uses the default architecture domain name “Architecture_1”. If there are multiple sections of the device with different architecture setup, add this parameter and add the corresponding architecture setup name.

The following image shows a simple design of cache and the connectivity with other blocks.



Statistics:
User can observe the statistics in Architecture setup “Interanl_Stats_Out” port

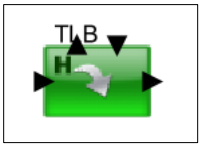
Cache_I_1_A_Hit_Ratio = 81.2,
Cache_I_1_A_Miss_Ratio = 18.8,
Cache_I_1_A_Number_Entered = 1000,
Cache_I_1_A_Number_Returned = 1000,
Cache_I_1_A_Prefetch_Accuracy = 0.0,
Cache_I_1_A_Prefetch_Completed = 433,
Cache_I_1_A_Prefetch_Issued = 433,
Cache_I_1_A_Prefetch_Useful = 0,
Cache_I_1_Buffer_Occupancy = 0,
Cache_I_1_Buffer_Overflow = 0,
Cache_I_1_Latency_Avg = 3.2569999999999E-8,
Cache_I_1_Latency_Max = 1.38E-7,
Cache_I_1_Latency_Min = 7.9999999999091E-9,
Cache_I_1_Outstanding_Requests = 0,
Cache_I_1_Read_Hit_Ratio = 81.2,
Cache_I_1_Read_MBs = 0.032,
Cache_I_1_Read_MBs_per_Second = 64.00000128,

Cache_I_1_Total_Blocks_Evicted = 261,
Cache_I_1_Total_Blocks_Write_Backed = 0,
Cache_I_1_Total_MBs = 0.071744,
Cache_I_1_Total_MBs_per_Second = 143.4880028697601,
Cache_I_1_Total_Reads = 1000,
Cache_I_1_Total_Writes = 0,
Cache_I_1_Utilization = 1.8557000352583,
Cache_I_1_Write_Hit_Ratio = NaN,
Cache_I_1_Write_MBs = 0.039744,
Cache_I_1_Write_MBs_per_Second = 79.48800158976}

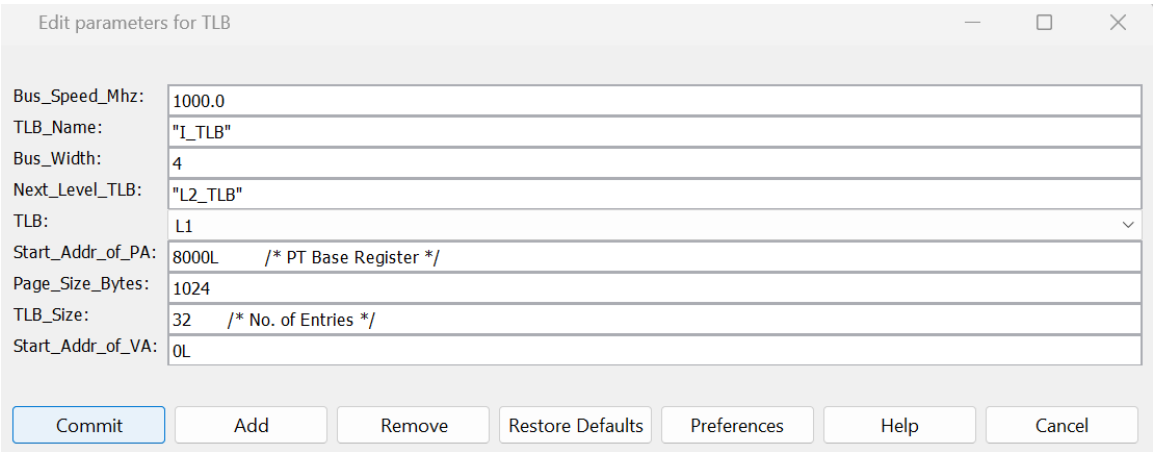
TLB

Block Description:

The TLB block emulates the address translation in the design by keeping track of entries and replace them when there is no invalid entries for new translations. This acts like a cache and it only focus on the timing aspects of the TLB. The data stored about the address translation is based on the Start_Address_of_VA and Start_Address_of_PA. The Page table is usually designed with a stochastic RAM block. The L1 TLBs are connected to L2 TLB virtually. But the L2 TLB connected to the Page table (RAM block) through miss_out and miss_in ports.



Ports:
Input : receives a data structure that contains A_Addres field
Output : provides a response of address translation, A_Addres will contain translated Address and Virtual_Address field will contain the original virtual address.
miss_out : send the request to next level TLB
miss_in : receive response from next level TLB



Parameters:

Bus_Speed_Mhz
It determines the speed of the bus between current TLB and the next level TLB. User does not have to connect a bus between them. The block takes care of the delay internally.

TLB_Name
Name of this TLB block. Must be a unique name.

Bus_Width
Determines the size of the bus width between the current TLB and next level TLB.

Next_Level_TLB
Name of the next level TLB.

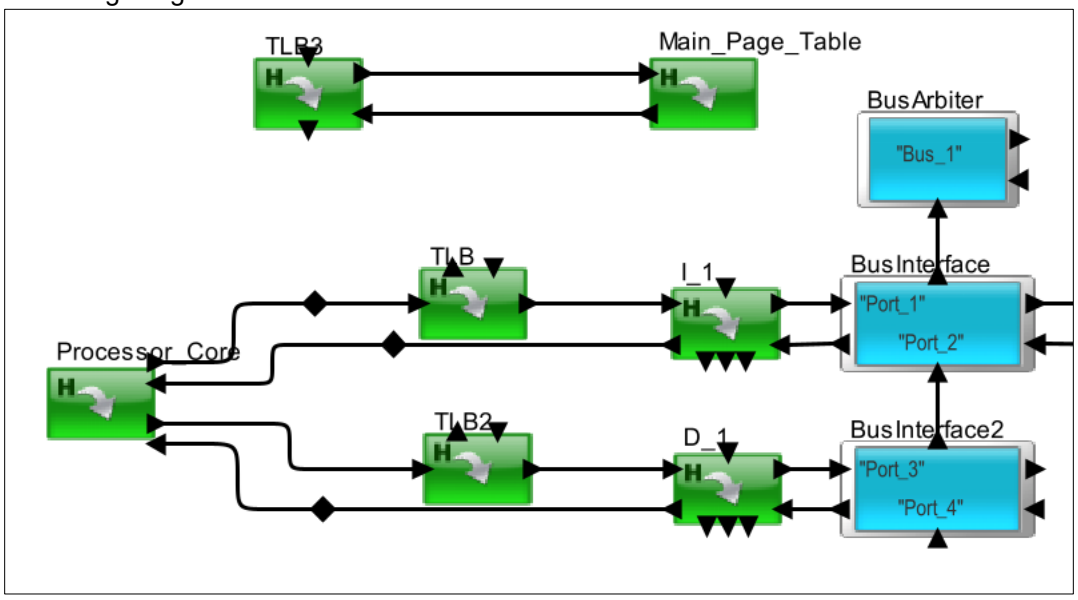
TLB
Defines the TLB type. Options are L1, L2 and L3.

Start_Addr_of_PA
Starting address of the physical address. This is used for translating the virtual address.

TLB_Size
Size of the TLB in terms of entries.

Start_Addr_of_VA
Starting address of the Virtual address. This is used for translating the virtual address.

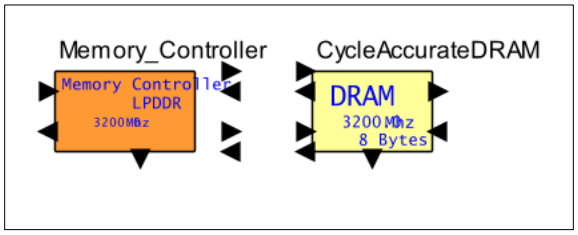
The following image shows the connection of L1 and L2 TLB blocks.



Cycle Accurate Memory controller and DRAM.

Block Description:

This is a low level design of DRAM and the memory controller. The memory controller receives inputs from various devices through an interconnect such as AXI or NoC and buffers them in the command queue. It uses First come First Ready (FCFR) Scheduling algorithm by default and generate the appropriate commands to the DRAM for proper memory access operation. The DRAM block emulates the banked memory organization with refresh handling and IO bus transfers between DRAM IO buffer and memory controller.
The cycle accurate DRAM supports different memory technologies. The user have to configure the timing according the memory type and the data rate. The user can select DDR3 to DDR5 and LPDDR3 to LPDDR5.



Block Operation:

Memory scheduling

The memory controller loads the request either in separate Read/Write queues or a unified queues and maintains the bank and row information of each request. It uses FCFR scheduling algorithm, where the request to the open page is scheduled first and if there are no requests that can be scheduled in that cycle to an open page, the request to a closed page (activation=RCD or precharge=RP +activation=RCD) will be scheduled.
The memory controller will make sure the sequential Read/Write commands will be issued with a interval of CCD (CAS to CAS delay). If the next command is different from previous command such as read to write or write to read then the appropriate delay will be scheduled for the command before it is issued to the DRAM.

Similarly, the activation and precharge commands are issued with the corresponding interval. For activations it will maintain RRD (RCD to RCD Delay) between two activation commands. For Precharge command it will maintain RAS delay.
Along with this the precharge can be issued with a constraint based on the previous command to that bank. For a read command it will issue the precharge after RTP(Read to Precharge) delay, similarly for the write command the precharge will be issued after WR (Write Recovery)delay.

The memory controller maintains the following flow for scheduling the commands. The delays are aligned such a way to compare with different cases.

Activation to a idle bank:
Generic Timings: *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).

Activating a new Row in a bank:
Generic Timings: [*tRAS* | *(tRTP or tWR)*] + *tRP* + *(tRL or tWL)* + *Data transfer tim*e (through channel).

Activating a new Row in different bank and different Bank group:
Differnt bank (Closed bank) :
[*tRRD* | *tRC*] + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Differnt bank (Open bank) :
[*tRRD* | *tRAS* | *(tRTP or tWR)*]+ *tRP* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Same bank (Closed bank) :
{ *tRC* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Same bank (Open bank) :
[*tRAS* | *(tRTP or tWR)*]+ *tRP* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).

(Applicable only for DDR4, DDR5 and LPDDR5)
Different bank group (closed bank) :
[*tRRD_S* | *tRC*] + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Different bank group (open bank) :
[*tRRD_S* | *tRAS* | *(tRTP or tWR)*]+ *tRP* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Same BG Differen bank (closed bank) :
[*tRRD_L* | *tRC*] + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Same BG same bank (closed bank) :
[*tRC* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Same BG different bank (open bank) :
[*tRAS* | *(tRTP or tWR)*]+ *tRP* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).
Same BG same bank (open bank) :
[*tRAS* | *(tRTP or tWR)*]+ *tRP* + *tRCD* + *(tRL or tWL)* + *Data transfer time* (through channel).

Consecutive Read or Write:
Generic Timings: tCCD + *(tRL or tWL)* + *Data transfer time* (through channel)

(Applicable only for DDR4, DDR5 and LPDDR5)
Different bank group : *tCCD_S* + *(tRL or tWL)* + *Data transfer time* (through channel)
Same bank group : *tCCD_L* + *(tRL or tWL)* + *Data transfer time* (through channel)

Read to Write or Write to Read:
Generic Timings
Read to Write : BL/2 + *(tRL or tWL)* + *Data transfer time* (through channel)
Write to Read : Write completion + *WTR* + *(tRL or tWL)* + *Data transfer time* (through channel)

(Applicable only for DDR4, DDR5 and LPDDR5)
Different bank group : *Write completion* + *WTR_S* + *(tRL or tWL)* + *Data transfer time* (through channel)
Same bank group : *Write completion* + *WTR_L* + *(tRL or tWL)* + *Data transfer time* (through channel)

Multiple activation are restricted by the ***tFAW*** limit. if there is a 5th request for a new activation the with in the tFAW window, the 5th request will be issued in the next window.

Refresh:

The DRAM support the all bank refresh for DDR (DDR1 to DDR5) memory technologies and per bank refresh for LPDDR(LPDDR1 to 5) memories. The DDR5 can be configured to same bank refresh by enabling the Same_Bank_Refresh parameter in DRAM.

Ports:

Memory controller ports:

rd_wr_data_fm_bus: receive request from master devices.
rd_data_wr_rep_to_bus: sends the response to the master devices
Status: send debug messages out.
rd_wr_data_to_mem: send memory access commands and data to DRAM
rd_data_fm_mem: receive data response from DRAM
ctrl_to_mem: Send control commands for activate and precharge to DRAM
ctrl_fm_mem: receive ack commands from DRAM

DRAM ports:

Port_1: receive memory access commands from memory controller. Connects to rd_wr_data_to_mem port
Port_2: sends the response to memory controller. Connects to rd_data_fm_mem port.
fm_ctrl: receive control command from memory controller.
to_ctrl: send ack to memory controller.
port: debug port of DRAM.
port_3: Read/write access is forwarded to this port if EnableExternalData parameter is set
port_4: receive responser from external device if EnableExternalData parameter is set.

Parameter:

Memory Controller Parameters:

Controller_Name

Name of this Memory Controller. This is a unique name in the entire model.
Eg: "DDR_Controller" /* DDR4 typical shown */

DRAM_Type

Determines the DRAM type so that the memory controller can send appropriate commands to DRAM.
Eg: "DDR4" /* SDR, DDR, DDR2, DDR3, DDR4, DDR5, LPDDR, LPDDR2, LPDDR3, LPDDR4, LPDDR5 */

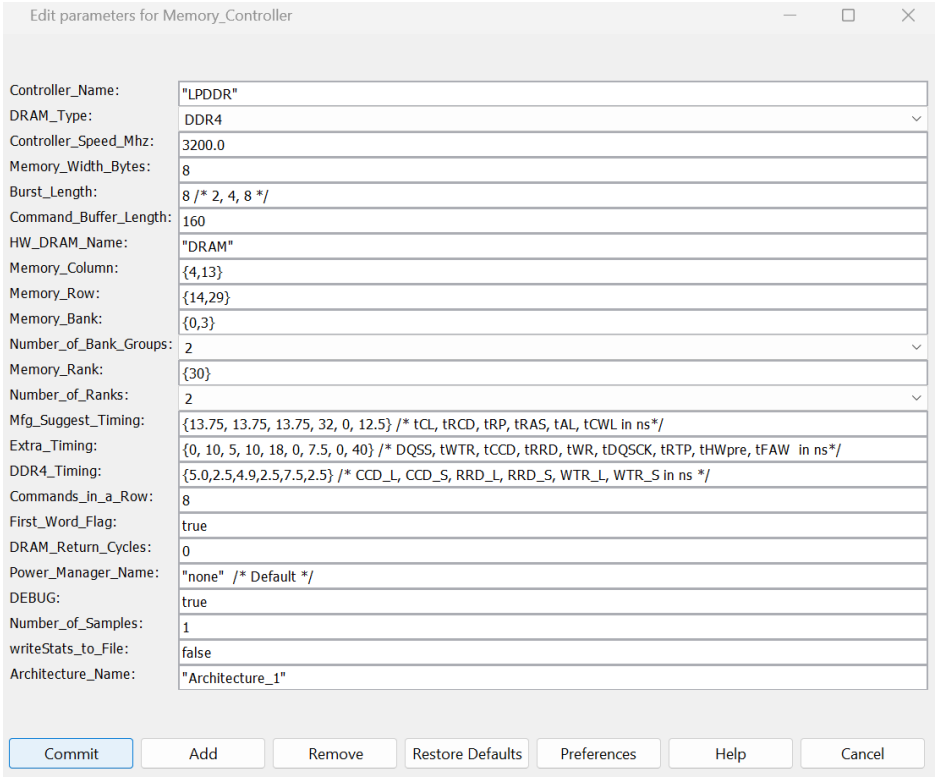
Controller_Speed_Mhz

Speed of this controller in Mhz. It determines the scheduling and issue rate to DRAM. (it can be different from DRAM data rate)
Eg: 3200.0 /* DDR4 typical */

Memory_Width_Bytes

Determines the word size in memory. (x8 - 1 Byte, x16 - 2 Byte, x32 - 4 Byte)
Eg: 1 /* DDR4 typical */

Burst_Length
Determines the number of bursts that will be prefetched through a single command.
Eg: 8 /* 2, 4, 8, 16 DDR4 typical shown */



Command_Buffer_Length
Determines the size of command queue. Array of 2 integer defines a split queue for Read and Write. Single integer defines the unified queue which can hold both Read and Write in single queue but maintian equal space for read and write.
Eg: {160,160} /* Split Queue {Read, Write} - {32,32}*/ /* Unified Queue(Read and Writ)e - 64

HW_DRAM_Name
Name of the DRAM block that is connected to this memory controller.
Eg: "DRAM"

Memory_Column
It defines the specific or range of bits/pins used for defining the total columns. The format is {start_bit, End_bit} It will be used for address decoding.
Eg: {4,13} /* DDR4 typical shown */ /*{{4},{6,13}}*/

Memory_Row
It defines the specific or range of bits/pins used for defining the total rows. The format is {start_bit, End_bit} It will be used for address decoding. User should make sure the bits are not overlapped with Column,Rank and Bank.
Eg: {14,29} /* DDR4 typical shown */ /*{{5},{14,29}}*/

Memory_Bank
It defines the specific or range of bits/pins used for defining the total banks. The format is {start_bit, End_bit} It will be used for address decoding. User should make sure the bits are not overlapped with Row,Rank and Column.
Eg: {0,3} /* DDR4 typical shown */ /*{{0},{2,3}}*/

Number_of_Bank_Groups
it will configure the memory organization as bank groups. Each bank group will be subdivided into set of banks. (applicable only for DDR4, DDR5 and LPDDR5)
Eg: 2

Memory_Rank
It defines the specific or range of bits/pins used for defining the total ranks. The format is {start_bit, End_bit} It will be used for address decoding. User should make sure the bits are not overlapped with Row,Bank and Column.
Eg: {30}

Number_of_Ranks
Currentrly not used. it will be supported in future
Eg: 2

Mfg_Suggest_Timing
Array of timing, based on tCL, tRCD, tRP, tRAS. Consistent with JEDEC JESD 209 standard. The recommended timing for the DRAM as an array of six double values (in nano sec) . These six entries refer to the following settings, in this order: tCL - tRCD - tRP - tRAS - tAL - tCWL.
CL = CAS Latency time (Read);
tRCD = DRAM RAS# to CAS# Delay;
tRP = DRAM RAS# Precharge;
tRAS = Active to Precharge delay.
tAL = Additive latency in read and write.
tCWL = CAS latency for write.

Eg: {13.75, 13.75, 13.75, 32, 0, 12.5} /* tCL, tRCD, tRP, tRAS, tAL, tCWL in ns*/ /* DDR4 typical shown */

Extra_Timing
Extra timing for Memory Controller. Array of 9 added timing values: DQSS, tWTR, tCCD, tRRD, tWR, tDQSCK, tTRP, tHWpre, tFAW to further refine memory timing. This is an array of additional parameters used by vendors to describe other intermediate latencies. The format is an array of double values in ns.
DQSS is the time for the BiDirectional bus strobe.
tWTR is the minimum time interval between end of WRITE and the start of READ command.
tCCD is the minimum time interval between two sequential reads./writes.
tRRD is the minimum time interval between successive ACTIVE commands to different banks.
tWR is minimum time interval between end of WRITE and PRECHARGE command.
tDQSCK is the data queue strobe clock.
tRTP is Read to Precharge timing.
tHWpre can be used to add cycles to tRAS, tRTP, tWR parameters, default is 0.
tFAWis the time period in which only 4 activates can be issued. if 5th activate needs to be issued, it must be issued in the next FAW window

Eg: {0, 7.5, 0, 0, 15, 0, 7.5, 0, 21} /* DQSS, tWTR, tCCD, tRRD, tWR, tDQSCK, tRTP, tHWpre, tFAW in ns*/ /* DDR4 typical shown */

DDR4_Timings
Defines the timing information for DDR4, DDR5 and LPDDR5.
CCD_L - CCD long defines the CCD between sequential access that goes to same bank group
CCD_S - CCD Short defines the CCD between sequential access that goes to different bank group
RRD_L - RRD long defines the RRD between activates that goes to same bank group
RRD_S - RRD Short defines the RRD between activates that goes to different bank group
WTR_L - WTR long defines the WTR between end of write to start of read that goes to same bank group
WTR_S - WTR Short defines the WTR between end of write to start of read that goes to different bank group
Eg: {5.0,2.5,4.9,2.5,7.5,2.5} /* CCD_L, CCD_S, RRD_L, RRD_S, WTR_L, WTR_S in ns */

Commands_in_a_Row
Maximum number of Read/Write sequential to be searched and transfered from the command buffer before switching to the other type of command.
Eg: 8

First_Word_Flag
Initiate the the response once the first word is obtained from DRAM. if it is disabled the response will be initiated at the last word.
Eg: true

DRAM_Return_Cycles
Extra hardware cycles before the Read data is sent back to the bus.
Eg: 0

Power_Manager_Name
Name of the power manager. "none" will disable the power. Valid PowerTable name will enable the power.
Eg: "none" /* Default */

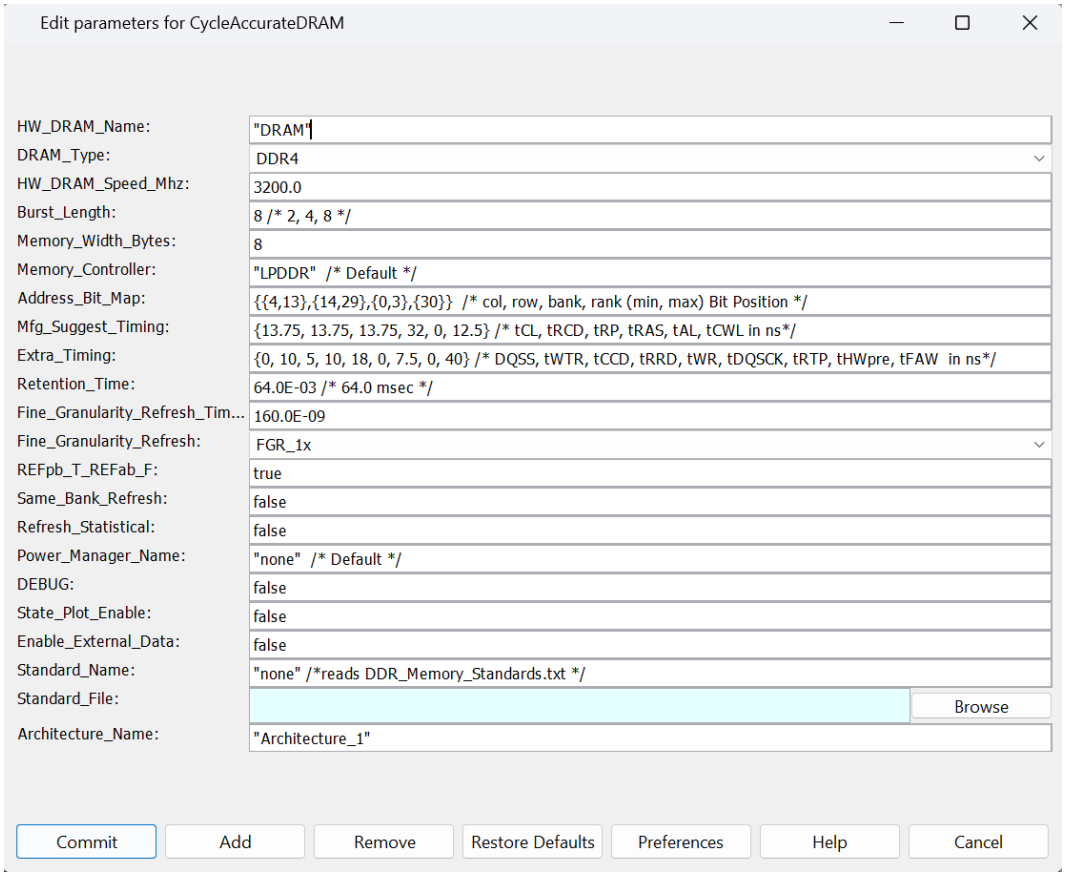
DEBUG
Enable or disable the debug option
Eg: true

Number_of_samples
It defines the number of statistics to be generated for memory controller and DRAM.
Eg: 1

WriteStats_to_File
by setting true, statistics will be written in a file for memory controller..
Eg: true

Architecture_Name
The name of the ArchitectureSetup that this block is associated with. If the Architecture_setup is not available, a error will be thrown.
Eg: "Architecture_1"

Cycle Accurate DRAM parameters:



HW_DRAM_Name
Unique name for the CycleAccurateDRAM
"DRAM"

DRAM_Type
This is determines the type of memory.
Eg: "DDR4" /* SDR, DDR, DDR2, DDR3, DDR4, DDR5, LPDDR, LPDDR2, LPDDR3, LPDDR4, LPDDR5 */

HW_DRAM_Speed_Mhz
Speed of CycleAccurateDRAM in Mhz (Data Rate)
Eg: 3200.0 /* DDR4 typical shown */

Burst_Length
Number of data transfer in bursts.
Eg: 8 /* DDR4 typical shown */

Memory_Width_Bytes
Width of the memory interface in bytes.
Eg: 1 /* DDR4 typical shown */

Memory_Controller

Name of the memory controller connected to this DRAM.
Eg: "none" /* Default */ /* eg: "DDR_Controller" */

Address_Bit_Map
Determines the size and organization of the memory. it must be 2D array with 4 internal array entries. Each internal array must have the range of address pins (or specific bits) {{Column range}, {Row_Range},{Bank_Range},{Rank_Range}}
Eg: {{4,13},{14,29},{0,3},{30}} /* col, row, bank, rank (min, max) Bit Position */ this configures the memory as single rank 8GB package. 4 bank groups, 4 banks per bank group, and 1KB page size.

Mfg_Suggest_Timing
The recommended timing for the DRAM as an array of six double values (in nano sec) . These six entries refer to the following settings, in this order: tCL - tRCD - tRP - tRAS - tAL - tCWL.
CL = CAS Latency time (Read);
tRCD = DRAM RAS# to CAS# Delay;
tRP = DRAM RAS# Precharge;
tRAS = Active to Precharge delay.
tAL = Additive latency in read and write.
tCWL = CAS latency for write.
Eg: {13.75, 13.75, 13.75, 32, 0, 12.5} /* tCL, tRCD, tRP, tRAS, tAL, tCWL in ns*/ /* DDR4 typical shown */

Extra_Timing
Extra timing for Memory Controller. Array of 9 added timing values: DQSS, tWTR, tCCD, tRRD, tWR, tDQSCK, tTRP, tHWpre, tFAW to further refine memory timing. This is an array of additional parameters used by vendors to describe other intermediate latencies. The format is an array of double values in ns. DQSS is the time for the BiDirectional bus strobe. tWTR is the minimum time interval between end of WRITE and the start of READ command. tCCD is the minimum time interval between two sequential reads./writes. tRRD is the minimum time interval between successive ACTIVE commands to different banks. tWR is minimum time interval between end of WRITE and PRECHARGE command. tDQSCK is the data queue strobe clock. tRTP is Read to Precharge timing. tHWpre can be used to add cycles to tRAS, tRTP, tWR parameters, default is 0. tFAW is the time period in which only 4 activates can be issued. if 5th activate needs to be issued, it must be issued in the next FAW window
Eg: {0, 7.5, 0, 0, 15, 0, 7.5, 0, 21} /* DQSS, tWTR, tCCD, tRRD, tWR, tDQSCK, tRTP, tHWpre, tFAW in ns*/ /* DDR4 typical shown */

Retention_Time
Refresh window for entire memory to be refreshed once.
Eg: 64.0E-3 /* LPDDR2 and DDR3 typical shown */

Fine_Granularity_Refresh_Time
Refresh time(tRFC) for each refresh command
Eg: 166.0E-09

Fine_Granularity_Refresh
Fine tune the refresh interval. (Only applies to DDR4 and DDR5. For other DRAM type it is by default FGR_1x)
Eg: FGR_1x /*None, FGR_1x, FGR_2x, FGR_4x, FGR_8x, FGR_16x*/

REFpb_T_REFab_F
Select Per Bank or All Bank refresh. "true" enable the Per Bank refresh and "false" enables All Bank refresh.

Eg: true

Same_Bank_Refresh
Enable same bank refresh (Only used in DDR5)
Eg: false

Refresh_Statistical
Enable or disable statistical refresh. (stochastic version of refresh).
Eg: false

Power_Manager_Name
Name of the power manager. "none" will disable the power. Valid PowerTable name will enable the power
Eg: "none" /* Default */ /*eg: "Manager_1" */

Debug
If true, an activity trace is output on port. If false, no output is received. This provides the user information on the actions with the block.
Eg: true /*Debug data */

State_Plot_Enable
Enable or disable the timing diagram of this DRAM.
Eg: false

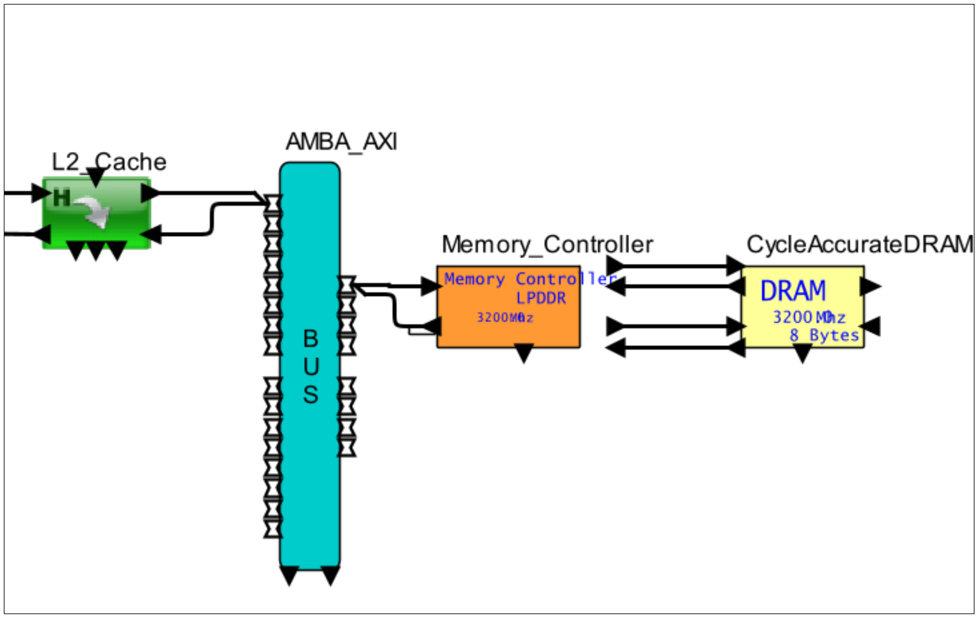
Enable_External_Data
If true, the data structure is sent out on port_3. The DRAM does not continue for that data structure, until it receives the data structure back on port_4. If false, port_3 is bypassed.
Eg: false /* No output on port_3 */

Standard_Name
(Only used if the standard timings are configured through a file). Memory standard name to select the set of timings from the standard file.
Eg: "none" /*eg: DDR_Memory_Standards.txt */

Standard_File
Another way to configure the timing values using a tex file. The text file must contain the timings along with the memory standard name.
option to browse and select the file.

Architecture_Name
Unique architecture name, typically “Architecture_1”
Eg: "Architecture_1"

The following images shows a simple contection between DRAM and interconnect devices.



Statistics:

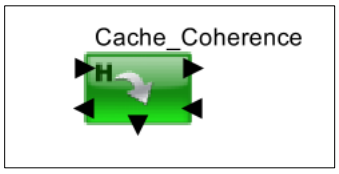
MC_DRAM_DRAM_00_Total_Requests = 56,
MC_DRAM_DRAM_01_Completed_Requests = 56,
MC_DRAM_DRAM_02_Total_MB_per_Second = 35.8522105458677,
MC_DRAM_DRAM_03_Total_Bytes = 3584,
MC_DRAM_DRAM_04_Read_Bytes = 3584,
MC_DRAM_DRAM_05_Write_Bytes = 0,
MC_DRAM_DRAM_06_Read_MB_per_Second = 35.8522105458677,
MC_DRAM_DRAM_07_Write_MB_per_Second = 0.0,
MC_DRAM_DRAM_08_Read_Requests = 28,
MC_DRAM_DRAM_09_Write_Requests = 0,
MC_DRAM_DRAM_10_Max_Queue_Usage = 4,
MC_DRAM_DRAM_12_Queue_Removal_Position = {56, 0},
MC_DRAM_DRAM_14_Request_Overflow = 0,
MC_DRAM_DRAM_15_Read_Request_Overflow = 0,
MC_DRAM_DRAM_16_Write_Request_Overflow = 0,
MC_DRAM_DRAM_17_Reads_Per_Bank = {56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
MC_DRAM_DRAM_18_Writes_Per_Bank = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
MC_DRAM_DRAM_19_Activates_Per_Bank = {13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
MC_DRAM_DRAM_20_Precharges_Per_Bank = {12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
MC_DRAM_DRAM_21_Total_Activates = 13,
MC_DRAM_DRAM_22_Total_Precharges = 12,
MC_DRAM_DRAM_23_Total_RRD_L_S = {{0, 0}, {12, 0}},
MC_DRAM_DRAM_24_Total_CCD_L_S = {{55, 26}, {0, 0}},
MC_DRAM_DRAM_25_Total_WTR_L_S = {{0, 0}, {0, 0}},
MC_DRAM_DRAM_26_Total_RTP_WR_RAS_RTW = {{0, 0}, {0, 0}, {0, 0}, {0, 0}},
MC_DRAM_DRAM_27_Refresh_Percent = 1.920016,
MC_DRAM_DRAM_28_DRAM_Delay_Min = 1.6249E-8,

MC_DRAM_DRAM_29_DRAM_Delay_Max = 1.6251000000001E-8,
MC_DRAM_DRAM_30_DRAM_Delay_Mean = 1.6250660714286E-8,
MC_DRAM_DRAM_31_DRAM_Delay_StDev = 5.0978943365943E-13

Cache Coherence Block

Block Description:

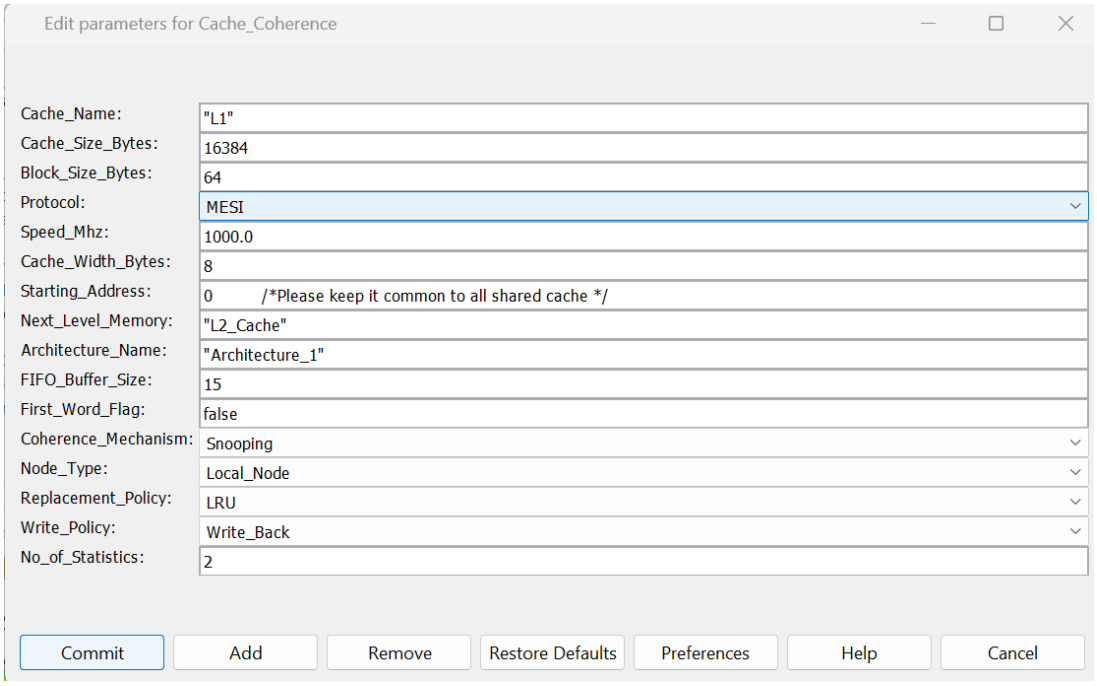
The cache coherence block enables multiple cores to share the memory and maintain coherency with all the caches that are using shared memory address range. This block by default uses MESI protocol and it support snooping and directory based architecture.



Ports:

Input: receive input from from master devices
Output: provides response to the master device
To_Bus: send snooping commands to the inteconnect or home node
Frm_Bus: Response from the remote node or home node
Stats: not used

Parameters:



Cache_Name
This will define the name of the cache block. User has to enter the unique name to avoid overlap with other blocks.
Eg: "Cache_1"

Cache_Size_Bytes
The cache memory size is determined by this parameter.
Eg: 4096

Block_Size_Bytes
Block size will help the cache to subdivide the memory into set of blocks. Cache will request for the miss based on this block size. Eg: if cache size is 8192 bytes and the block size is 64Bytes then the memory will be organized as 128 blocks. These blocks can be accessed by the source and all shared caches.
Eg: 64

Protocol
This determines the coherence protocol for the cache. Based on the option internal operation can vary.
Eg: MESI /* MESI, MOESI, MESFI*/

Speed_Mhz
Speed of the cache in Mhz. It determines the clock cycle and internal timing of the Cache block. User can analyze the output time using this speed value.
Eg: 500.0

Cache_Width_Bytes
This is the maximum width (in terms of byte) that this cache block can process in a single clock cycle. User can use this parameter for analyzing the output and its internal operation.
Eg: 4

Starting_Address
Defines the starting address for shared memory region.
Eg: 0

Next_Level_Memory
User has to enter the next level memory in this field. This will help this cache block to send the request in case of a miss. If the name is wrong or there is no next level memory with this name then the buses will throw an error as “Destination not found”.
Eg: "L2_Cache"

Architecture_Name
This defines the architecture setup name. This will help the tool to observe the model and make sure the model built is correct. Please keep the name same as the Architecture setup block.
Eg: "Architecture_1"

FIFO_Buffer_Size
Currently not used.
Eg: 16

First_Word_Flag
This will determine the output transaction sequence. If this is true then the first word(Cache width length) of the requested bytes will be sent out. If not, then the last word (requested byte length) will be sent out..

Eg: true

Coherence_Mechanism
This determines the coherence method,user can choose either Snooping or Home node based mechanism.
Eg: Snooping /* Snooping, Directory_Based */

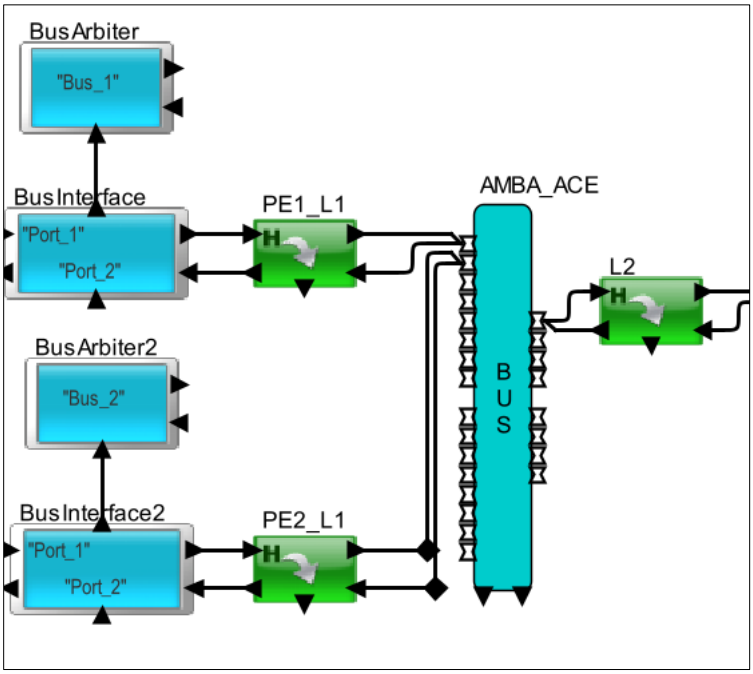
Node_Type
This determines the cache as local node or home node.
Eg: Local_Node /*Local_Node, Home_Node */

Replacement_Policy
Cache blocks will be replaced based on these algorithms.
Eg: Pseudo-LRU /*Pseudo-LRU, LRU, Random */

Write_Policy
User can choose the Write policy for this cache block for the entire simulation. The operation of these policies is explained in this document.
Eg: Write_Through /*Write_Through, Write_Back */

No_of_Statistics
User can analyze the operation in the cache by observing the Statistics of the cache. It defines the number of samples of statistics for this cache block. User can view statistics by connecting a text display to stats port at bottom of the block.
Eg: 2

The following image shows the connection of snooping protocols with cache coherence block.



Statistics:

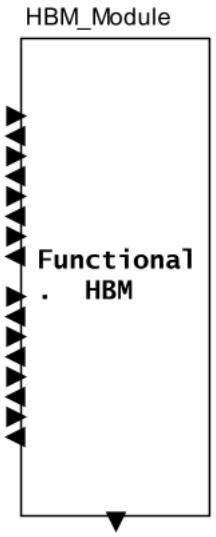
Coherence_Cache_Proc2_L1_A_Hit_Ratio = 83.3766233766234,

Coherence_Cache_Proc2_L1_A_Miss_Ratio = 16.6233766233766,
Coherence_Cache_Proc2_L1_A_Read_MBs = 0.030112,
Coherence_Cache_Proc2_L1_A_Read_MBs_per_Second = 60.22400120448,
Coherence_Cache_Proc2_L1_A_Total_MBs = 0.033856,
Coherence_Cache_Proc2_L1_A_Total_MBs_per_Second = 67.71200135424,
Coherence_Cache_Proc2_L1_A_Write_MBs = 0.003744,
Coherence_Cache_Proc2_L1_A_Write_MBs_per_Second = 7.48800014976,
Coherence_Cache_Proc2_L1_Fill_Buffer_Occupancy = 0,
Coherence_Cache_Proc2_L1_Input_Buffer_Occupancy = 0,
Coherence_Cache_Proc2_L1_Number_Entered = 770,
Coherence_Cache_Proc2_L1_Number_Returned = 770,
Coherence_Cache_Proc2_L1_Snoop_Invalidates = 147,
Coherence_Cache_Proc2_L1_Snoop_Requests = 259,
Coherence_Cache_Proc2_L1_Snooped_Count_per_Device = {},
Coherence_Cache_Proc2_L1_Snooped_Devices = {},
Coherence_Cache_Proc2_L1_Utilization = 1.7432000348639

High Bandwidth Memory

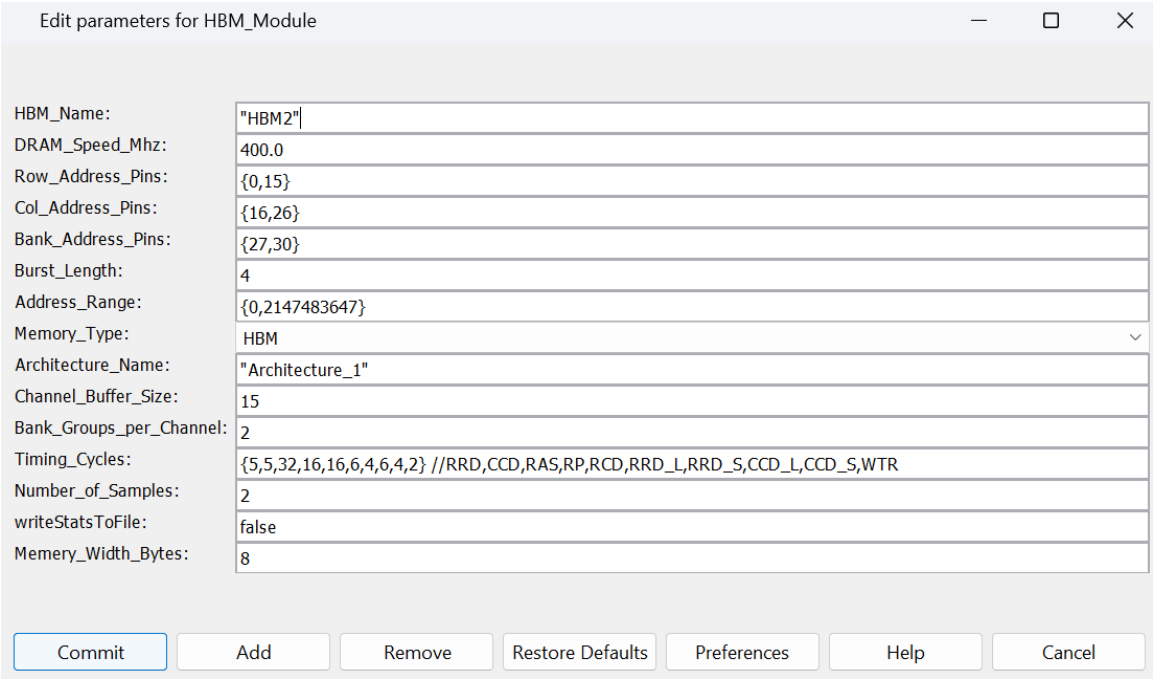
Block Description:

HBM block is a 8 channel DRAM architecture and it is implemented with stochastic RAM block and simple controller logic.



Ports:
Channel_0_IN, Channel_0_OUT……. Channel_7_IN, Channel_7_OUT: DRAM connections with SoC devices.
Statistics: stats port

Parameters:



Architecture_Name
Unique architecture name, typically “Architecture_1”
Eg: "Architecture_1"

HBM_Name
Unique name for the CycleAccurateDRAM
Eg: "SDRAM"

DRAM_Speed_Mhz
Speed of CycleAccurateDRAM in Mhz
Eg: 400.0 /* LPDDR2 typical shown */

Row_Address_Pins
Defines the set or group of bits for Row addressing
Eg: {0,15}

Column_Address_Pins
Defines the set or group of bits for Column addressing
Eg: {16,26}

Bank_Address_Pins
Defines the set or group of bits for Bank addressing
Eg: {27,30}

Burst_Length
Burst Length or BL.
Eg: 4

Address_Range

Defines the total range of address for this HBM block
Eg: {0,2147483647}

Memory_Type
Defines the standards of HBM
Eg: HBM or HBM2

Channel_Buffer_Size
Buffer length for each indpended channels
Eg: 15

Bank_Groups_per_Channel
Number of bank groups for each channel Total banks will be equally divided as set of banks
Eg: 2

Timing_Cycles
Number of cycles for the memory controller operations such as RRD, CCD, RAS, RP, RCD, RRD_L, RRD_S, CCD_L, CCD_S, WTR
Eg: {5,5,32,16,16,6,4,6,4,2}

Number_of_Samples
Number of Statistics samples
Eg: 2

writeStatsToFile
Alternate option to analyse the statistics, true will enable the block to write the statistics in a text file.(local ddirectory to the model)
Eg: false

Memory_Width_Bytes
Width of the DRAM channel.

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

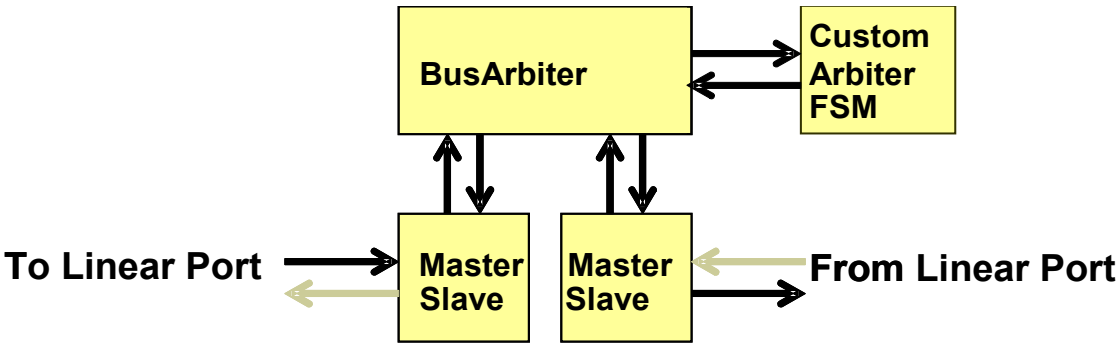


Figure 58 Bus Arbiter Flow Diagram

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 Bus Interface 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

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 pulldown. 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.

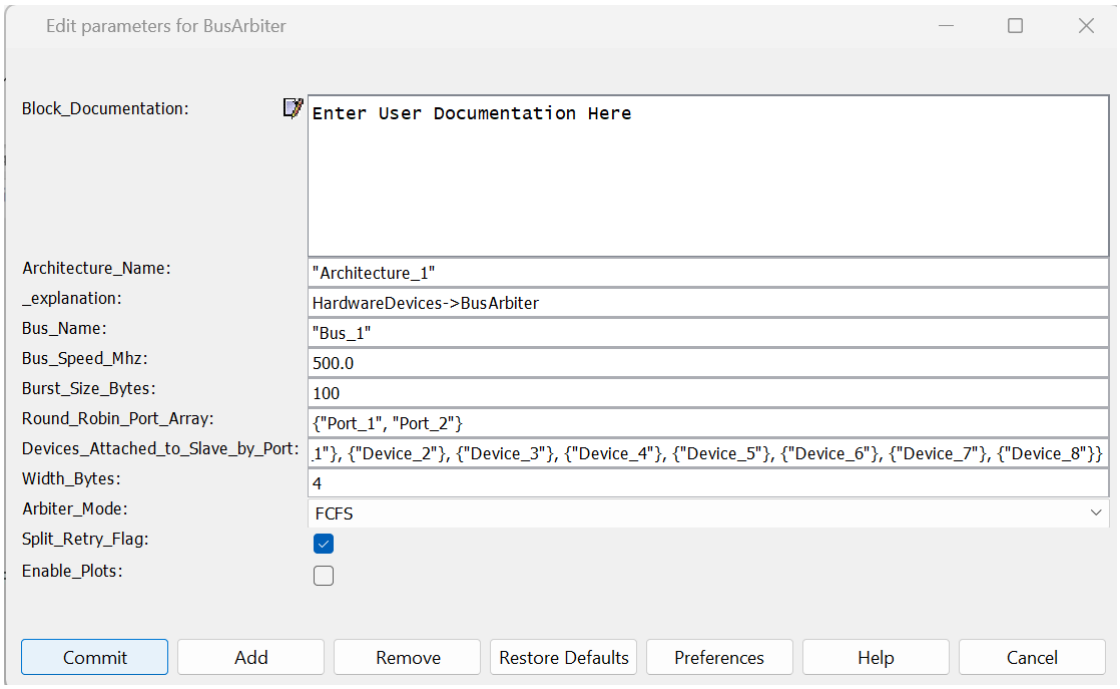


Figure 59 BusArbiter Configuration Window

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

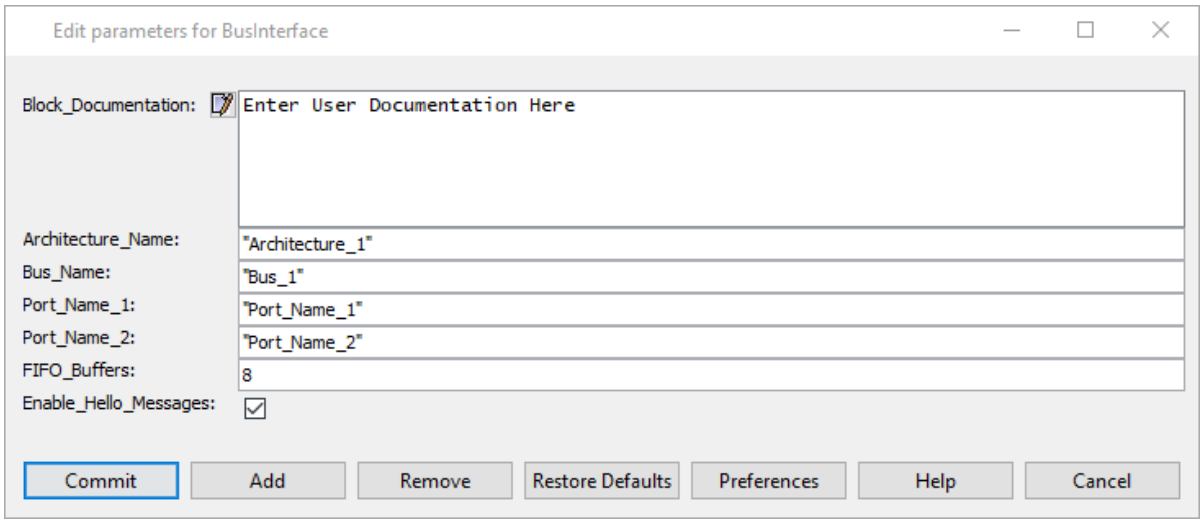


Figure 60 BusInterface Configuration Window

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

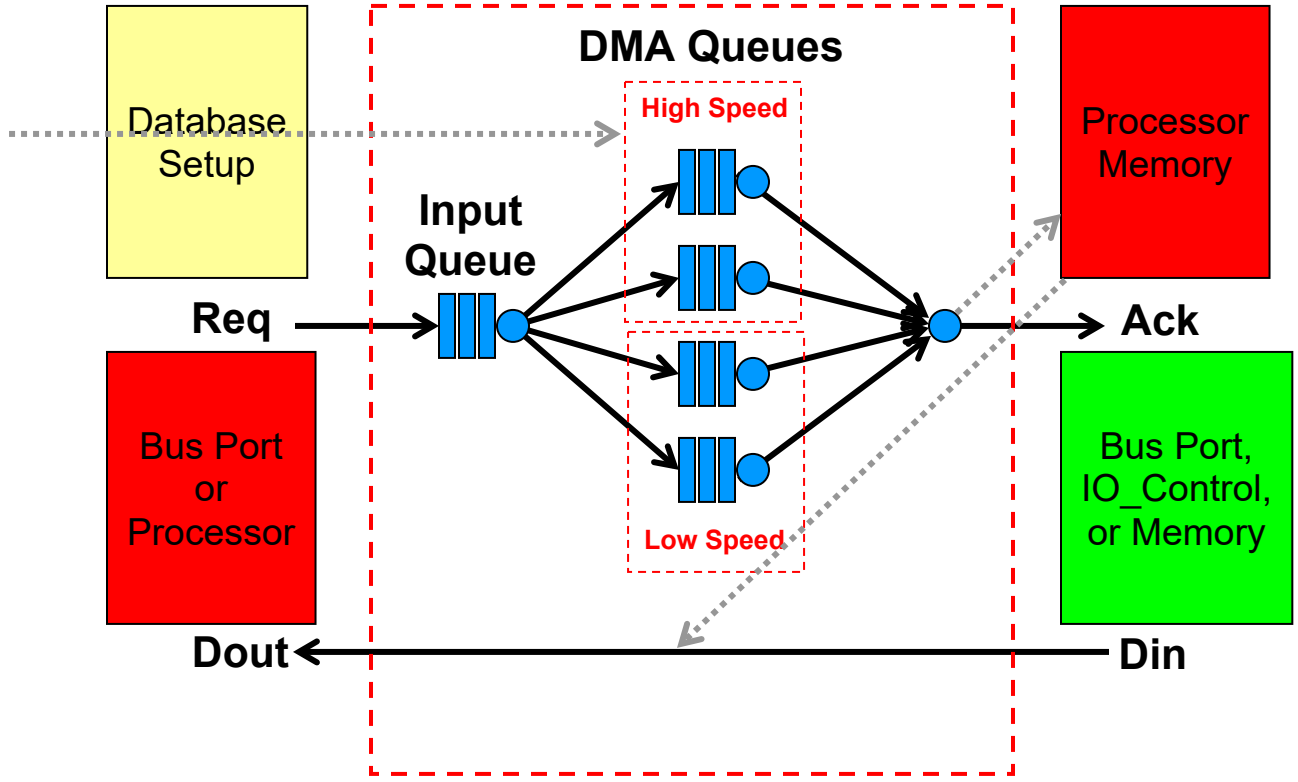


Figure 61 DMA Flow Diagram

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.

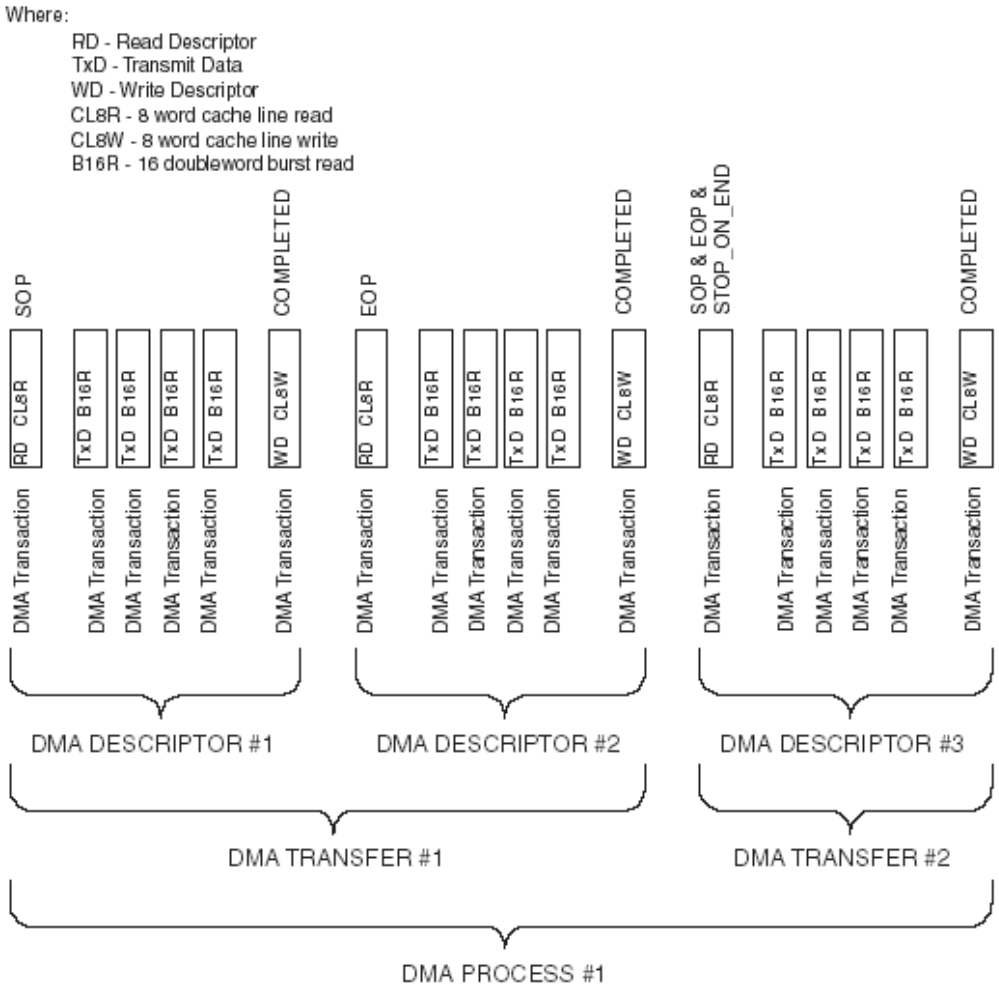


Figure 62 CDMAC Illustration of Tx Engine Flow

The DMA can also be triggered by the input data structure fields instead of configuring the data base for each task. To enable this the Memory_Databas_Reference parameter in DMA block must be set to “None”, then the input data structure must contain the following fields with required format.

Fields	Example value format	Explanation
A_Task_Name	"Activity1"	Name of the task
DMA_Command	{"Read","Write"}	Sequence of commands to be performed
DMA_Destination	{"RAM","Display"}	Corresponding destination for the commands listed
Priority	irand(1,10)	Priority of the task
A_DMA_Bytes	{1024,1024}	Corresponding data size of for the mentioned commands
A_DMA_Channel	{1,1}	Channel allocation for each command
A_DMA_Burst_Bytes	{64,64}	Maximum burst size for each command

Example format in Expression list:

input.A_Task_Name = "Activity1"
input.A_DMA_Command = {"Read","Write"}
input.A_DMA_Destination = {"RAM","Display"}
input.A_Priority = irand(1,10)
input.A_DMA_Bytes = {1024,1024}
input.A_DMA_Channel = {1,1}
input.A_DMA_Burst_Bytes = {64,64}

Routing Functionality

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

Example of DMA in a Model

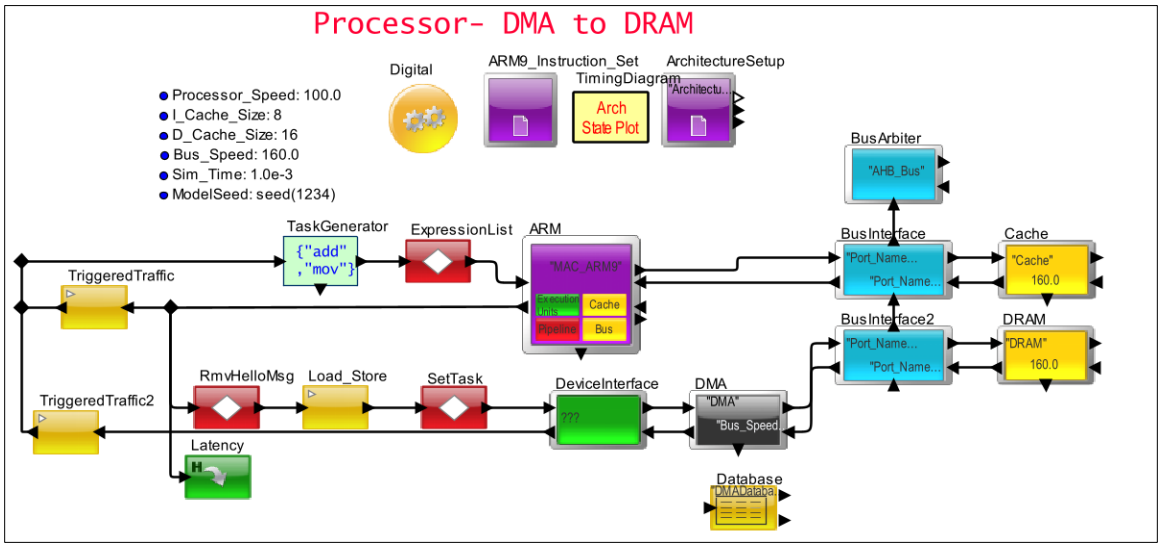


Figure 63 Using DMA in a Model

Statistics

IO_per_sec: Input and output transactions per second.

Parameters

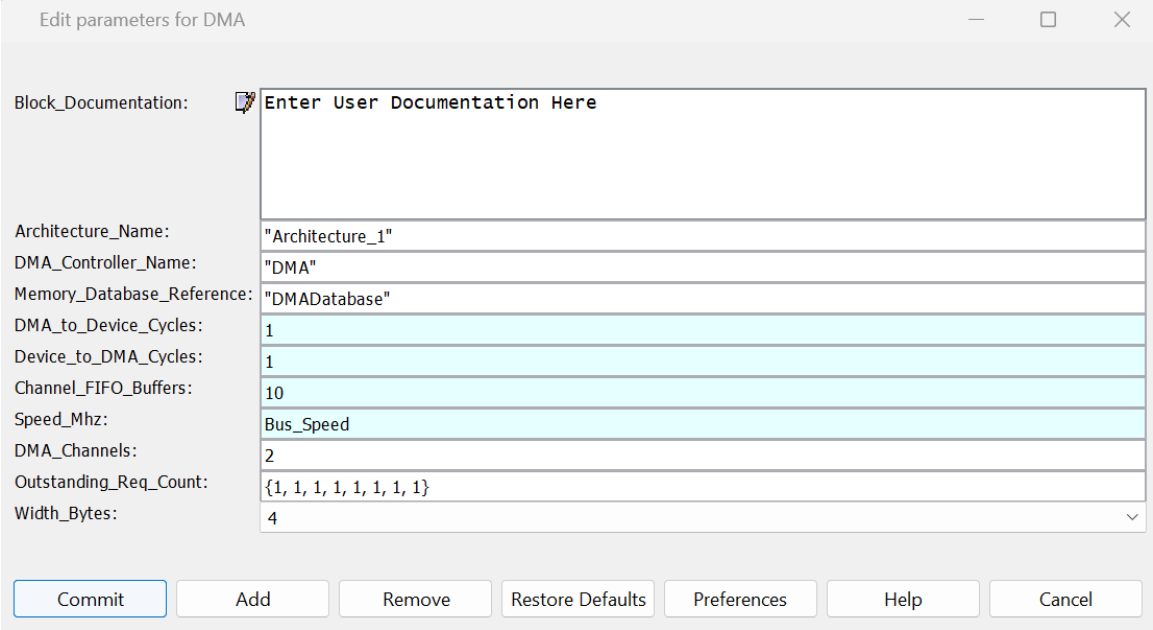


Figure 64 DMA_Controller Configuration Window

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.

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 / Speed in Mhz.

DMA_Channels
The number of channels in the DMA

Outstanding_Req_Count
The number of outstanding requests per channel.

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.

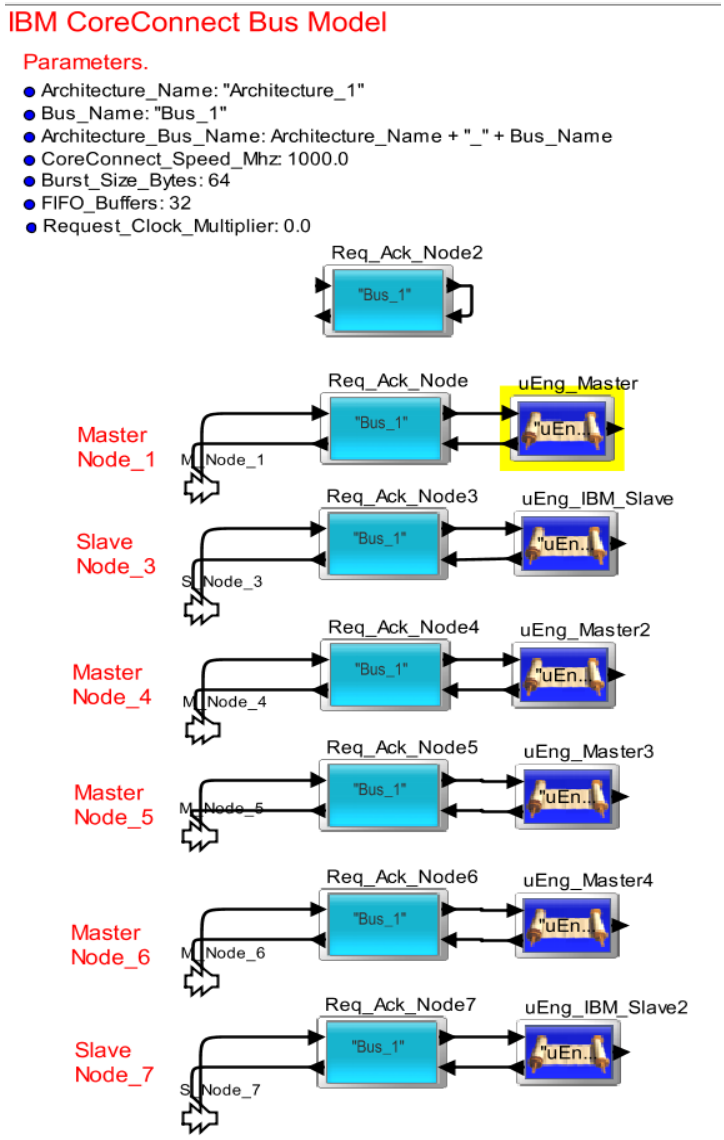


Figure 65 Master, Slave and Controller Block Connection

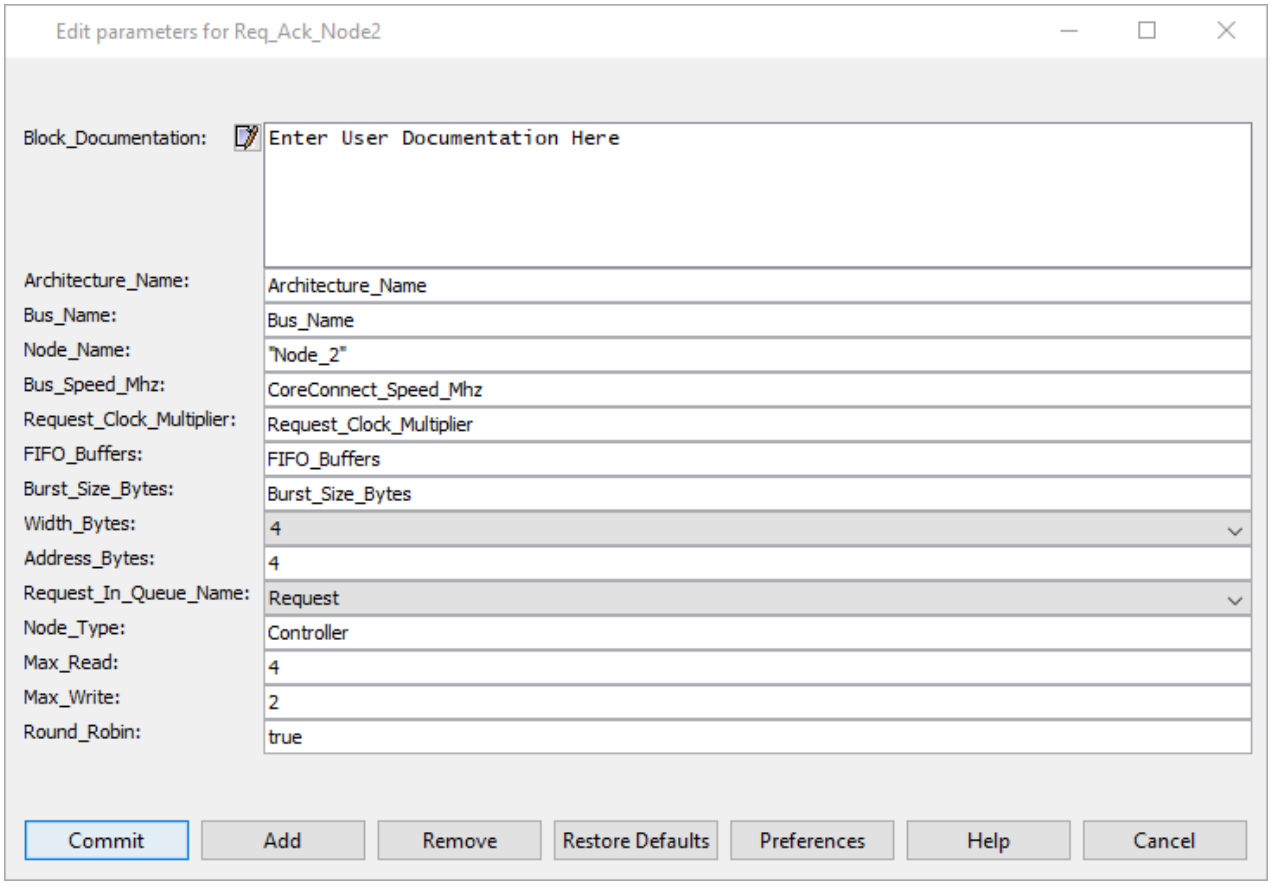


Figure 66 Req_Ack Node View

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

	Size on Controller Node, user must add this parameter to make it valid.
▪ Max_Write	User added for maximum queue Size on Controller Node, user must add this parameter to make it valid.
▪ Round_Robin	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:

Req_Ack_Node Bus Commands	A_Command field
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
A_Bytes	int	64
A_Command	string	"Read_Memory"
A_First_Word	boolean	true
A_Priority	int	1
A_Source	string	"IO_1"
A_Destination	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:

Master Commands	Action
"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:

Controller Commands	Action
"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:

Slave Commands	Action
"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_MBs*** – 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 = 0.3333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
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,

Processing Block

1. **Defining Flows blocks**: These are programming blocks used in defining expressions, evaluating, assignments, and making control decisions. The types of operations includes:

   ⇒ Mathematical or Logical Statement
   ⇒ if-else
   ⇒ while
   ⇒ switch-case
   ⇒ Field selection

   The Data flow blocks are used to evaluate expressions, assign values to a Field or variable, randomize Field values, calculate processing cycles on a scheduler blocks, compute statistics such as latency (TNOW - TIME), utilization and throughput and create assumption values. Control Flow blocks are used to make decisions, create loops, and take branches.

2. **Using Database**: The Database block is a high-performance lookup utility and resembles a relational database. Create a lookup table using this block and link it with other database blocks in the model. The table can be read by one Database block when it is written by another block. The lookup table can have up to 4000 entries and each entry can contain multiple columns of data. This example VS_AR\doc\Training_Material\General\Lookup shows the usage of lookup, write and remove operation.

Virtual Connections: The IN, OUT, MUX, and DEMUX blocks constitute a set of powerful blocks that are intuitive to use and do not require any routing information, except the user entry of the proper source and destination names. In the simplest form, the IN, OUT Blocks are similar to references in a CAD drawing linking to a specific point in the model. If an IN and OUT Block have the same reference name, then they are linked directly in the Graph Editor space. The biggest concern of user is that named references are unique; else certain connections are inadvertently linked to others in Graph Editor. These blocks can be used for model routing, making decisions, and triggering events in the model. Use RegEx function called readAllVirtual to get list of virtual connections in model.

Script Block

This block is used to write scripts and a set of blocks that use a C-like programming format to describe the logic or behaviors. The three supporting blocks are Script block, and Virtual_Machine_Untimed. The block can be used anywhere that detailed programming would be more suitable than using multiple blocks. This block can contain all the RegEx functions and contains standard C functions such as if, else-if, else, while, and SWITCH/CASE. RegEx algorithms can be created to perform decoding, loops, blocking waits, non-blocking waits, or execution based on input transactions, or model level variable. Input or Output ports can be added for additional functionality or virtual transactions can be sent, without ports or wires, into or out of the Script block. Event or time-based queues can be declared inside the Script.

The Script block accepts data structures on the input ports, processes a sequence of C-like code, and outputs a data structure. The data structure can arrive at the ports or via virtual connections to the block. All the data structures are placed in a single non-priority input queue in the order of arrival. The data structures are executed in the order of arrival. The executions are requests using SEND to LABEL within the block and events being sent to this block. In the code, the currently executing data structure is named “port_token”. The arriving data structure can be associated with an arriving port, SystemResource (return from SystemResource) or “virtual” (virtual connection), and “event” (user event).

The sequence of code can utilize the incoming data structures field, variable (local, global and variable-defined inside this block), RegEx language and parameter values. Using the RegEx functions, this block can send the Data Structures to Virtual Connection, SystemResources and other Script blocks.

To define this code, drag the Script or Virtual_Machine_Untimed block to the Editor. To access the code, right-click the block and select "**Open Block**". The following are the main differences between the three blocks:

Feature	Script Block	Virtual_Machine_ Untimed
Time and Events	Yes	No
Queue	Must be defined	Must be defined
TIMEQ and Queue	Yes	Only Queue
Plotting	Yes	Yes
Pop port	No	No

When the code is entered, make sure you save in the Text Editor and the Block Diagram Editor for your changes to be visible. When simulation is started, the block compiles the code and translates it into machine language. Use Listen to Block to view the compile sequence, to identify errors, and to check execution correctness. The translated code may look different in certain places. For example, the line before LABEL: BEGIN

contains a GTO(END) including that all the initialization code is executed without executing the operational code. Other changes include a JIF for if, if-else and else statements; GTO(LABEL) for SWITCH operators and LABEL for CASE.

By default, the block starts executing the code that is placed before the LABEL: BEGIN, and the logic code written after the LABEL: BEGIN will be executed when there is an input at any one of the input ports.

The code execute in sequence. The code can contain actions, logical operations, and delays. The delays can wait on time or on a required event. To learn more about events, check the section on events in this document. A WAIT in the code creates a blocking operation and prevents any further execution of any data structure until the time/event has elapsed.

Additional ports on the input/output can be added. The structures arriving at the input ports are accumulated in the input queue and executed in the order of arrival. Multiple input ports are useful for a visual view of the connections and for cases where the arrival source must be identified for code execution. Values on all ports do not have to arrive for the code to execute. It takes all the inputs and places them in the input queue, in the order of arrival. The SEND function sends the required values to the output ports, Labels in this code, other Script or virtual connections. SEND also supports a delayed output.

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. It covers task-based power modeling, transitions, and management logic across hardware, software, and network components. The user can setup PowerTable to add power management into the model get the power consumption reports and graphs. The PowerTable library consists of the Power_Table, Battery, Power Harvesters, Temperature and RegEx functions for making state changes and accessing the statistics content of the table.

Multiple Power_Tables can be maintained in a single model. Any number of devices can be associated with a single PowerTable block. Each standard device can have any number of states, single transition time between states that is used for all states and a reference to the state that is currently associated with Active and Standby. VisualSim libraries blocks support Active and Standby. The Processor has Wait and RAM block uses Wait for Refresh. The **VisualSim** PowerTable is based on dynamic system operation and enables 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 State Transition delay. This value may be different for different power states. It is possible for a designer to assign delay for individual states.

Let us take an example. When the Processor is in standby state, the PowerTable maintains the current power consumed by this processor to be value in the standby column for this processor. When processor moves to active to busy state, the PowerTable delays by that amount as mentioned in t_OnOff. 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 to update the state change or a new power level.

8 Power Table 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	Existing State	OffState	OnState	t_OnOff	Mhz	Volts	Domain
Architecture_Name + “_” + Block_Name	Double	Double	Double	Double	Double	Double	Double	Double	Double	Double	Integer
Scheduler + “_” + Block_Name	Double	Double	Double	Double	Double	Double	Double	Double	Double	Double	Integer

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. For example, the processor maintains 4 states while the cache has only two states. t_OnOff determines the transition time between one state to another, zero is a legal number. 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.

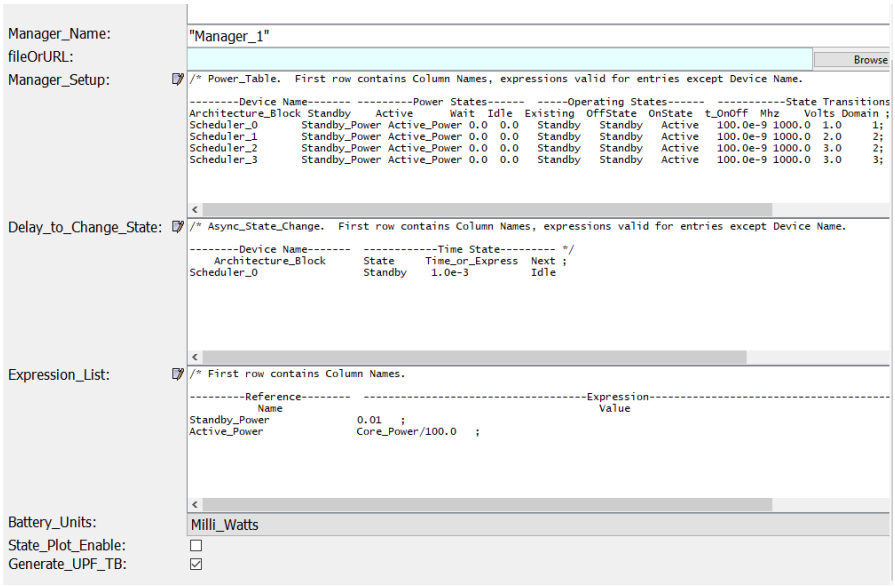
8.1 How it works

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, it might cause an error in the device.
- Power Change RegEx function allows one to modify the power dynamically during simulation execution to reflect voltage changes on top of state information.
- powerManager() RegEx functions can obtain all power information from Power table.
- Instantaneous power is updated at the end of the Transition Cycles.
- Active State is minimal entry for any referenced block, others optional.

8.2 Block Level Parameters



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

Part 1: Manager_Setup

1. Architecture_Block Column will have the Resource Blocks model (check Documentation for Resource specific syntax)
2. The Power Values for each state can be entered in values (taken from documentation of the specific IP) or they can be expressions defined in the Expression_List part of the PowerTable
3. The powerTable supports any number of states for each of the Architecture_Block
4. The Block will be in OffState (default: Standby) when there is no transaction. When there is a transaction, it will transition to OnState (default: Active). The delay between the transition will be t_OnOff.
5. **Mhz, Volts, Domain** are optional columns for the user. They can be used if the user wants to add Power Management algorithms.
   For Example: Observe the impact of changes in voltage, frequency on total power consumption, partition blocks into domains and observe its impact on total power.

Part 2: Delay_to_Change_State

1. The user can enter the name of the Block and the transition from "State" to "Next" will happen if the Block happens to be in the same "State" for a duration specified in "Time_or_Express".

Part 3: Expression_List

1. The user can define variables and it's value or an expression in this part of the table. These variables can be used in Manager_Setup.

Part 4: Options

1. Battery_Units : Unit for Power measurement (Watts/Milliwatts)
2. State_Plot_Enable : if this option is enabled along with writeStatsToFile at Digital Simulator, plot file (.plt) will be generated at the end of simulation which show the transition between different states when viewed using gnuplot.
3. Generate_UPF_TB : if this option is enabled, at the end of simulation 3 additional files will be generated in the <model_name>_results folder (<model_name>.upf , pmu.sv, testbench.sv)

8.3 Block Ports

The Power Manager has three output ports:

1. instantaneous power consumption.
2. Average Power consumption.
3. State change happening in each device that are listed in the power table

8.4 Block Methods

Eigth 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, uProcessor, "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 return the power value of the updating state.

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 readMemory(String Battery_Name+"Pcur")

Eg: Result_A = readMemory("Battery_1Pcur")

Description: This RegEx functions gets the current battery charge during the simulation. The Battery_Name must match name in the Battery 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 6:

Double stateChange(String Manager_Name, String Device name, String Operating State, String New State Name)

Eg: Result_A = stateChange("Manager_1","Dev_A","Existing","Standby")
Description: This function only used to update the state of a device during suimulation from Existing, OnState and OffState, for others it will throw an error. So the Operating State must be in one of these 3 state, and the new state can be anything.

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 = 4.60358700000002E-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}*

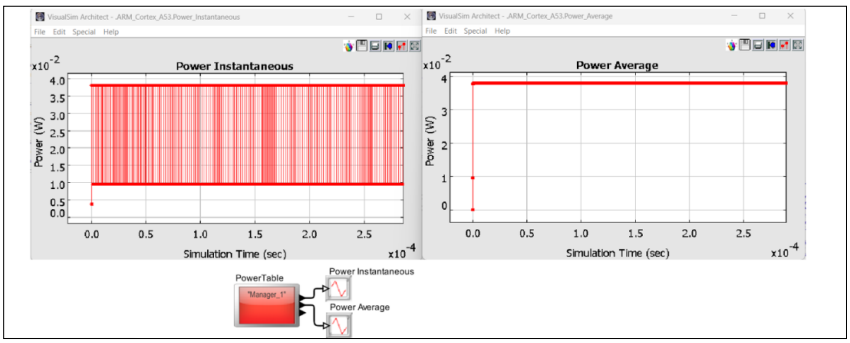
Function 8:

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. This can also be used in custom block with multiple instances, where the power each instance is maintained in a single entry. For example the wire power consumption is different in each wire, we can use a single entry for all the wire to observe the total power consumption in wires due to data transfere. This function returns the new power value in Watts.

8.5 Power Table Outputs:

1. **Instantaneous and Average Power Plots**



2. **Cumulative and Average Power Report for each Device and for each level of hierarchy**

```
Power_Manager  (Multi-Core_DVFS_Design.Manager_1)
Hier File Name (PowerStats_Multi-Core_DVFS_Design_Manager_1_Hier.txt)

Device,                                  Cumulative,          Average
Scheduler_0,                             5.16800E-7,          1.61500E-5
Scheduler_1,                             6.32912E-7,          1.97785E-5
Scheduler_2,                             5.96623E-7,          1.86444E-5
Scheduler_3,                             4.32176E-7,          1.35055E-5

TOTAL,                                   2.17851E-6,          6.80785E-5
```

3. **Power Log in the third port of PowerTable:**

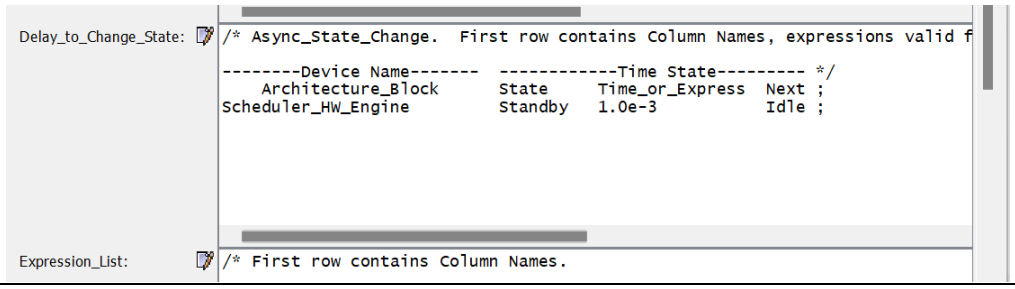
This log can be obtained by Right Click → listen to port on the third output port of the PowerTable. This contains all the transitions for each block and the corresponding power values at each instant. The user can also send this to a TextDisplay and save it if necessary for debugging.

4. **UPF file and System Verilog files**

High level UPF containing Domain and Voltage information Power test bench based on the state transitions of the blocks in the model) are also generated in the model Results Directory. To **enable, select the Generate_UPF_TB option in the power Table.**

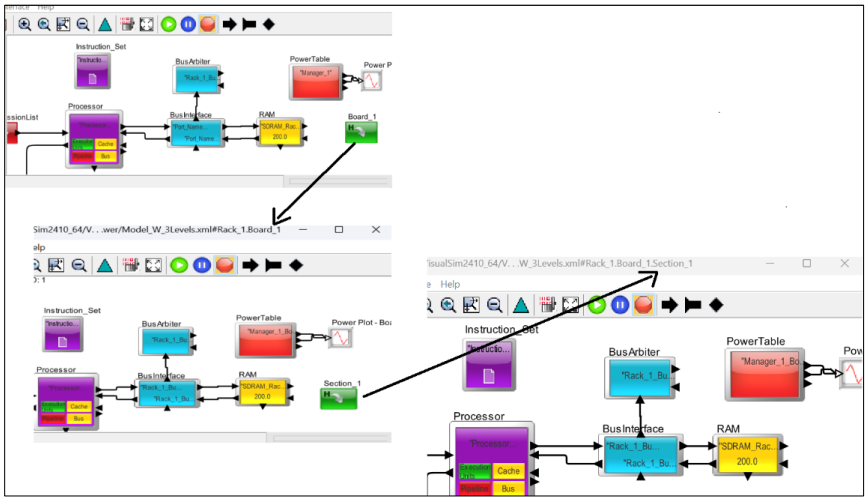
8.6 Power Management

VisualSim Provides an option to add power management algorithms in power table. The user can define the conditions and the state change for each device. For example, the user can set a processing element to move from Standby to Idle if the device is in standby state for 1ms.



8.7 Hierarchical Power Modeling

The Power Modeling can be done in a hierarchical methods, where each power table in the certain window/hierarchy/class file focus on the blocks that are in that level. This helps user to analyse the power consumption of the devices in each level separately.

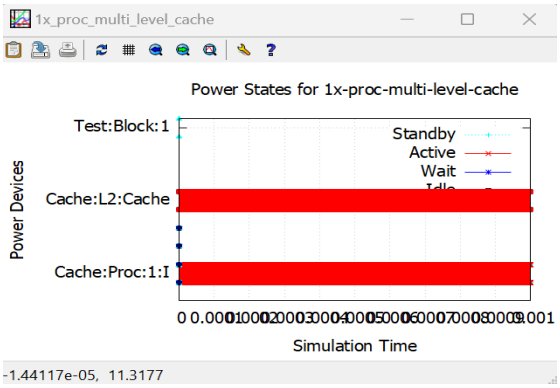


8.8 Power Debugging

In order to understand the Power spikes and unusual consumption patterns, VisualSim helps the user to obtain various power statistics to make understand the design well. The following statistics can be observed using power table.

1. Instantaneous power consumption
2. Average power consumption
3. State change of each device during the simulation
4. Cumulative power consumption of each device
5. State plot of each device

With the help of GNU plot user can observe the state change of each device at specific instance of time.



User can right click on the plot section and drag and drop to view the state changes of all the device in a particular time.

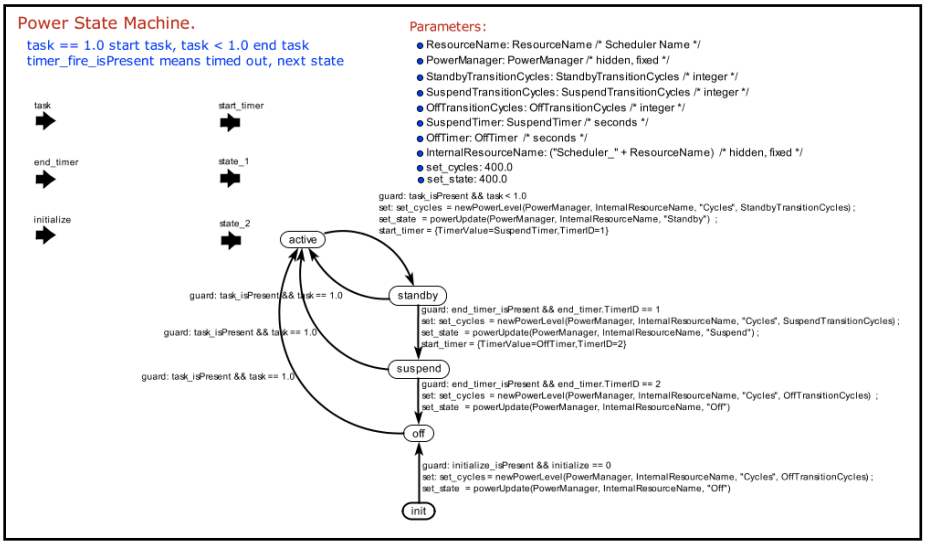
The another way of debugging the unusual activity is through checking the state changes of each device using power table's 3rd port. User can connect a text display to that port and can either view it or save it as a file. By checking the state changes in particular time periods provide an insight of which devices contributes to the most power consumption.

```
OUTPUT AT TIME                  ------ 0.0 ps ------
"      Time,                                  Block,                  Device,        State,        Power"

OUTPUT AT TIME                  ------ 1.0010 ns ------
"1.0010e-09,       1x_proc_multi_level_cache.PowerTable,          Cache_Proc_1_I,       Active,   3.7000e-01"

OUTPUT AT TIME                  ------ 2.0010 ns ------
"2.0010e-09,       1x_proc_multi_level_cache.PowerTable,          Cache_L2_Cache,       Active,   6.0000e-01"
OUTPUT AT TIME                  ------ 3.0010 ns ------
"3.0010e-09,       1x_proc_multi_level_cache.PowerTable,          Cache_Proc_1_I,      Standby,   3.7000e-01"
OUTPUT AT TIME                  ------ 4.0010 ns ------
"4.0010e-09,       1x_proc_multi_level_cache.PowerTable,          Cache_L2_Cache,      Standby,   1.4000e-01"
```

8.9 FSM power Modeling

- 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

Power FSM States and Transitions



8.10 UPF File Generation through Power Table:

Once the Generate_UPF_TB option is enabled in power table the user can get the following files in the result directory in model location

6. UPF file
7. PMU.sv file
8. Testbench.sv

The following explains the each section of the UPF file.

Power domains added based on 'Domain' column in powerTable

Supply nets and ports based on 'Voltage" column in powerTable

Power switch added by default for all domains – swctrl can be used in testbench to enable or disable the switch

Value from "Volts" will be added for Active state of the port. Other values are default

Power state table based on all the supply ports

```
create_power_domain PD_Top -include_scopecreate_power_domain \
-name PD_1_1.0 -elements {"Scheduler_0"}
create_power_domain PD_Top -include_scopecreate_power_domain \
-name PD_2_1.0 -elements {"Scheduler_1","Scheduler_2"}
create_power_domain PD_Top -include_scopecreate_power_domain \
-name PD_3_1.0 -elements {"Scheduler_3"}
create_supply_port -port VDD_1.0 -direction in -domain PD_Top
create_supply_port -port VSS_0.0 -direction in -domain PD_Top
create_supply_net VDD_1.0 -domain PD_Top
create_supply_net VSS_0.0 -domain PD_Top
connect_supply_net VDD_1.0 -ports VDD_1.0
connect_supply_net VSS_0.0 -ports VSS_0.0

#Power switch for switched domain
create_power_switch SW \
-domain PD_1_1.0 \
-output_supply_port {swout VDDsw}\
-input_supply_port {swin Pwr} \
-control_port {swctrl} \
-on_state {SWon swin swctrl} \
-off_state {SWoff !swctrl}

#adding port states
add_port_state VDD_1.0 \
-state {Active=1.0, Standby=0.8, Wait=0.8,IDLE=0.7}

add_port_state VSS_PROC \
-state {Active 0.0}

add_port_state SW/swout
-state {Active 1.0} \
-state {Standby 0.7}

create_pst pwr_state_table \
-supplies {VDD_1.0 SW/swout VSS_0.0}
```

The following gives you the overview of the pmu file generated in VisualSim.

```
module pmu_design (
  input logic Scheduler_0_Change_State,
  input logic [1:0] Scheduler_0_Next_State,
  output logic [1:0] Scheduler_0_Set_State,
  input logic Scheduler_1_Change_State,
  input logic [1:0] Scheduler_1_Next_State,
  output logic [1:0] Scheduler_1_Set_State,
  input logic Scheduler_2_Change_State,
  input logic [1:0] Scheduler_2_Next_State,
  output logic [1:0] Scheduler_2_Set_State,
  input logic Scheduler_3_Change_State,
  input logic [1:0] Scheduler_3_Next_State,
  output logic [1:0] Scheduler_3_Set_State,
  input logic power_clk
);
  // State bit values for Scheduler_0
  localparam Scheduler_0_Standby = 00;
  localparam Scheduler_0_Active = 01;
  localparam Scheduler_0_Wait = 10;
  localparam Scheduler_0_Idle = 11;
  // State bit values for Scheduler_1
  localparam Scheduler_1_Standby = 00;
  localparam Scheduler_1_Active = 01;
  localparam Scheduler_1_Wait = 10;
  localparam Scheduler_1_Idle = 11;
  // State bit values for Scheduler_2
  localparam Scheduler_2_Standby = 00;
  localparam Scheduler_2_Active = 01;
  localparam Scheduler_2_Wait = 10;
  localparam Scheduler_2_Idle = 11;
  // State bit values for Scheduler_3
  localparam Scheduler_3_Standby = 00;
  localparam Scheduler_3_Active = 01;
  localparam Scheduler_3_Wait = 10;
  localparam Scheduler_3_Idle = 11;
  always_comb begin
```

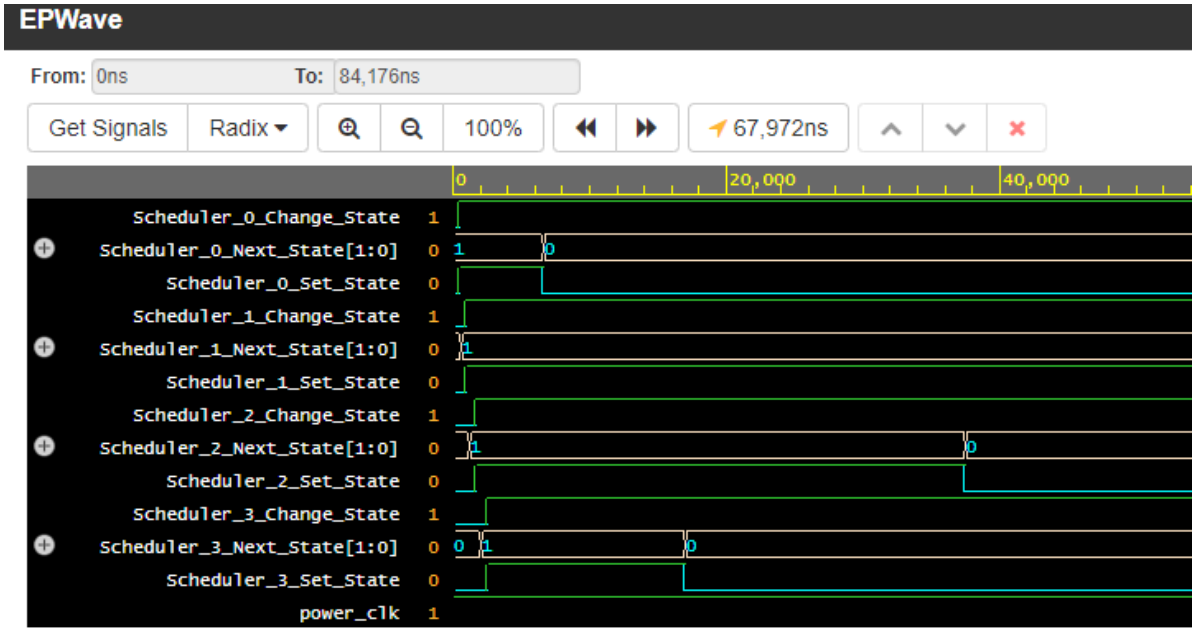
```
  always_comb begin
    if (power_clk && Scheduler_0_Change_State) begin
      Scheduler_0_Set_State = Scheduler_0_Next_State;
    end else begin
      Scheduler_0_Set_State = Scheduler_0_OFF;
    end
  end
  always_comb begin
    if (power_clk && Scheduler_1_Change_State) begin
      Scheduler_1_Set_State = Scheduler_1_Next_State;
    end else begin
      Scheduler_1_Set_State = Scheduler_1_OFF;
    end
  end
  always_comb begin
    if (power_clk && Scheduler_2_Change_State) begin
      Scheduler_2_Set_State = Scheduler_2_Next_State;
    end else begin
      Scheduler_2_Set_State = Scheduler_2_OFF;
    end
  end
  always_comb begin
    if (power_clk && Scheduler_3_Change_State) begin
      Scheduler_3_Set_State = Scheduler_3_Next_State;
    end else begin
      Scheduler_3_Set_State = Scheduler_3_OFF;
    end
  end
endmodule
```

The following image gives you the overview of testbench file generated in VisualSim.

```
    initial begin
     $dumpfile("dump.vcd");
     $dumpvars(0, tb_pmu_design);
     power_clk = 1;
     //forever #50000 power_clk = ~power_clk;
     #104.0; Scheduler_0_Next_State = 2'b01;
     #100.0;
     #104.0; Scheduler_0_Change_State = 1;
     #204.0; Scheduler_1_Next_State = 2'b01;
     #100.0;
     #204.0; Scheduler_1_Change_State = 1;
     #304.0; Scheduler_2_Next_State = 2'b01;
     #100.0;
     #304.0; Scheduler_2_Change_State = 1;
     #404.0; Scheduler_3_Next_State = 2'b01;
     #100.0;
     #404.0; Scheduler_3_Change_State = 1;
     #4104.0; Scheduler_0_Next_State = 2'b00;
     #100.0;
     #4104.0; Scheduler_0_Change_State = 1;
     #6104.0; Scheduler_3_Next_State = 2'b00;
     #100.0;
     #6104.0; Scheduler_3_Change_State = 1;
     #14360.0; Scheduler_2_Next_State = 2'b00;
     #100.0;
     #14360.0; Scheduler_2_Change_State = 1;
     #16104.0; Scheduler_1_Next_State = 2'b00;
     #100.0;
     #16104.0; Scheduler_1_Change_State = 1;
    end
endmodule
```

VCD file having input and output signal transitions for that model can be generated by using pmu.sv and testbench.sv in any of the opensource or commercial EDA tools that support System Verilog.

The user can observe the States that each of the Resources were in throughout the simulation. Signals:



1. Change_State = when it is high, the Power Manager applies the transition from current state to next state for that Block

2. Next_State = This signal holds the binary equivalent of the next state (Example: 00, 01, 10,11 – active, standby, wait, idle)
3. Set_State = This signal will indicate that the transition is complete successfully to the new intended state
The user can observe the delay between transitions by zooming in the waveform.

Note: These files can be useful for downstream integration in the RTL power verification environment.

9 Power Generators

The power generators emulate the time based and motor based power harvesters. In VisualSim there are two library blocks for charging a device such as battery.

1. Time Energy Harvester
2. Motor Energy Harvester

Time Energy Harvester provides three different types of power charge generation methods to trigger direct consumption of batteries.

1. **Constant Power Source**
   This is activated by the 'UseConstant" parameter set to true. This generates a fixed charge at a regular interval of time. The ChargeCapacity is converted in Watts-seconds and sent out through the output port.
2. **File Based**
   This is activated by the 'UseTraceFile" parameter set to true. The fileName field must be empty if this feature is not used. Otherwise a error will be generated. This uses an existing trace file. The trace file requires two fields 'timeStamp' and 'ChargeOutput' which are mandatory fields in the trace file for power generation.
3. **Time Based**
   This is activated by the 'UseTimeBased" parameter set to true. In this case, there is a time window called 'TimeDuration'. During the time window, the setup can define periods and charges for the periods. If the StartWHR is smaller than the EndWHR value, the charge will be computed in a downward slope. The next charge update time will have a lower charge value than the previous point. If the StartWHR is greater than the EndWHR value, the charge will be computed in an upward slope. The next charge update time will have a higher charge value than the previous point. If the StartWHR and EndWHR are the same, then the values will be a flat line. If the StartWHR and EndWHR are zero, then no charge is generated for that period of time. The Time-based mode is best used where the source of energy has different charge over time. Good examples are Solar Arrays and automobile motors.

Motor Energy harvester provide two three different types of power charge generation methods to trigger direct consumption of batteries.

1. **Wind Based**
   Use_Wind_Based parameter enables the Wind based power harvesting when it is set to true. Wind_Turbine_Setup defines a sequence of charge values with different amounts of time for each charge duration. The charge is constant during the time period. The Speed of rotation of the Rotor is specified in the table and the efficiency is specified. The power generated is calulated based on the Speed, Maximum Power Co-efficient, Air Density, rotor Radius and the Constant. All these are specified in the Parameters.
2. **Motor Based**
   Use_Motor_Generator parameter enables the Motor based power generation. The generator setup defines a sequence of charge values with different amounts of time for

each charge duration. The charge is constant during the time period. This mode also has a column for the RPM which can be used in downstream consumption. There is an efficiency value that reduces the usable charge.

The power generators can be directly connected to the top left port of the battery (Battery_Charge_Input).

10 Battery

Battery works in conjunction with PowerTable. It works with or without the power sourcer/generators. Battery module can be used to perform battery sizing, analyze battery characteristics, battery efficiency , instantaneous and average power consumption. Block parameters are provided to configure the battery for different types of battery and configurations.

The battery library has a table containing different type of the batteries. Each type will have different attributes that a regular battery has. These values can be changed to model behavior of a particular type of the battery. The users can create their own battery type by specifying all the required attributes mentioned in the table. A particular battery will then be considered once the Battery type(MyBat) is specified in the Battery_Selection parameter.

Any number of loads can be connected to the battery. Once a load is connected; power requirement of that particular load must be updated in the battery block, this makes that particular load to get power from the battery. Currently available energy harvesters are motor based, wind based, solar, constant power supply and file based. User can select any of the energy harvesters and can modify power output from those blocks.

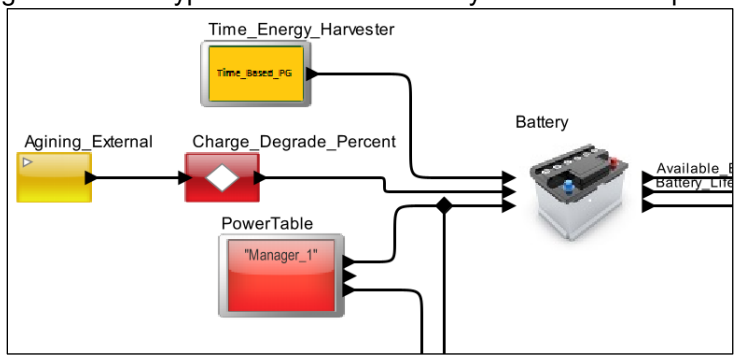
Input Ports:

1. Battery_Charge_Input: Connects to the energy harvester
2. Battery External Aging: Receives double value regarding the charge degradation
3. From PowerTable: Receives instantaneous power consumption.

Ouptut Ports:

1. Available Energy Capacity
2. Battery Life Remaining
3. Stats

The following image shows the typicl connection of battery blocks with the power table and power



11 Temperature

The temperature block calculate the heat and temperature changes with the entire design based on the instantaneous power consumption of each device from the power table. The temperature block accepts the inputs from user such as type of material, ambient temperature, initial temperature and dimensions of the design.
During the simulation the user can observe the change in temperature and heat dissipation for the devices that are configured in the power table.

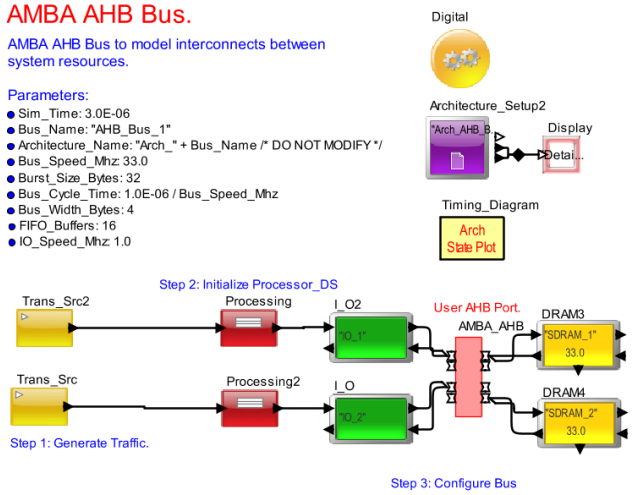
Bus and Interface Standards

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

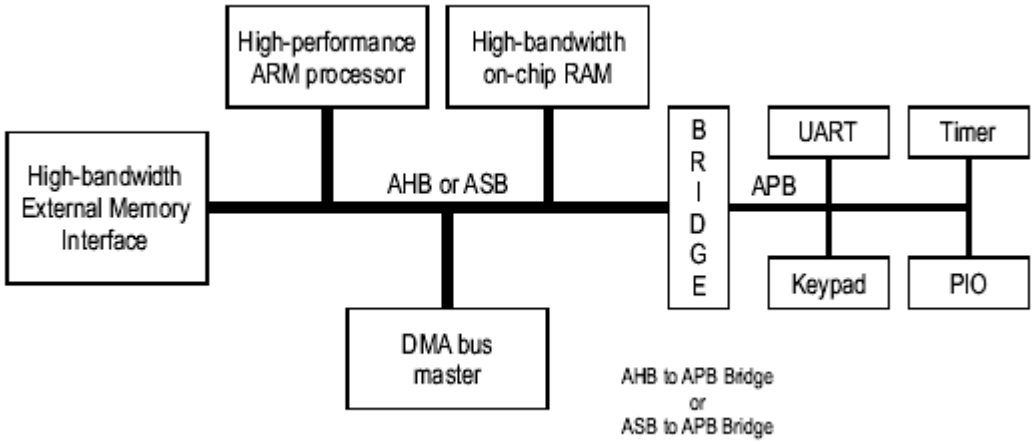
The Standard library blocks like “Traffic” block can be used to generate the necessary Processor_DS. The following Processor_DS fields needs to be initialized to send transactions through the AMBA AHB Bus. This can be achieved with “Processing” or “Expression_List” blocks

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.
Pl 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 Bus Interface 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                  = 1.3333333333333E6,
Bus_1_IOs_per_sec_Mean                 = 1.3333333333333E6,
Bus_1_IOs_per_sec_Min                  = 1.3333333333333E6,
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_MBs_Max   = 176.0,
Bus_1_Throughput_MBs_Mean = 176.0,
Bus_1_Throughput_MBs_Min    = 176.0,
Bus_1_Throughput_MBs_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.

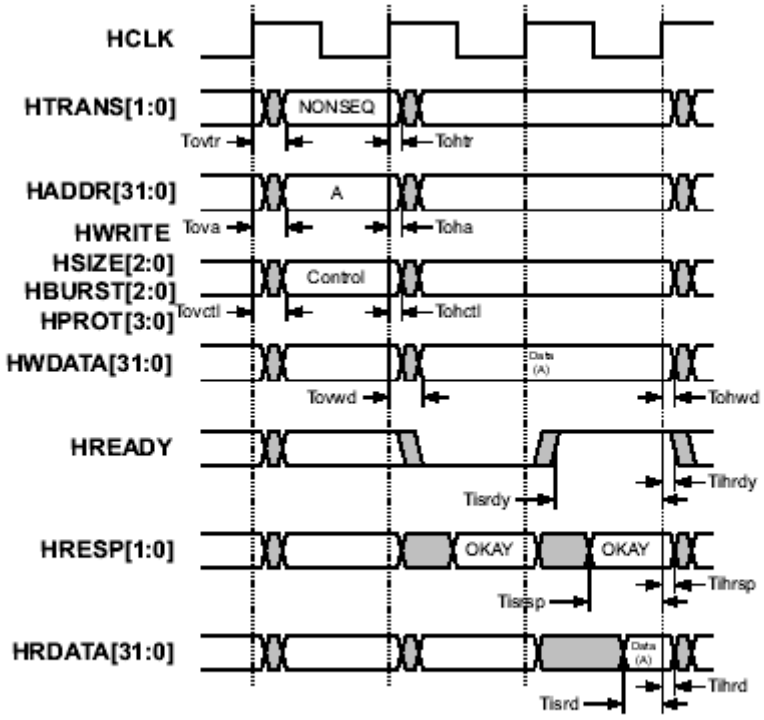


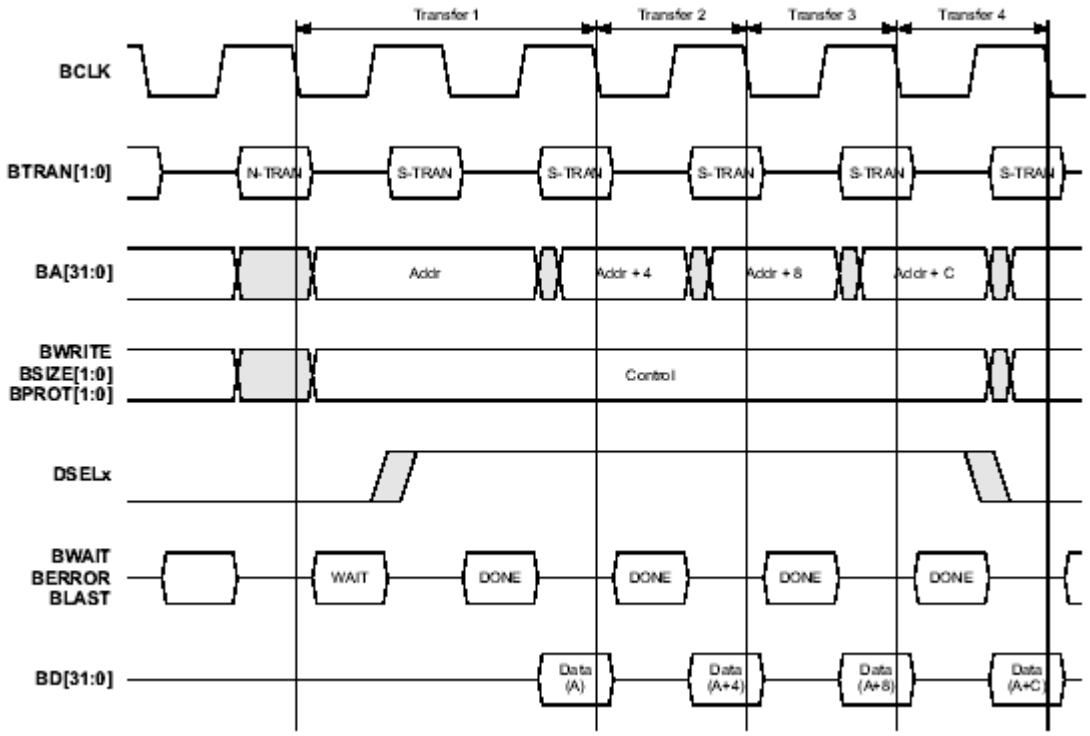
Figure 3-29 AHB master transfer timing parameters

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.

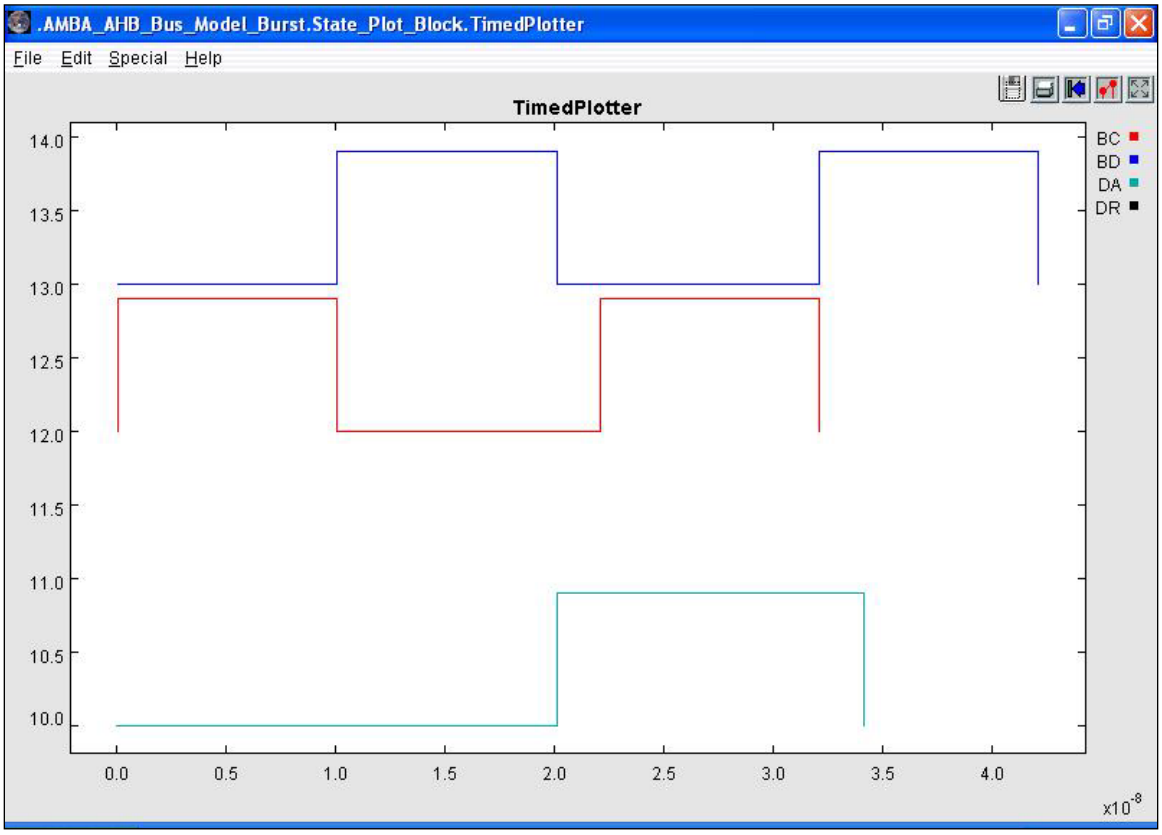


The VisualSim AMBA AHB Bus can be configured to have any of the supported data bus widths (8, 16, 32, 64, 128, 256, 512 or 1024-bits wide). However, it is recommended that a minimum bus width of 32 bits is used and it is expected that a maximum of 256 bits will be adequate for almost all applications.

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.

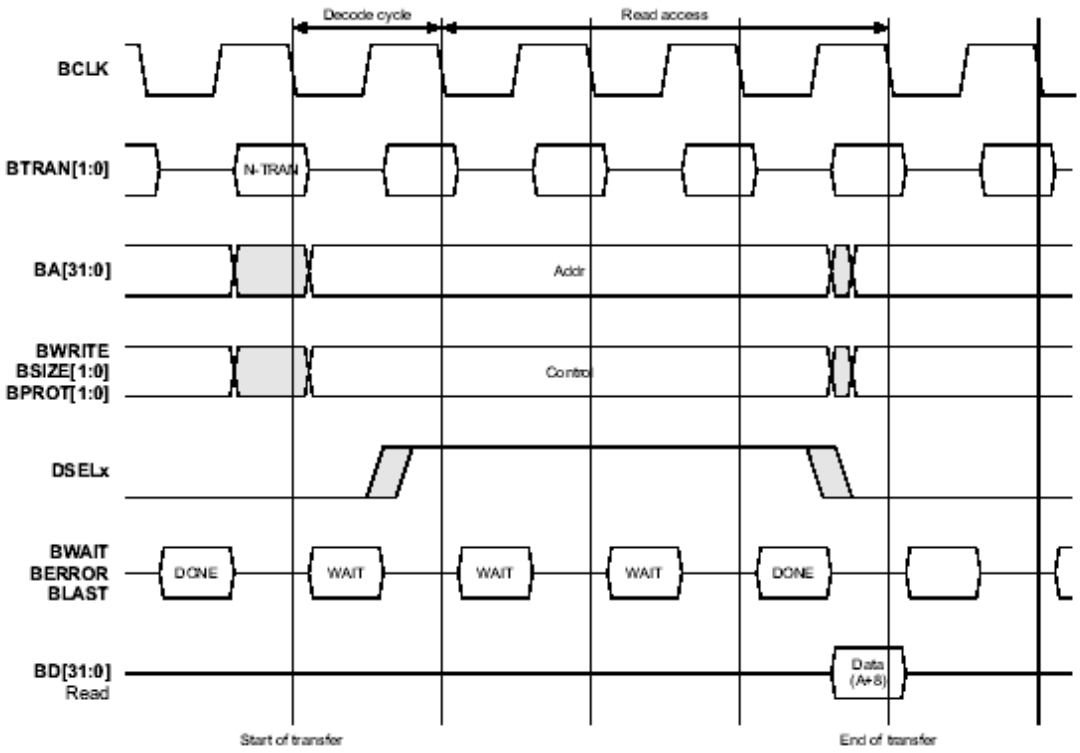
The Figure above shows address/control cycles followed by the data cycles for a Burst Read Operation. The VisualSim Timing diagram for a similar 4 byte Read transfer also depicts the same.

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 bet set to true in custom mode and the same would introduce an additional address/control cycle.

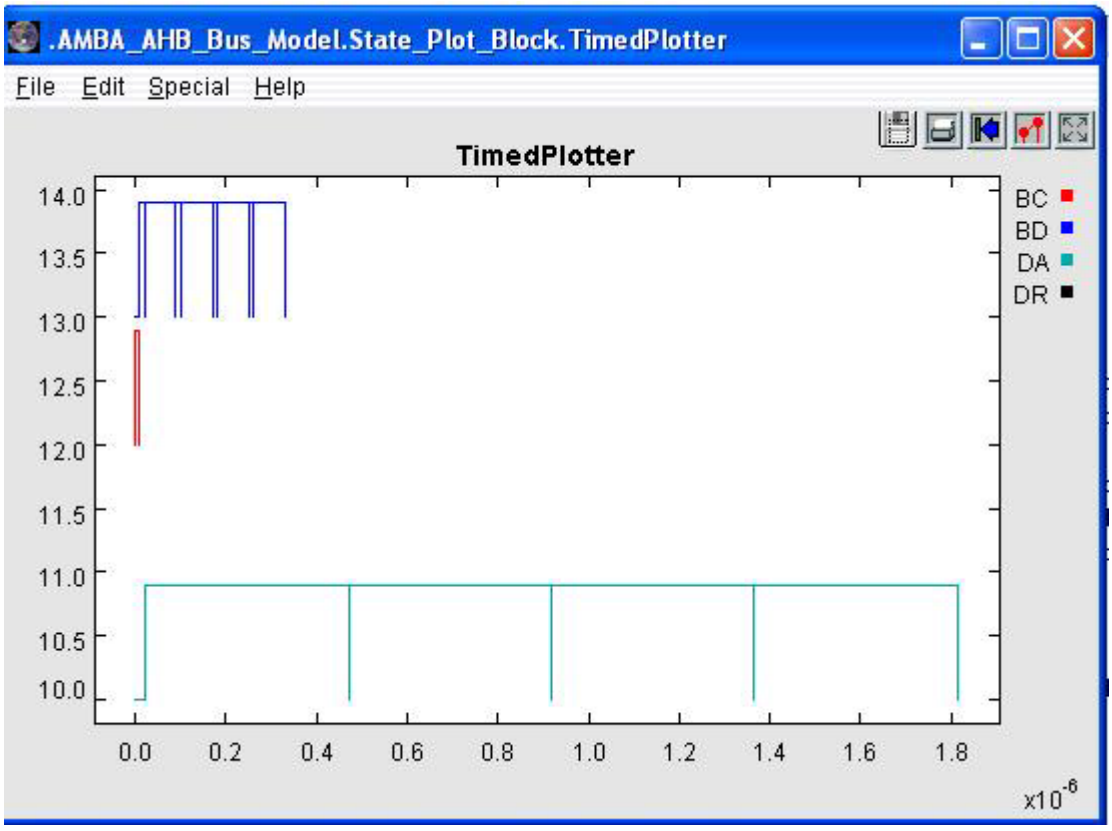
Operational View of the Model

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

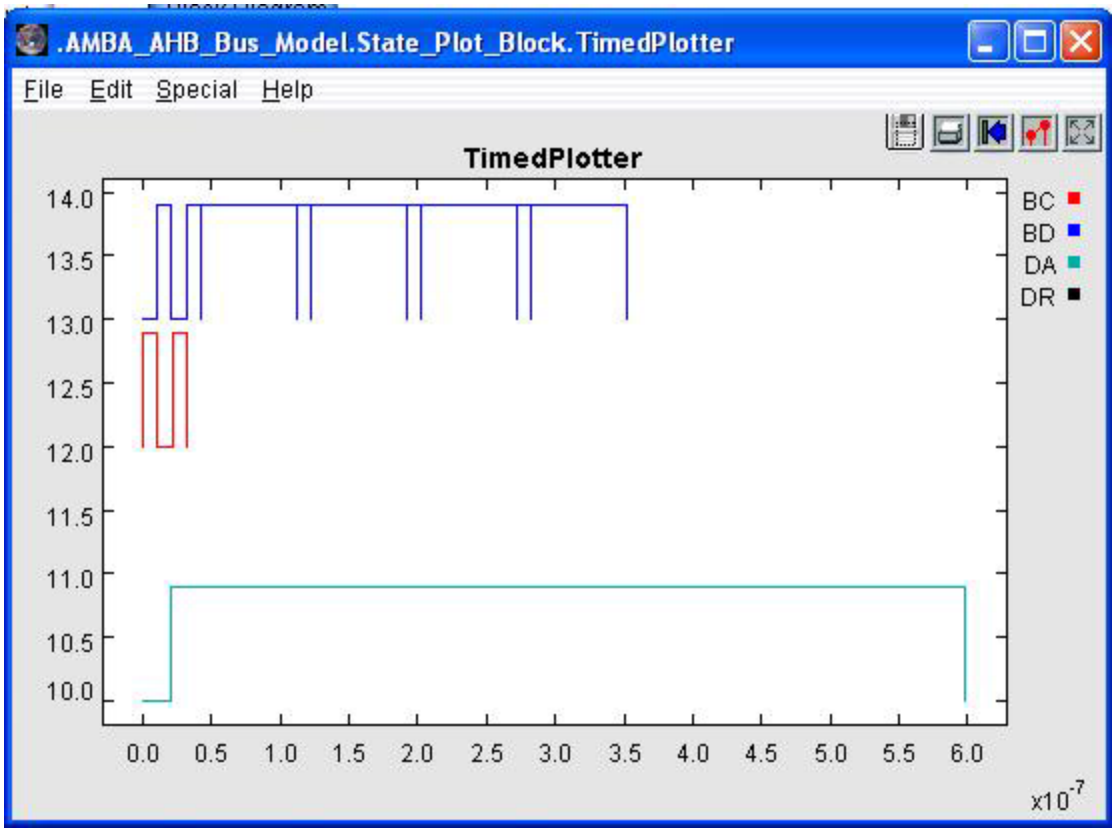
Burst #3
First fragment: {true, 256, 64, 64, Write} //sends first word, 8 bytes
Second fragment: {false, 256, 64, 64, Write} //sends remaining words, 56 bytes

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.

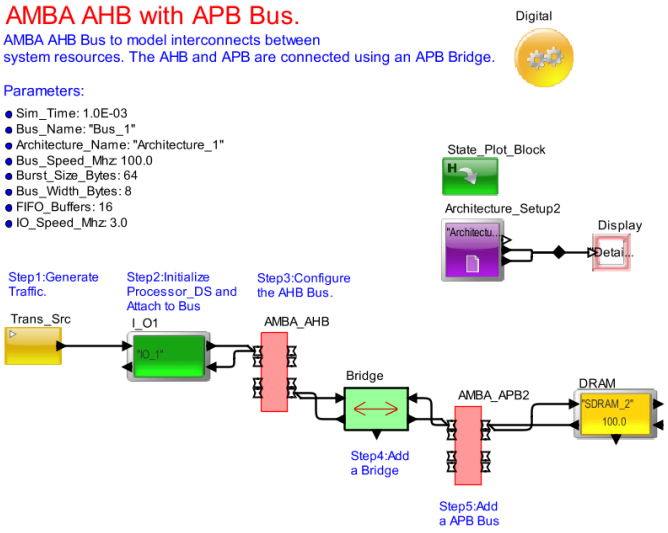
AMBA APB Hierarchical block models the APB Bus.

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

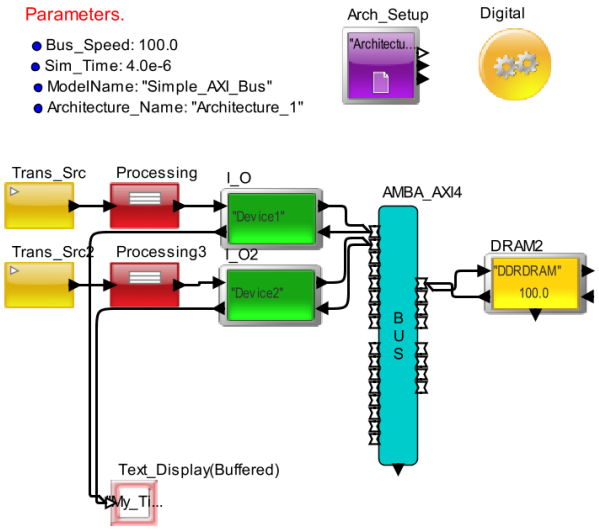


Figure 1-1 AMBA AXI Bus Model

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	{2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}	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
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.
DEBUG	false (Boolean)	If set to true, the lower port outputs the current time, the active transaction and the channel.

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},{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
Slave_First_Word_Flag	true (Boolean)	true returns the first word while a false sends out the last word back to the slave for a Read operation
Ports_to_Plot	{{1,1}} /* {{n,m},{}..} - nth master, mth slave */	Master and slave combination in an array for generating plot data which can be parsed to a gnu plot format usingAXI Parser in VisualSim.
Power_Manager_Name	“Manager_1”	Setting the power table name will enable power analysis in the AXI bus. Setting it to “none” disables the power.
Enable_Plots	False	Enable or disable the plot data generation between master and slave.

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 required and does not need to	Indicate the type of transaction (Request, Response, Address out, Data and ACK)

	exist in the incoming Data Structure	
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”	This field contains the Event name that the sending device is waiting on. 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.

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, input16	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)

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.
Plot_out (disabled)	Output port	Provide plot data based on the configuration

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	2	Number of outstanding Read Requests at the slave interface. Each slave has its own threshold count and this value is common for all the slaves.
Write_Threshold	2	Number of outstanding Write Requests at the slave interface. Each slave has its own threshold count and this value is common for all the slaves.
Master_Request_Threshold	{2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}	Master queue size combining both read and write. 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 Separate Read and Separate Write channel.
The Read_Threshold and Write_Threshold are used for the flow control, 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.

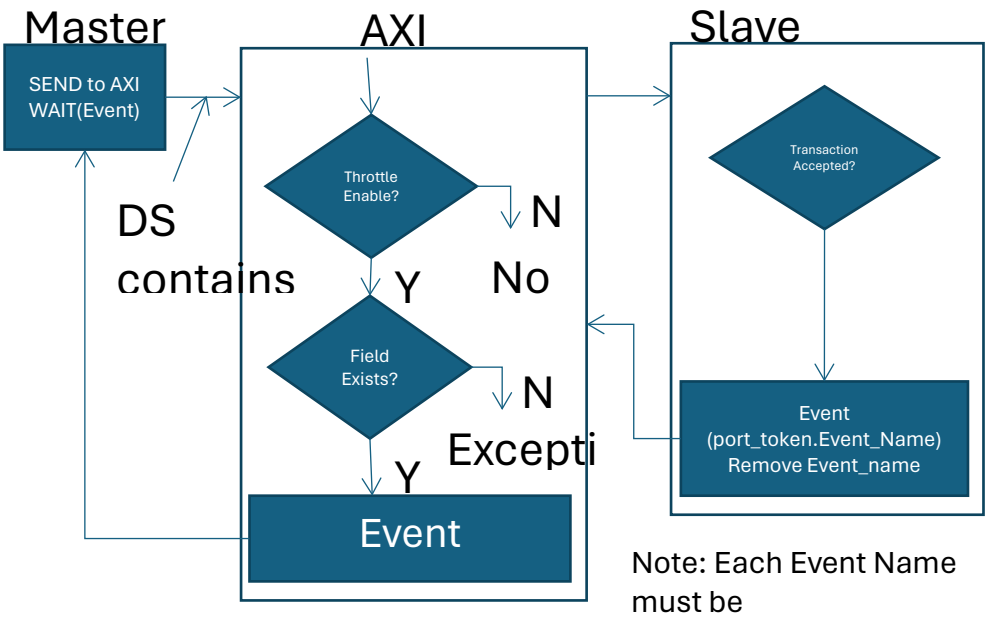
Round-Robin: This performs the round-robin between requests from the Master at every Slave. At the first clock cycle, it starts at Master 1 (input) and then scans each Master (order is based on the input port number). When it finds a request, it sends this out. The next clock cycle, it starts from the 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.

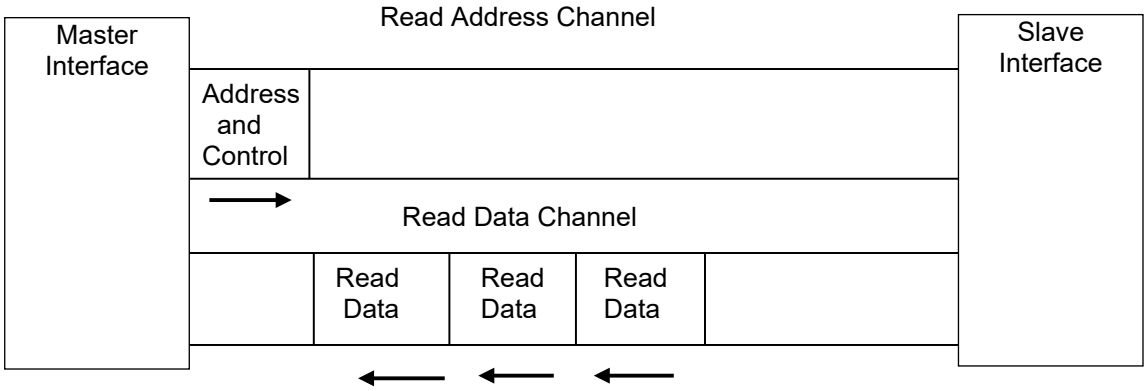


Fig. 1.2 Channel Architecture of Read Comman

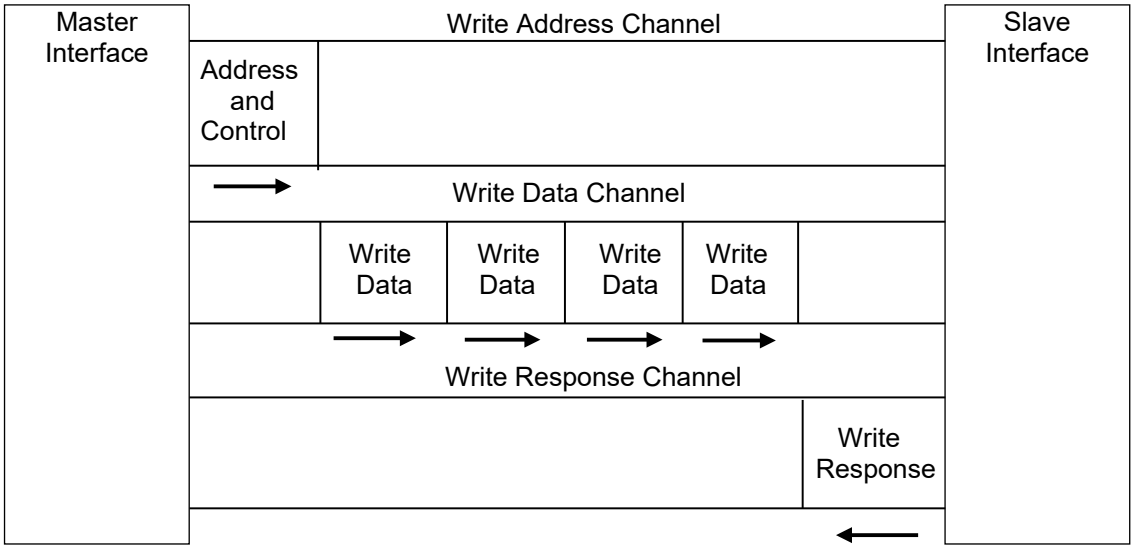


Fig. 1.3 Channel Architecture of Write Command

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.

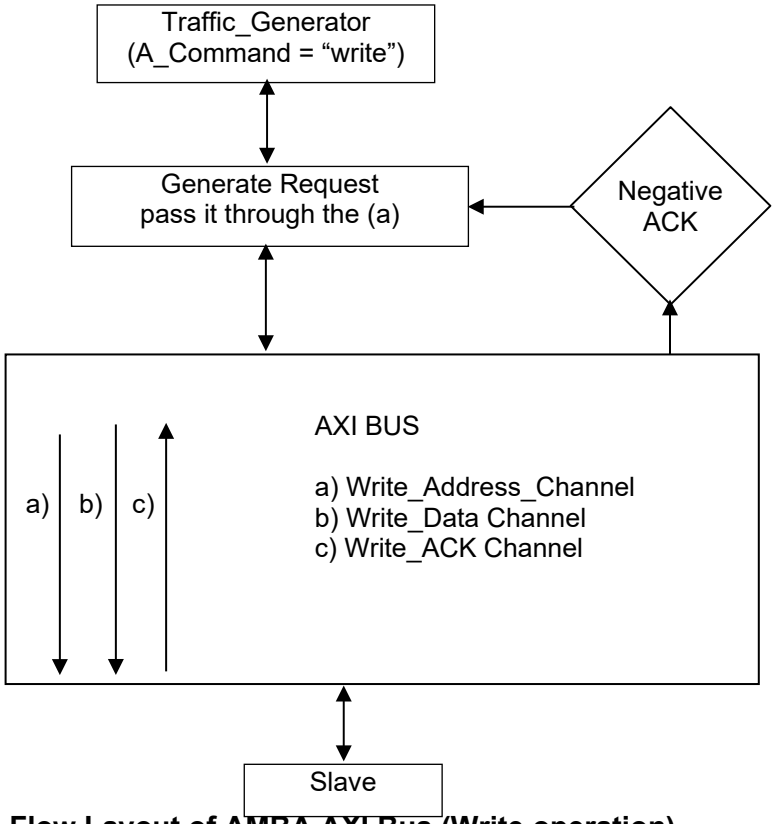


Fig. 1.4 Flow Layout of AMBA AXI Bus (Write operation)

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.

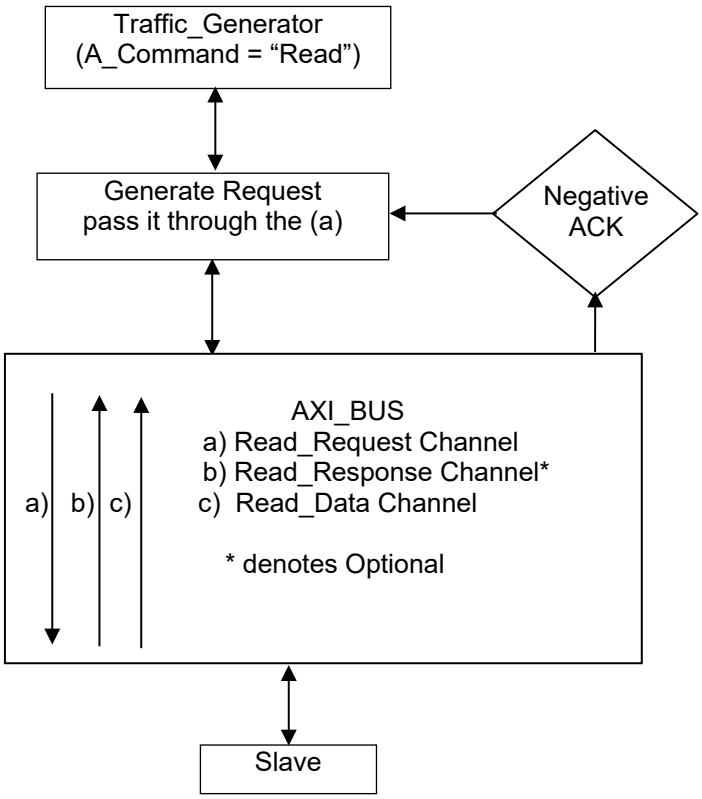


Fig. 1.5 Flow Layout of AMBA AXI Bus (Read operation)

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 if 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

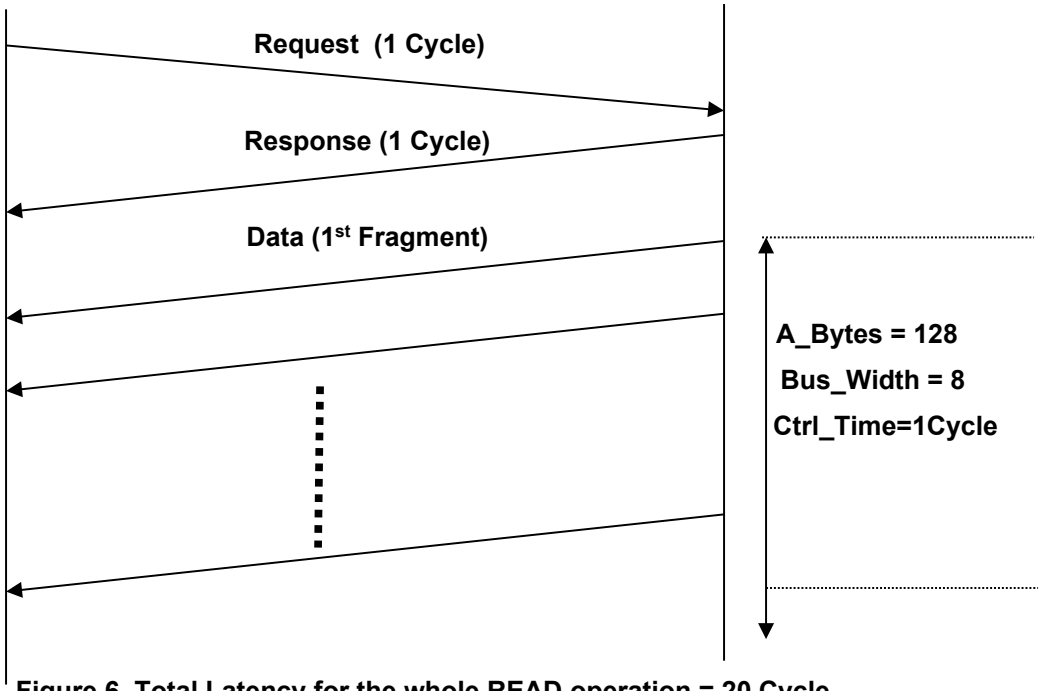


Figure 6 Total Latency for the whole READ operation = 20 Cycle

Note:
Overhead = Fragments * (Slave_Width / Slave_Speed) *(Bus_Speed / Bus_Width)
Total_Data_Cycle =Ctrl_Time + (A_Bytes/Bus_Width) + DRAM_Latency

Write Command

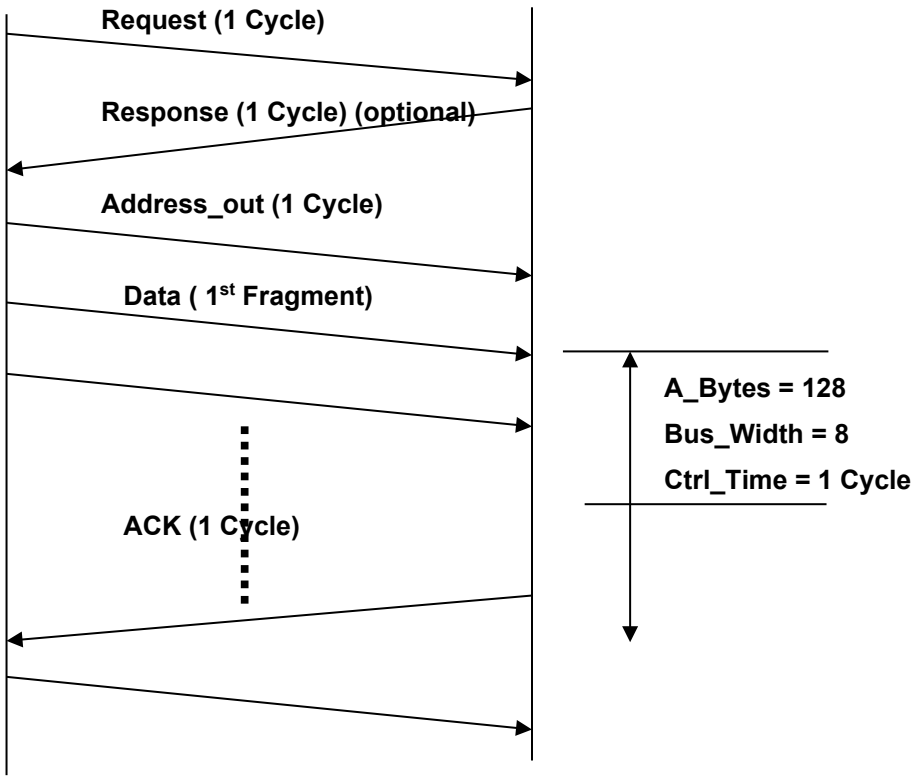


Figure 2: Total Latency for the whole WRITE operation = 22 Cycle

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

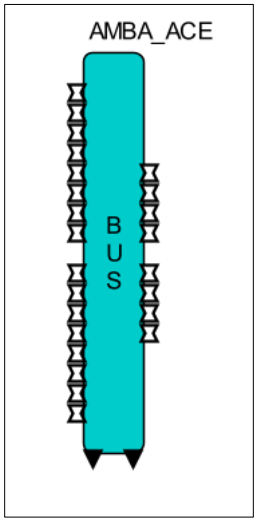
Data Transfer:

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.**

1.4 AMBA ACE Bus

Block Description

The AMBA ACE bus is a coherency extension of the AXI protocol which can be connected to any master devices with or without coherency support. The ACE bus supports all the feature an AXI bus support, it adds three additional channel for the coherency and enables master to master communication.



The coherency channels added on top of the AXI channels:
Snoop Address : Coherent address channel. (ACADDR)
Snoop Response: Coherent response channel. (CRRESP)
Snoop Data : Coherent data channel. (CDDATA)

The Snoop address channel broadcast the snoop command to coherent masters connected to the ACE bus. The Snoop response channel receive response from the coherent masters. The response for a read data can result in multiple response from the masters, the ACE bus select one of the masters based on the FCFS and receive data from that master through Snoop Data channel.

Input Fields for coherency support:

Snoop_Command – contains the snooping commands that will be broadcasted to all the coherent masters

Coherency Parameters on top of AXI bus:

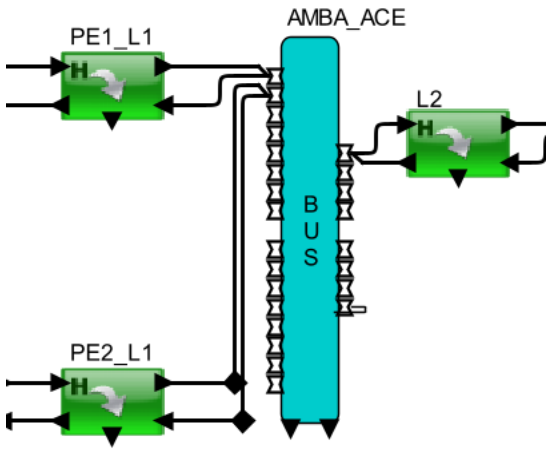
Coherent_Masters:
This parameter enables the coherency support for specific masters. The other masters will acts as a non coherent masters just as it is connected to the AXI bus.
Eg: {1,2,3} - master 1, 2 and 3 are enabled for coherency support.

Edit parameters for AMBA_ACE

Architecture_Name:	"Architecture_1"
Bus_Name:	"ACE_Bus"
AXI_Speed_Mhz:	1000.0
AXI_Cycle_Time:	1.0E-06 / AXI_Speed_Mhz
_explanation:	Interfaces and Buses->AHB->AXI_Bus
Bus_Width:	8
Read_Threshold:	2
Write_Threshold:	2
Master_Request_Threshold:	{2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}
Number_Masters:	16
Number_Slaves:	8
Threshold_Trans_T_Bytes_F:	true
Arbiter_FIX_1_RR_2_CUSTOM_3:	1
Slave_Speeds_Mhz:	ız, AXI_Speed_Mhz,AXI_Speed_Mhz, AXI_Speed_Mhz, AXI_Speed_Mhz, AXI_Speed_Mhz}
Extra_Cycles_for_RdReq_WrReq_RdData_WrData:	{0, 0, 0, 0, 0, 0, 0, 0}
Devices_Attached_to_Slave_by_Port:	ɜ_2"},{"Device_3"},{"Device_4"},{"Device_5"},{"Device_6"},{"Device_7"},{"Device_8"}}
Master_First_Word_Flag:	true
Master_Throttle_Enable:	{false,false,false,false,false,false,false,false,false,false,false,false,false,false,false,false}
Slave_Throttle_Enable:	{false,false,false,false,false,false,false,false}
DEBUG:	false
Custom_Arbiter_File:	"none"
Custom_Arbiter_Path:	"none"
Fixed_Priority_Array:	,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}}
Slave_First_Word_Flag:	true /* Not Active in Default Slave */
Custom_Slave_File:	"none"
Ports_to_Plot:	{0,0} /* master n, slave m, 0 disables */
Coherent_Masters:	{1,2,3}

Commit | Add | Remove | Restore Defaults | Preferences | Help | Cancel

The following image shows a coherent connection of multiple masters with ACE bus.



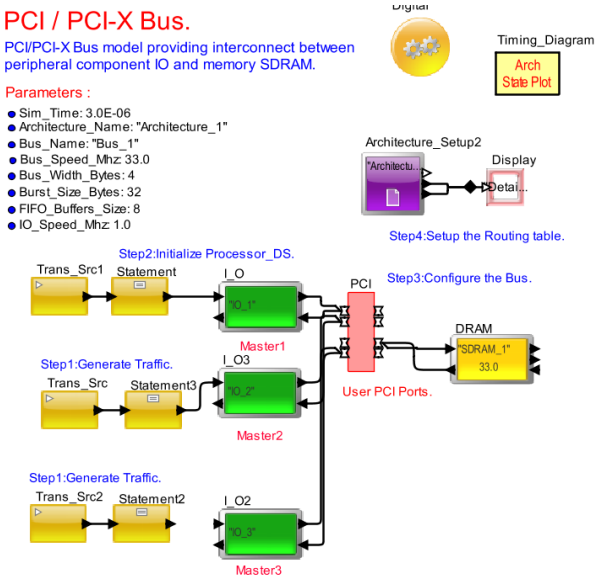
2 PCI Family of Buses

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.

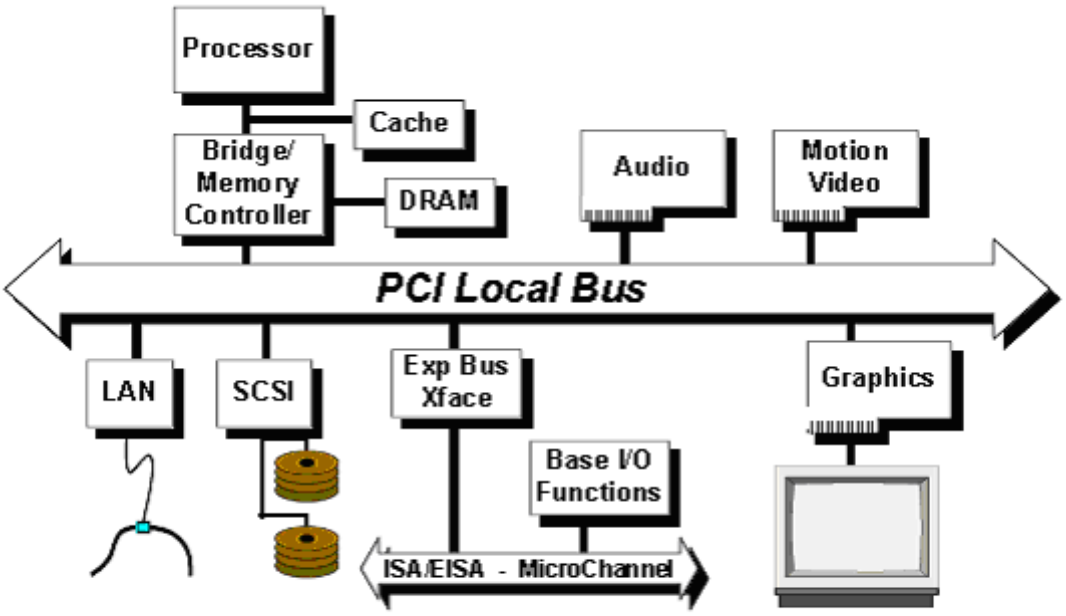
A_First_Word – default is true, Bus block sets to true, if the first word is sent out in a fragment, otherwise set to false.
A_Priority – can set the priority of the device, higher priority master gained the bus access. The multiple transactions flowing into the bus with different priorities are covered 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**
A_Bytes	int	128
A_Command	string	"Read_Memory"
A_First_Word	boolean	true
A_Priority	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.

2.1.5 A typical PCI, PCI-X system



PCI System Block Diagram

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_MBs*** – 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 = 10.6666666666667,
Bus_1_Input_Buffer_Occupancy_in_Words_Min = 0.0,
Bus_1_Input_Buffer_Occupancy_in_Words_StDev = 15.084944665313,
Bus_1_Throughput_MBs_Max = 85.3333333333333,
Bus_1_Throughput_MBs_Mean = 85.3333333333333,
Bus_1_Throughput_MBs_Min = 85.3333333333333,
Bus_1_Throughput_MBs_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_MBs_Max = 96.0,
SDRAM_1_Throughput_MBs_Mean = 96.0,
SDRAM_1_Throughput_MBs_Min = 96.0,
SDRAM_1_Throughput_MBs_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 = 61.3333333333333,
SDRAM_1_Utilization_Pct_Mean = 61.3333333333333,
SDRAM_1_Utilization_Pct_Min = 61.3333333333333,
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:

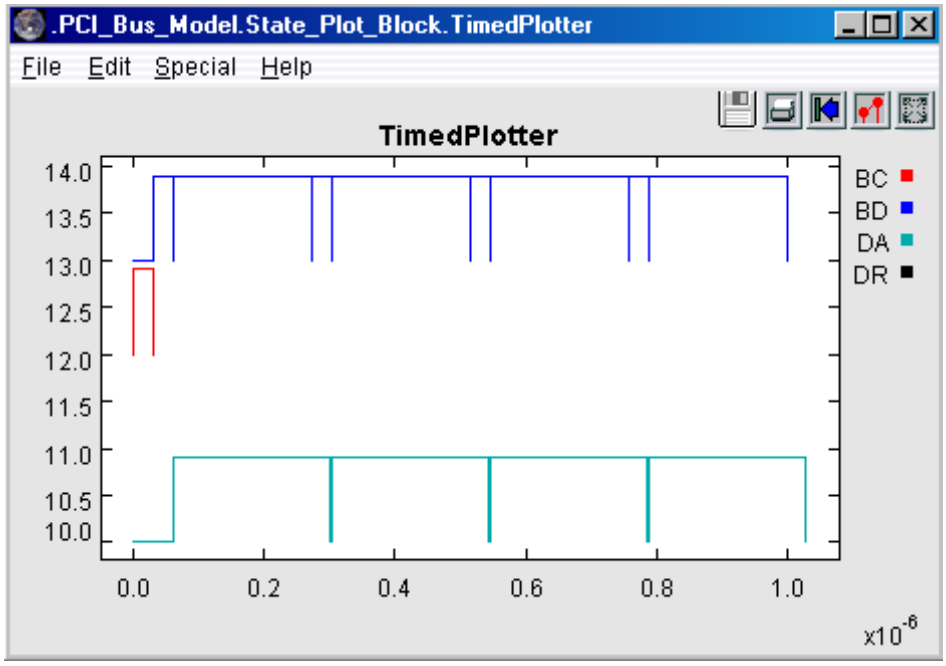
Write Transaction:

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

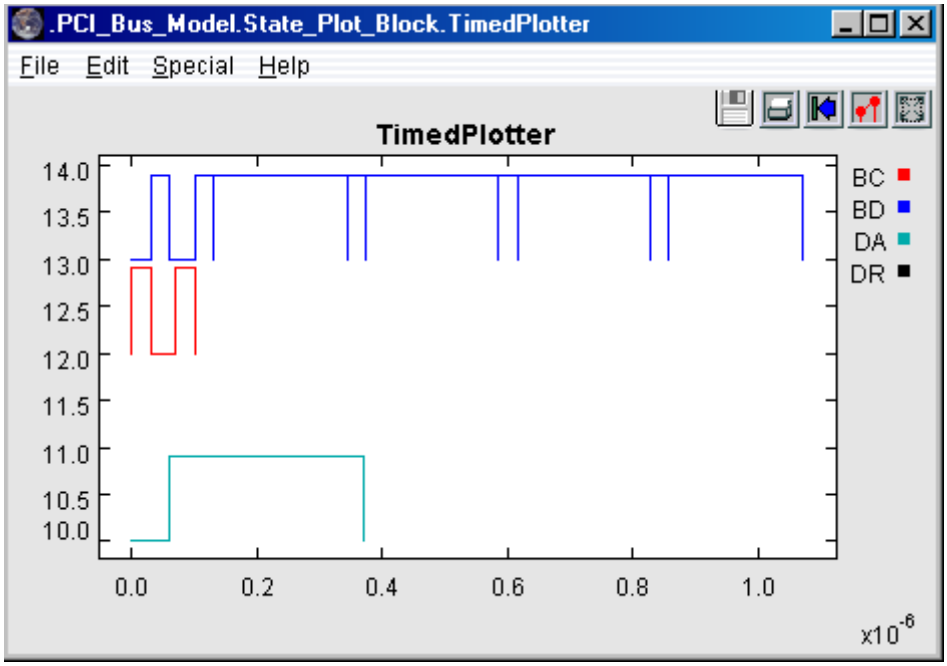
Burst #4
First fragment : {true, 128, 0, 32, “Write”} //sent first word, 4 bytes
Second fragment: {false, 128, 0, 32, “Write”} //sent remaining words, 28 bytes

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

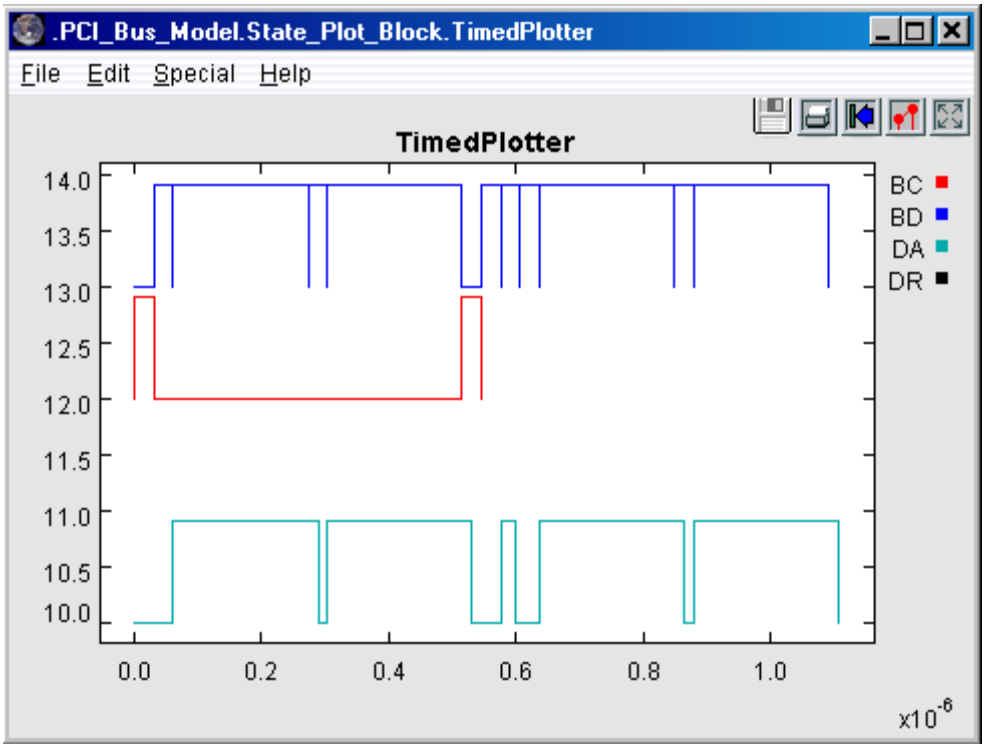
The PCI_Bus block is designed to have more than one master device. If several devices may require the use of the PCI bus to perform a data transfer at the same time, the PCI arbiter effectively control these devices in the listed arbitration schemes in *chapter 4, Architecture Modeling.*

- 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

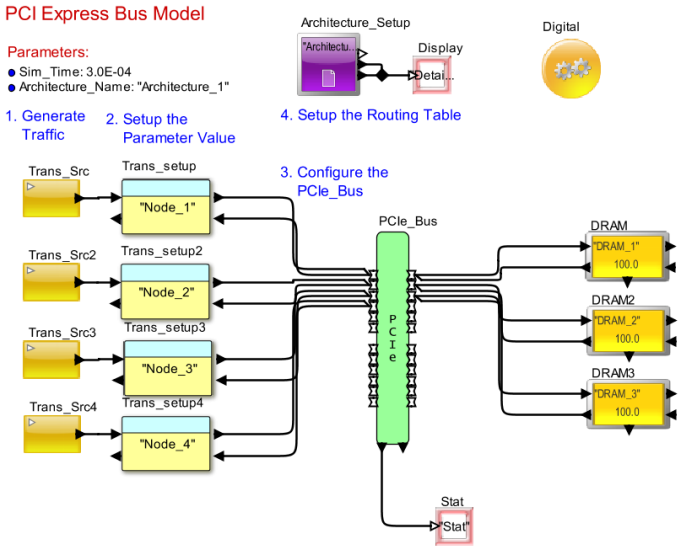


Figure 2-1 PCIe Bus Model

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 intput/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,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,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

Port Name	*Type*	*Explanation*
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 8List 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.

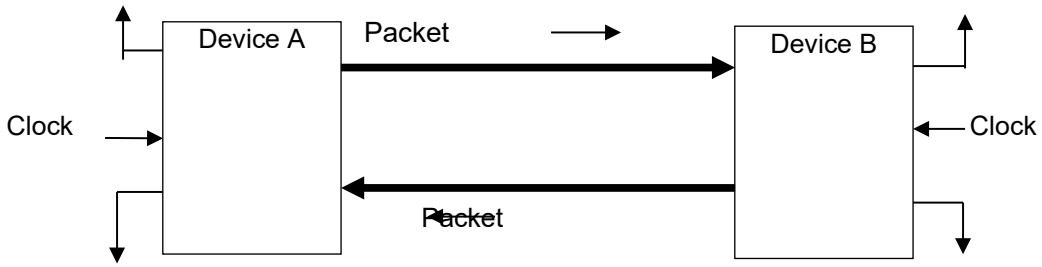


Fig. 1.2 A PCI Express link uses transmit and receive signal pairs

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.

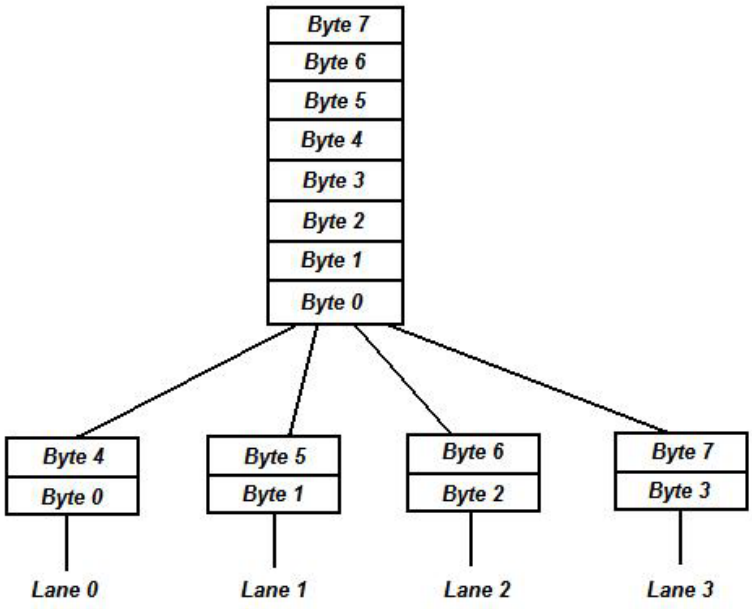


Fig. 1.3 A PCI Express Link consists of one or more lanes

2.2.6 Bus Statistics

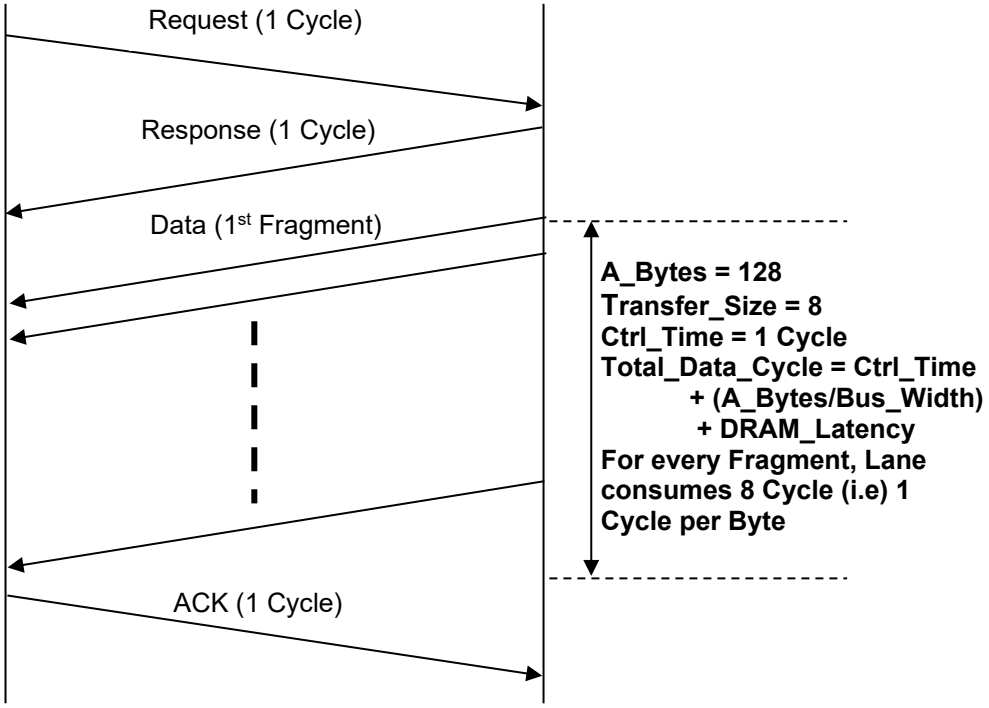
Statistics collected for the PCI Express Bus

- 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.

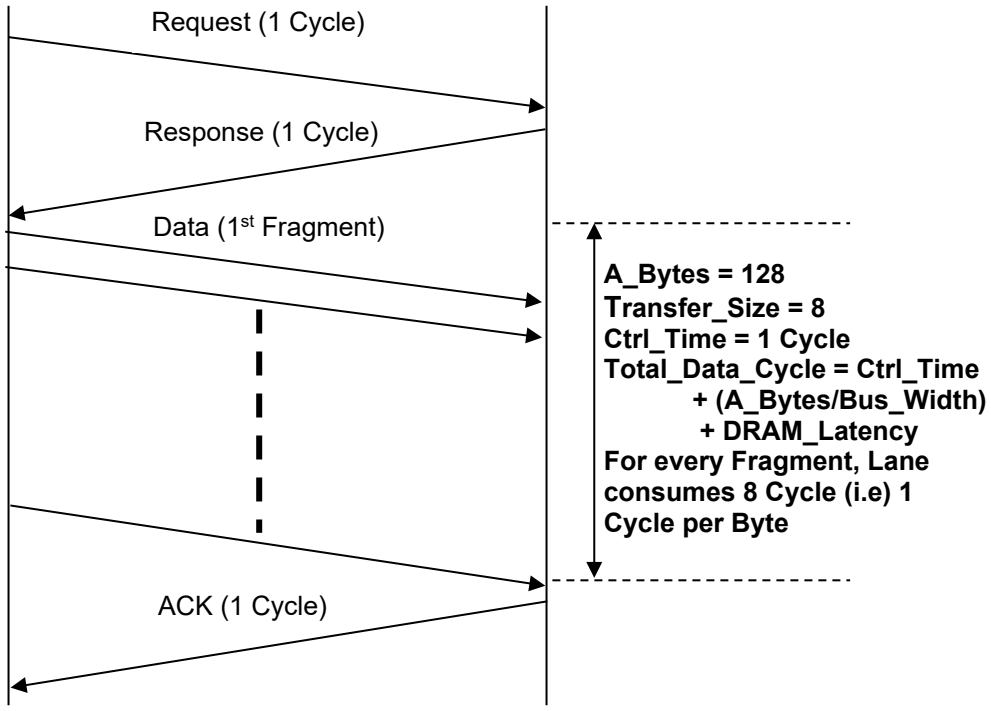
Read Command



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:

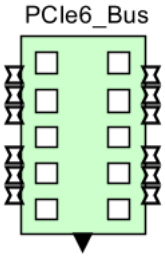
First fragment: {128, 120, 8, Write} //sends Request, 8 bytes

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 PCIe 6

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.



Key features:

1. The PCI Express library comprehends future performance enhancements via speed upgrades and advanced encoding techniques**.**
2. Generation of Bus statistics. Throughput in Mbps, Bus utilization in percentage, Input/Output transactions /sec, Root complex buffer occupancy and End point buffer occupancy values
3. Block debug feature, this enables the user to visualize the active transaction, current time and channel.
4. User can apply flow control for root complex and end point .
5. PAM4 Signalling
6. Forward Error Correction(FEC)
7. FLIT Mode
8. L0P Mode

To include the PCIe block in a model, the following steps must be followed:

- PCIe6_ Bus is available in Interfaces and Buses->PCI->PCIe6_Bus.
- Drag the block and place it in your model
- Now connect the input and outport of all the Root Complex or Masters to the multiport (single port contains both input and output) located on the left side.
- Similarly connect all the End Point or Slave devices to the right-side ports of the PCIe6_Bus.
- Set the block parameters to match your block diagram. Refer to the Parameter selection for the possible parameter values.

Required Input Data structure fields :

A_Source	Source or Root Complex device name

A_Destination	Destination or End Point device name
A_Hop	Transaction flow path towards destination
A_Command	“Read” or “Write” operation
A_Bytes	Number of data Bytes for this transaction
A_Bytes_Remaining	Set Internally to monitor burst transaction flow
A_Bytes_Sent	Set Internally to monitor burst transaction flow

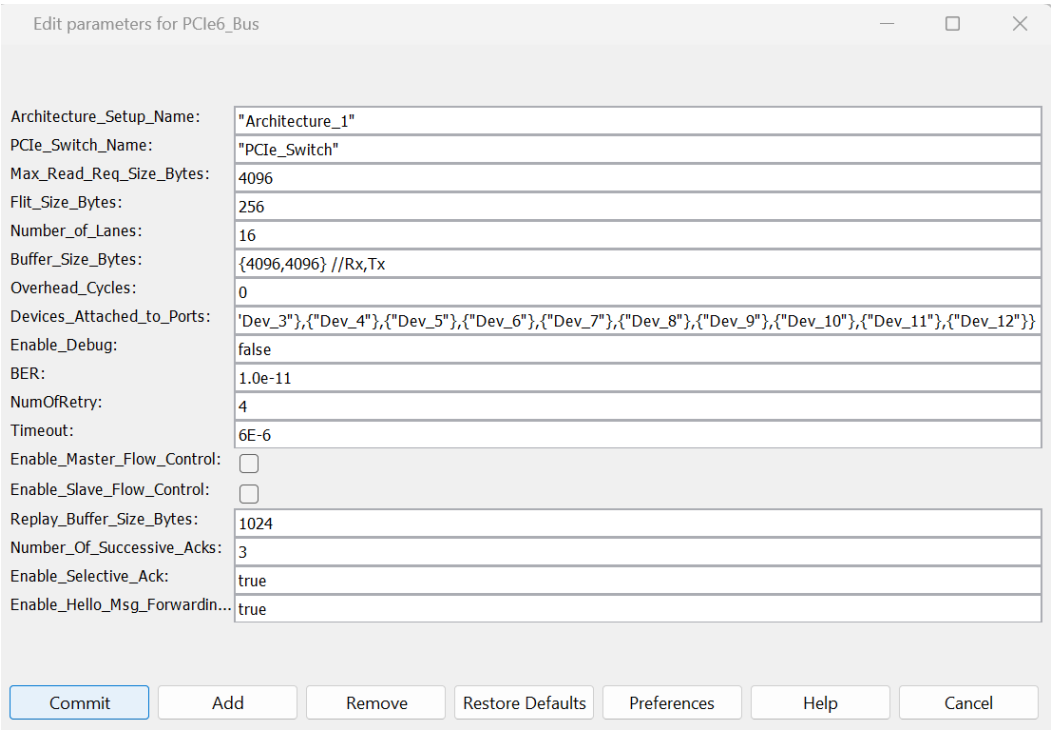
Blocks which can be connected to the PCIe6_Bus

- As long as the incoming data structure contains all requires inputs as specified in Data Structure field.
- PCIe6_Bus can be connected to Device Interfaces and Memory Modules directly to any one of the ports
- PCIe6_Bus can be connected to other interface pheriperals like AMBA_AXI, AMBA_AHB, Switch interface ...etc.

Ports:

- 6 Multiports on each side (left and right). Can be connected directly to the Root Complex, Masters, EndPoints, Slaves, Sinks, Network interfaces/bridges etc.
- Debug Port

Parameters:



Architecture_Name
Name of the Architecture_Setup block
Eg: “Architecture_1” (String)

PCIe_Switch_Name
Unique name for this Bus. Different from all architecture blocks and global model memories.
Eg: “PCIe6_Switch” (String)

Max_Read_Req_Size_Bytes
This is the max Request size for a packet that can be recieved at any Endpoint ports of the switch. If it is greater than the value specified here, then an error is thrown.
Eg: 4096 (int) (bytes)

Flit_Size_Bytes
Flit_Size is the fixed size in Bytes which is used to define the packet size when sending the packet from sender to receiver ports.
Eg: 256 (int)

Number of Lanes
1,2,4,16,32, and 64 are the only possible values. The user is restricted to these values only. This can be a single value or an array of the Number of Master.
Eg: 16(int)

Buffer_Size_Bytes
Number of Bytes irrespective of the number of transactions for Rx and Tx.
Eg: {4096,4096} //RX,TX

Overhead_Cycles
For Each frame - EndPoint Rx Buffer to Crossbar and Crossbar to EndPoint Tx Buffer, Overhead_Cycles will be added to the total delay cycles.
Eg: 0 (int)

Devices_Attached_to_Ports
This is a 2-D array which the user can use to specify the device names of the devices attached to each EndPoint. It the connected Device has the capability to send the Hello Message, then user shouldnt worry about updating this array as it will be updated automatically once it receives the hello message.
Eg: {{"Dev_1"},{"Dev_2"},{"Dev_3"},{"Dev_4"},{"Dev_5"},{"Dev_6"},
{"Dev_7"},{"Dev_8"},{"Dev_9"},{"Dev_10"},{"Dev_11"},{"Dev_12"}}

Enable_Debug
Set it to true and connect a TextDisplay to the Debug_Port (south side of the Switch) to view all the sequences happening within the PCIe_Switch.
Eg: true (boolean)

BER
Range is 0.0 to 1.0. During CRC check, a random number is generated. If the number is below this BER, a Nack is returned. If above, the transaction is accepted.
Eg: 1.0E-11 (double)

NumberOfRetry
If the transfer gets failed due to CRC Check sum failure, then it would be retransmitted. But if the same frame gets failed continuously, then it cannot go above the NumOfRetry value. If it goes above the NumOfRetry value, then the transaction is dropped at the PCIe6_Switch.
Eg: 4 (int)

Timeout
Timeout specifies the amount of time to wait for an acknowledegment to arrive for a FLIT that was transmitted prior. If the Ack is not received within the Timeout expires, then the corresponding FLIT is retransmitted.
Eg: 6E-6 (double)

Enable_Master_Flow_Control
Set this parameter to be true if Flow Control needs to be enabled between the transaction initiator and the PCIe6_Bus. Setting this parameter would mean that All EndPoints are now operating under the effect of Flow Control (Meaning that the incoming Data Structure must contain "Event_Name" field). User can add a field called "Master_Flow_Control_Enable" and set it to be false if the connected EndPoint doesnt wish to have Flow Control Enabled.
Eg: false (boolean)

Enable_Slave_Flow_Control
Set this parameter to be true if Flow Control needs to be enabled between the PCIe6_Bus and Destination EndPoint. Setting this parameter would mean that All EndPoints are now operating under the effect of Flow Control (Meaning that the PCIe6_Bus would wait for the Event to be generated so that it can send the next packet out to the connected EndPoint). User can add a field called "Slave_Flow_Control_Enable" and set it to be false if the destination EndPoint doesnt support Flow Control.
Eg: false (boolean)

Replay_Buffer_Size_Bytes
Store a copy of all Transmitted TLPs until the remote receiver acknowledges them.
Eg: 1024

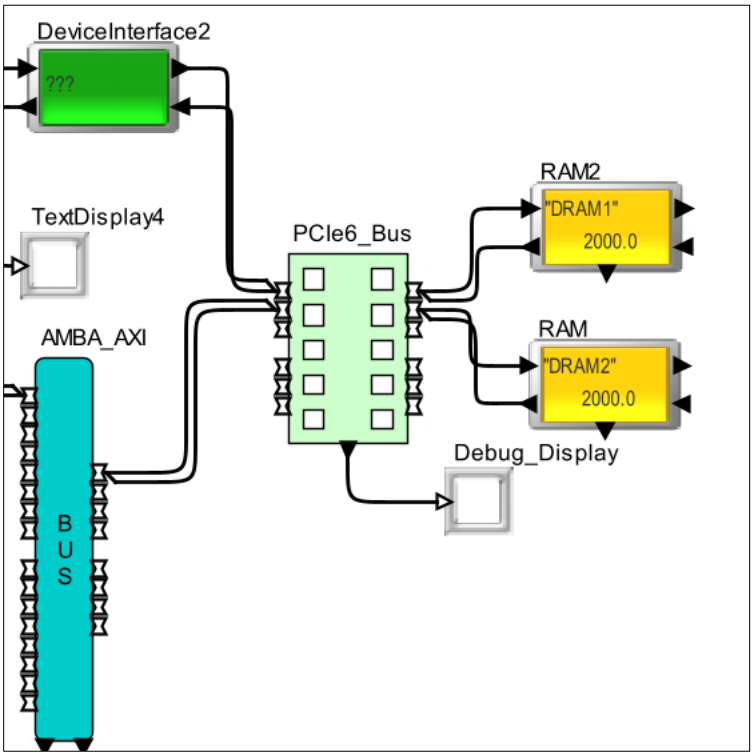
Number_Of_Successive_Acks
Number of Successive Acks used by Selective naks provided Selective Aak is supported. PCIe6 supports Selective Nak on top of classical Nak. With Selective Nak, just a selective FLIT can alone be retransmitted there by saving bandwidth. For this to happen, "n" number of consecutive Naks should be received at the sender . Then the sender identifies it as a retransmission request and retransmits the FLIT. ("n" is the value set for this parameter). For Classical Nak, it request for a group of FLITS to be retransmitted. Often happens when FLITS are received in out of order or a FLIT in between found to be missing etc..
Eg: 3

Enable_Selective_Ack
Setting value true would enable Selective Ack feature on top of Classical Ack.With Selective Nak, just a selective FLIT can alone be retransmitted there by saving bandwidth. For this to happen, "n" number of consecutive Naks should be received at the sender . Then the sender identifies it as a retransmission request and retransmits the FLIT. ("n" is the value set by Number_Of_Successive_Acks). For Classical Nak, it request for a group of FLITS to be retransmitted. Often happens when FLITS are received in out of order or a FLIT in between found to be missing etc..
Eg: true

Enable_Hello_Msg_Forwarding
In VisualSim, All connected devices send a message called "Hello_Message" at startup. This is to make sure that the Buses, interconnects, bridges know where each of the devices are located. Enable this parameter to forward the hello message received on one port to all other ports. In a big network, enabling it would be helpful (provided there are no bus connections in loop).

Eg: true

The following image shows a simple connection of PCIe6 with other devices.



The following are the stats that can be observed in PCIe6 design.

DS_Name = "PCIe_Switch_PCIe_Switch_Stats",
PCIe_Switch_PCIe_Switch_Port_10_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_10_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_10_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_10_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_11_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_11_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_11_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_11_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_12_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_12_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_12_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_12_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_1_Drop_Count = 9372,
PCIe_Switch_PCIe_Switch_Port_1_Rx_MBps = 88152.08815208815,
PCIe_Switch_PCIe_Switch_Port_1_Total_MBps = 178252.17825217824,
PCIe_Switch_PCIe_Switch_Port_1_Tx_MBps = 90100.0901000901,
PCIe_Switch_PCIe_Switch_Port_1_to_Port_8_Max_Latency = 1.23986E-7,
PCIe_Switch_PCIe_Switch_Port_1_to_Port_8_Mean_Latency = 1.1238603392857E-7,

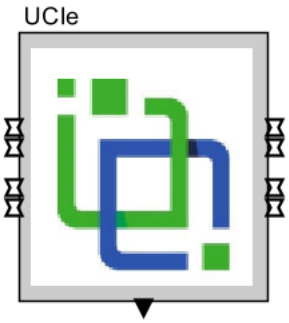
PCIe_Switch_PCIe_Switch_Port_1_to_Port_8_Min_Latency = 4.538E-9,
PCIe_Switch_PCIe_Switch_Port_2_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_2_Rx_MBps = 23680.02368002368,
PCIe_Switch_PCIe_Switch_Port_2_Total_MBps = 47360.04736004736,
PCIe_Switch_PCIe_Switch_Port_2_Tx_MBps = 23680.02368002368,
PCIe_Switch_PCIe_Switch_Port_2_to_Port_7_Max_Latency = 3.0541E-8,
PCIe_Switch_PCIe_Switch_Port_2_to_Port_7_Mean_Latency = 1.75918E-8,
PCIe_Switch_PCIe_Switch_Port_2_to_Port_7_Min_Latency = 4.7379999999999E-9,
PCIe_Switch_PCIe_Switch_Port_3_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_3_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_3_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_3_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_4_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_4_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_4_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_4_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_5_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_5_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_5_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_5_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_6_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_6_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_6_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_6_Tx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_7_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_7_Rx_MBps = 23680.02368002368,
PCIe_Switch_PCIe_Switch_Port_7_Total_MBps = 47360.04736004736,
PCIe_Switch_PCIe_Switch_Port_7_Tx_MBps = 23680.02368002368,
PCIe_Switch_PCIe_Switch_Port_7_to_Port_2_Max_Latency = 1.0072E-8,
PCIe_Switch_PCIe_Switch_Port_7_to_Port_2_Mean_Latency = 7.782275E-9,
PCIe_Switch_PCIe_Switch_Port_7_to_Port_2_Min_Latency = 5.619E-9,
PCIe_Switch_PCIe_Switch_Port_8_Drop_Count = 72,
PCIe_Switch_PCIe_Switch_Port_8_Rx_MBps = 89568.08956808958,
PCIe_Switch_PCIe_Switch_Port_8_Total_MBps = 178252.17825217824,
PCIe_Switch_PCIe_Switch_Port_8_Tx_MBps = 88684.08868408868,
PCIe_Switch_PCIe_Switch_Port_8_to_Port_1_Max_Latency = 1.22804E-7,
PCIe_Switch_PCIe_Switch_Port_8_to_Port_1_Mean_Latency = 1.0054470175439E-7,
PCIe_Switch_PCIe_Switch_Port_8_to_Port_1_Min_Latency = 8.718E-9,
PCIe_Switch_PCIe_Switch_Port_9_Drop_Count = 0,
PCIe_Switch_PCIe_Switch_Port_9_Rx_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_9_Total_MBps = 0.0,
PCIe_Switch_PCIe_Switch_Port_9_Tx_MBps = 0.0}

4 UCIe

Block Description

This library emulates all the functionality of the UCIe interconnect standard, an open specification for a die-to-die interconnect media to integrate the various chiplets. UCIe can accommodate the bulk of designs today from 8 Gbps per pin to 32 Gbps per pin for high-bandwidth applications from networking to hyperscale data centers. Use this IP to model chiplet to chiplet communication and detemiine theoptimal configuration suiting your application. The controller IP supports PCIe, CXL, and other widely used protocols for latency-optimized network-on-chip (NoC)-to NoC links with streaming protocols; for example, bridging to CXS interfaces and to AXI interfaces.

For this block to operate correctly, the model must contain two blocks as a prerequisite-Architecture_Setup, and Digital Simulator.
There are a total of 8 chiplet interface ports and a debug port for this block.



Parameters:

Architecture_Setup_Name
This parameter is used to define the Architecture_Setup block name.

UCIe_Switch_Name
This parameter is used to define a unique name for the UCIe interconnect. Stats generated from this UCIe interconnect will use this name as the identifier.

Package_Type
This is a pull down parameter. User can select Advanced or Standard type. Accordingly the number of lanes available would differ.

Max_Link_Speed_GTps
This is a 1-D array. User can set the chiplet max transfer speed in Giga Transfers per second (GTps).

Protocols_Per_Port
This is a 2-D array. So each index is for "per-port". If multiple protocols are listed per port, then arbitrations is made to select the frame from each protocol layer.

Number_of_Modules
This is a 1-D array. User can set the number of modules setup for each chiplet.

Retimer_Enable_Arr
This is 1-D array parameter. It accepts boolean value. If we want to extend the reach, then we ned to enable retimer. So set the retimer corresponding to the target port where the reach needs to be extended.

Max_Read_Req_Size_Bytes
This is the max Request size for a packet that can be recieved at any ports of the switch. If it is greater than the value specified here, then an error is thrown.

Devices_Attached_to_Ports
This is a 2-D array which the user can use to specify the device names of the devices attached to each port. It the connected Device has the capability to send the Hello Message, then user shouldnt worry about updating this array as it will be updated automatically once it receives the hello message.

Replay_Buffer_Size_Bytes
Size of the buffer which stores a copy of all Transmitted TLPs until the remote receiver acknowledges them.

Retimer_Timeout
This parameter accepts valuein seconds. The corresponding value will be the time which determines the Timeout at a retimer. So once a packet is sent out, the retimer will wait for "Retimer_Timeout" seconds before retransmitting the frame again.

Number_Of_Successive_Acks
Number of Successive Acks used by Successive naks; successive naks are required for the sender to identify and retransmit.

Enable_Selective_Ack
Set this parameter to true to enable selective Nak. In standard Nak, all the frames starting from the unacknowledged frame is done while in selective, only the unacknowledged frame is retransmitted.

Buffer_Size_Bytes
Max bytes that can be stored in Rx Buffer, Tx Buffer and retimer Rx buffer.

Overhead_Cycles
For Each frame - Tx Buffer to Crossbar and Crossbar to Rx Buffer, Overhead_Cycles will be added to the total delay cycles.

BER
Range is 0.0 to 1.0. During CRC check, a random number is generated. If the number is below this BER, a Nack is returned. If above, the transaction is accepted.

NumOfRetry
If the transfer gets failed due to CRC Check sum failure, then it would be retransmitted. But if the same frame gets failed continuously, then it cannot go above the NumOfRetry value. If it goes above the NumOfRetry value, then the transaction is dropped at the UCIe.

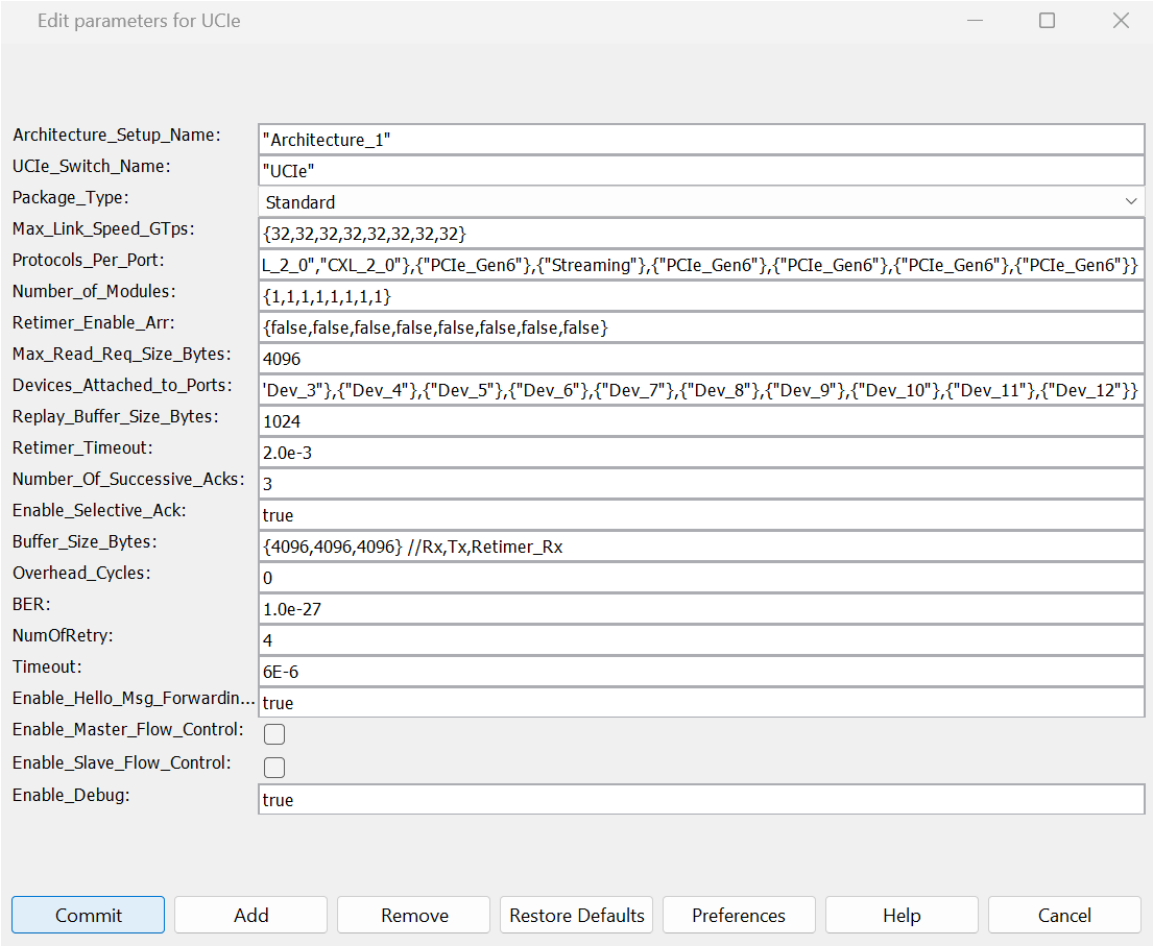
Timeout
Timeout specifies the length of the time to wait for a response when establish the communication. it also represents error or disconnection that occurs because of inactivity of data transmission.

Enable_Hello_Msg_Forwarding
A routing path setup message transmitted from sender to receiver to determine the best path forward.

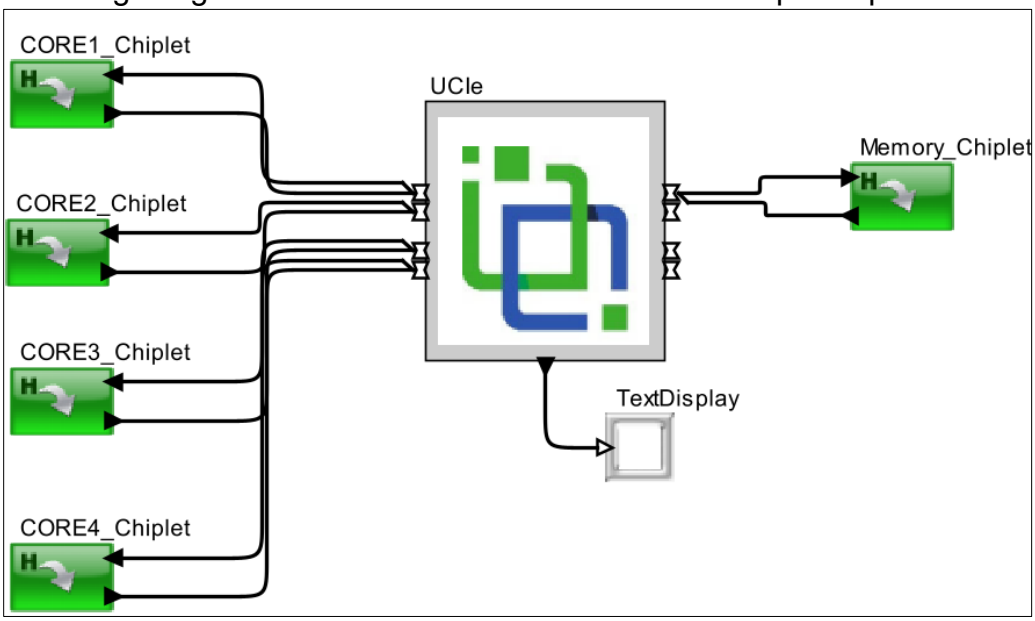
Enable_Master_Flow_Control
Set this parameter to be true if Flow Control needs to be enabled between the transaction initiator chiplet and the UCIe. Setting this parameter would mean that All chiplets are now operating under the effect of Flow Control (Meaning that the incoming Data Structure must contain "Event_Name" field). User can add a field called "Master_Flow_Control_Enable" and set it to be false if the connected chiplet doesnt wish to have Flow Control Enabled.

Enable_Slave_Flow_Control
Set this parameter to be true if Flow Control needs to be enabled between the UCIe and Destination chiplet. Setting this parameter would mean that All chiplets are now operating under the effect of Flow Control (Meaning that the UCIe interconnect would wait for the Event to be generated so that it can send the next packet out to the connected chiplet). User can add a field called "Slave_Flow_Control_Enable" and set it to be false if the destination chiplet doesnt support Flow Control.

Enable_Debug
Set it to true and connect a TextDisplay to the Debug_Port (south side) to view all the sequences happening within the UCIe interconnect.



The following image shows a connection of UCIe with multiple chiplets.



The following are the stats observed in the UCIe design.

DS_Name = "UCIe_Switch_UCIe_Stats",
UCIe_Switch_UCIe_Port_1_Drop_Count = 0,
UCIe_Switch_UCIe_Port_1_Rx_MBps = 0.027256,
UCIe_Switch_UCIe_Port_1_Total_MBps = 0.035576,
UCIe_Switch_UCIe_Port_1_Tx_MBps = 0.00832,
UCIe_Switch_UCIe_Port_1_to_Port_5_Max_Latency = 1.53695313E-7,
UCIe_Switch_UCIe_Port_1_to_Port_5_Mean_Latency = 1.2145312987048E-8,
UCIe_Switch_UCIe_Port_1_to_Port_5_Min_Latency = 4.6953128052252E-9,
UCIe_Switch_UCIe_Port_2_Drop_Count = 0,
UCIe_Switch_UCIe_Port_2_Rx_MBps = 0.022614,
UCIe_Switch_UCIe_Port_2_Total_MBps = 0.02658,
UCIe_Switch_UCIe_Port_2_Tx_MBps = 0.003966,
UCIe_Switch_UCIe_Port_2_to_Port_5_Max_Latency = 1.50257813E-7,
UCIe_Switch_UCIe_Port_2_to_Port_5_Mean_Latency = 9.7468754804576E-9,
UCIe_Switch_UCIe_Port_2_to_Port_5_Min_Latency = 2.3515629443693E-9,
UCIe_Switch_UCIe_Port_3_Drop_Count = 0,
UCIe_Switch_UCIe_Port_3_Rx_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_3_Total_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_3_Tx_MBps = 0.0,
UCIe_Switch_UCIe_Port_4_Drop_Count = 0,
UCIe_Switch_UCIe_Port_4_Rx_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_4_Total_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_4_Tx_MBps = 0.0,
UCIe_Switch_UCIe_Port_5_Drop_Count = 0,
UCIe_Switch_UCIe_Port_5_Rx_MBps = 0.012342,
UCIe_Switch_UCIe_Port_5_Total_MBps = 0.0621,
UCIe_Switch_UCIe_Port_5_Tx_MBps = 0.049758,
UCIe_Switch_UCIe_Port_5_to_Port_1_Max_Latency = 3.0007813123234E-8,
UCIe_Switch_UCIe_Port_5_to_Port_1_Mean_Latency = 1.7382812993454E-8,

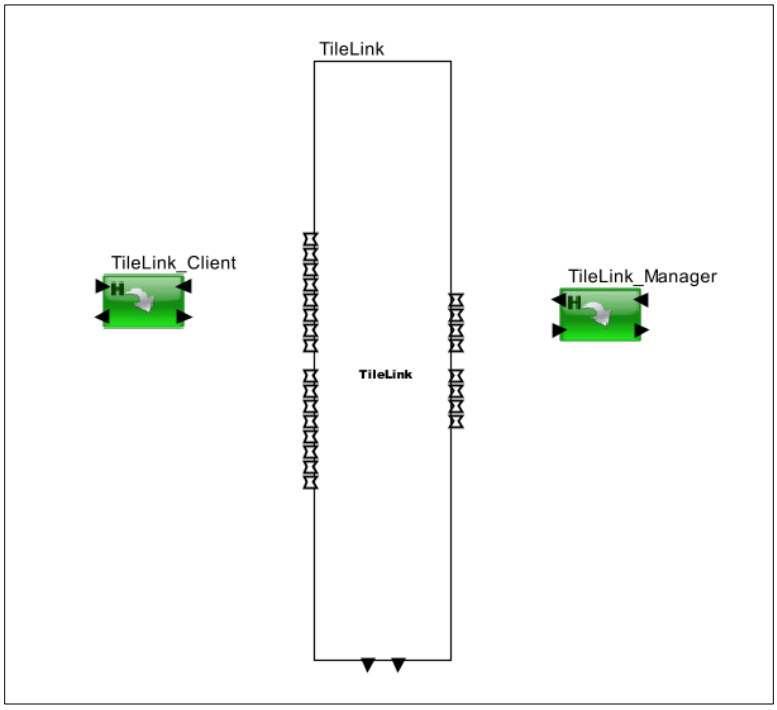
UCIe_Switch_UCIe_Port_5_to_Port_1_Min_Latency = 4.7578128103964E-9,
UCIe_Switch_UCIe_Port_5_to_Port_2_Max_Latency = 2.6164063138268E-8,
UCIe_Switch_UCIe_Port_5_to_Port_2_Mean_Latency = 1.3742188008697E-8,
UCIe_Switch_UCIe_Port_5_to_Port_2_Min_Latency = 1.3203129700656E-9,
UCIe_Switch_UCIe_Port_6_Drop_Count = 0,
UCIe_Switch_UCIe_Port_6_Rx_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_6_Total_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_6_Tx_MBps = 0.0,
UCIe_Switch_UCIe_Port_7_Drop_Count = 0,
UCIe_Switch_UCIe_Port_7_Rx_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_7_Total_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_7_Tx_MBps = 0.0,
UCIe_Switch_UCIe_Port_8_Drop_Count = 0,
UCIe_Switch_UCIe_Port_8_Rx_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_8_Total_MBps = 5.6E-5,
UCIe_Switch_UCIe_Port_8_Tx_MBps = 0.0}

5 TileLink

Block Description:
TileLink stands out as a highly efficient interconnect protocol tailored specifically for RISC-V based System-on-Chip (SoC) architectures, offering several key advantages over other protocols. One of its notable strengths is its support for cache block motion and multiple levels of cache. TileLink facilitates seamless movement of data blocks between different cache levels within the system hierarchy, optimizing data access latency and enhancing overall performance. Additionally, TileLink is designed with a deadlock-free architecture, ensuring uninterrupted transaction progress, crucial for maintaining system reliability and performance, particularly in complex SoC setups with numerous processing elements and peripherals. Moreover, TileLink boasts high performance and scalability, seamlessly integrating with RISC-V processors and accommodating various SoC configurations. Its implementation and verification processes are straightforward, contributing to its simplicity and ease of use.

Furthermore, TileLink supports coherent memory access and offers advanced features such as fine-grained access control and configurable quality of service (QoS), providing flexibility and customization options to meet diverse application requirements.

In embedded systems, multicore configurations, networking equipment, storage solutions, custom SoC designs, and general-purpose computing tasks, TileLink delivers unparalleled efficiency, reliability, and adaptability, ensuring seamless operation and optimal performance across diverse comptuting environments.



The TileLink protocol is defined in terms of a graph of connected agents that send and receive messages over point-to-point channels within a link to perform operations on a shared address space.

Standard Definitions :

***Operation*:** A change to an address range's data values, permissions or location in the memory hierarchy.

Agent: An active participant in the protocol that sends and receives messages in order to complete operations.

Channel: A one-way communication connection between a master interface and a slave interface carrying messages of homogeneous priority.

Message: A set of control and data values sent over a particular channel.

Link: The set of channels required to complete operations between two agents.

Standard Messages:

The TileLink follows the standard message flows within the block.

Acquire

Sent by a master to request ownership or permission for a cache block.Specifies the block and desired ownership/permissions.

GrantData

Sent by a slave in response to an Acquire message.Grants permission for the requested data block.

ReadData

Sent by a slave in response to a Read request.Provides the requested data.

Release

Sent by a master to release ownership or permission for a data block.Indicates the master no longer requires access to the block.

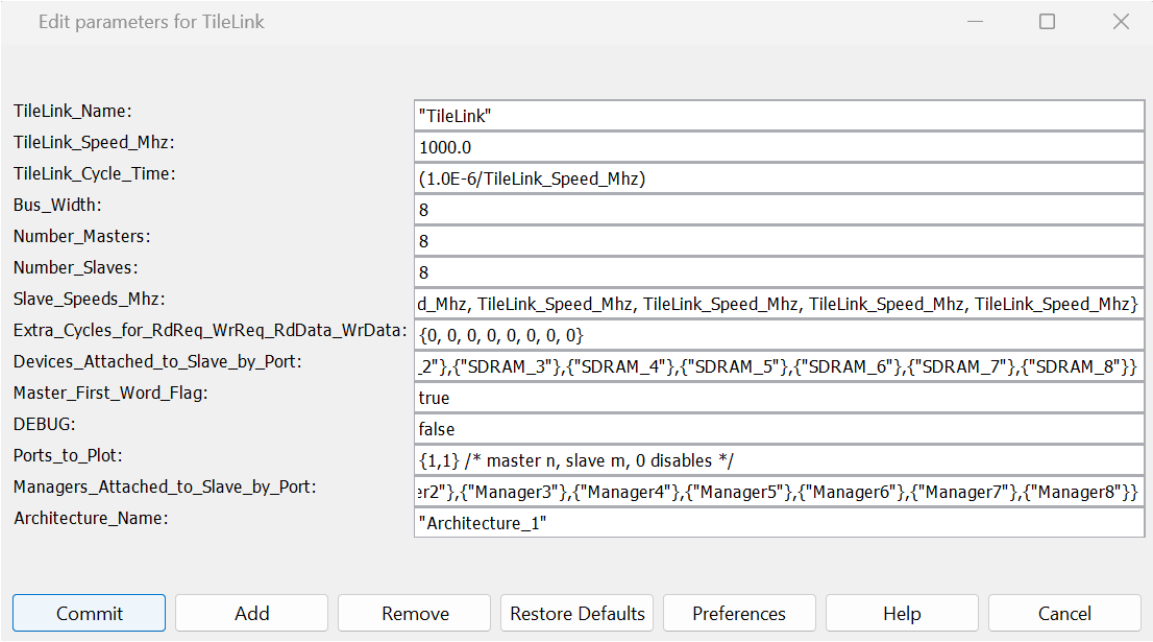
GrantAck

Acknowledges receipt of GrantData by the requesting master.Confirms successful acquisition of ownership/permissions.

Required Input Fields:

A_Bytes
A_Bytes_Remaining
A_Bytes_Sent
A_Command
A_Priority
A_Source
A_Destination
A_Task_Flag

Parameters of TileLink:



TileLink_Speed_Mhz
Speed of the bus in terms of Megahertz. It can be linked to the top level parameter.
Eg: 1000.0

TileLink_Cycle_Time
Cycle time for all the Master ports are defined using the TileLink_Speed_Mhz parameter.
Eg: 1.0E-06 / TileLink_Speed_Mhz

Bus_Width
It defines the maximum bytes that the bus can transfer within a single cycle. User can define the value based on the requirement. It can be linked to top level parameter.
Eg: 8

Number_Masters
It defines the number of actively configured masters that are connected to the TileLink Bus.
Eg: 16

Number_Slaves
It defines the number of actively configured slaves that are connected to the TileLink Bus
Eg: 8

Slave_Speeds_Mhz
It defines the speed of the each slave in this TileLink bus. This can be used to define the clock cycle and internal timing of each slave. Each index is for the slave ports in order.
Eg: {TileLink_Speed_Mhz, TileLink_Speed_Mhz, TileLink_Speed_Mhz, TileLink_Speed_Mhz,TileLink_Speed_Mhz, TileLink_Speed_Mhz, TileLink_Speed_Mhz, TileLink_Speed_Mhz}

Extra_Cycles_for_RdReq_WrReq_RdData_WrData
It defines the overhead delays for each operation in the TileLink bus. Each points to the corresponding operations specified in the parameter.
User can put number of delay cycles based on the requirement. Array with 0 values will not provide any delay to that operation.
Eg: {0, 0, 0, 0}

Devices_Attached_to_Slave_by_Port
This is a array of arrays. Each array contains the list of devices that are accessed via a Slave Port and are connected to this port through a TileLink Manager.
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.
TileLink bus uses this information for routing the input request reaching the master.
If user forgets to give the slave name then the bus will throw an error.
Eg: TileLink_Master_1 did not find Slave named L2
{{"Cache"}, {"SDRAM"}, {"SDRAM3"}, {"Device_4"}}

Managers_Attached_to_Slave_by_Port
This is a array of arrays. Each array contains the list of devices that are accessed via a Slave Port and are connected to this 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.
If user forgets to give the slave name then the bus will throw an error.
Eg: Cannot Find destination (Manager) among possible destinations
{{"Manager"}, {"Manager2"}, {"Manager3"}, {"Manager4"}}

Master_First_Word_Flag
It defines that the TileLink bus will send the First word (to master device) whenever the master receives data from the slave.
User can modify it as false if the bus has to send the last word of that request data.
Eg: true
DEBUG
This will help the user to see how the transaction is happening and the statistics of the TileLink bus. User can perform debugging by setting the value true and connecting a text display to TileLink's "stats_out" port.
Eg: False

Ports_to_Plot
This will enable the TileLink Bus to generate ploting information for specific master and slave. User can observe the Timing diagram for the internal operations.
This can be achieved by connecting a TimeDataPlotter to the plot out port in TileLink.
{m,n}// enable Timing Diagram for mth master, nth slave
where m and n are integers from 1 to number of ports
Eg: {0,0} disables the plot

TileLink Client:
The TileLink Client block provides an interface between the core processing component such as L1 cache and the TielLink bus. It initiates the internal commands to the TileLink manager according to the input commands.

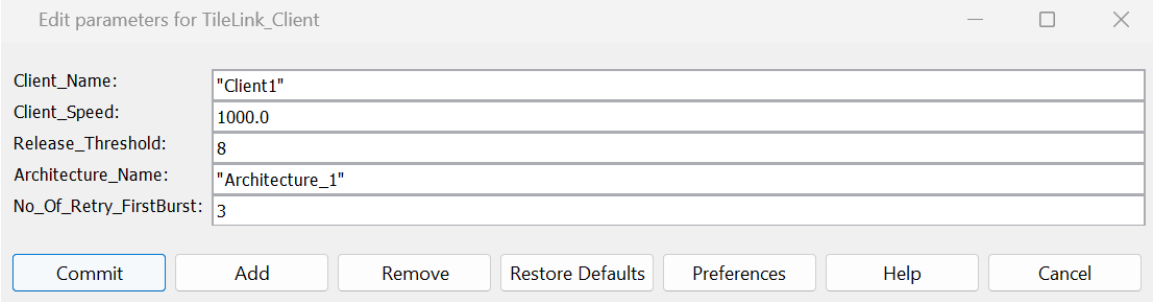
Client Parameter:
Client_Name
It defines unique name of client in the TileLink network.

Client_Speed
It defines the speed of the client that is connected to a master device.

Release_Threshold
It defines the threshold for number of release in the request grant messsage.

Architecture_Name
It defines the name of the architecture setup block.

No_of_Retry_FirstBurst
It defines the number of retry that can be initiated on the first word reject.



TileLink Manager :

The TileLink manager provides an interface between the slave devices such as L2 cache or DRAM with TileLink bus. It initiate commands and handles the request from the client devices.

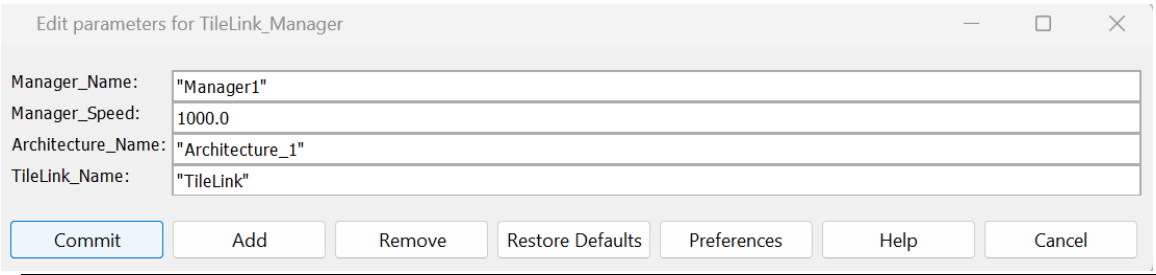
Manager parameter:

Manager_Name
It defines a unique name for the manager in the TileLink network.

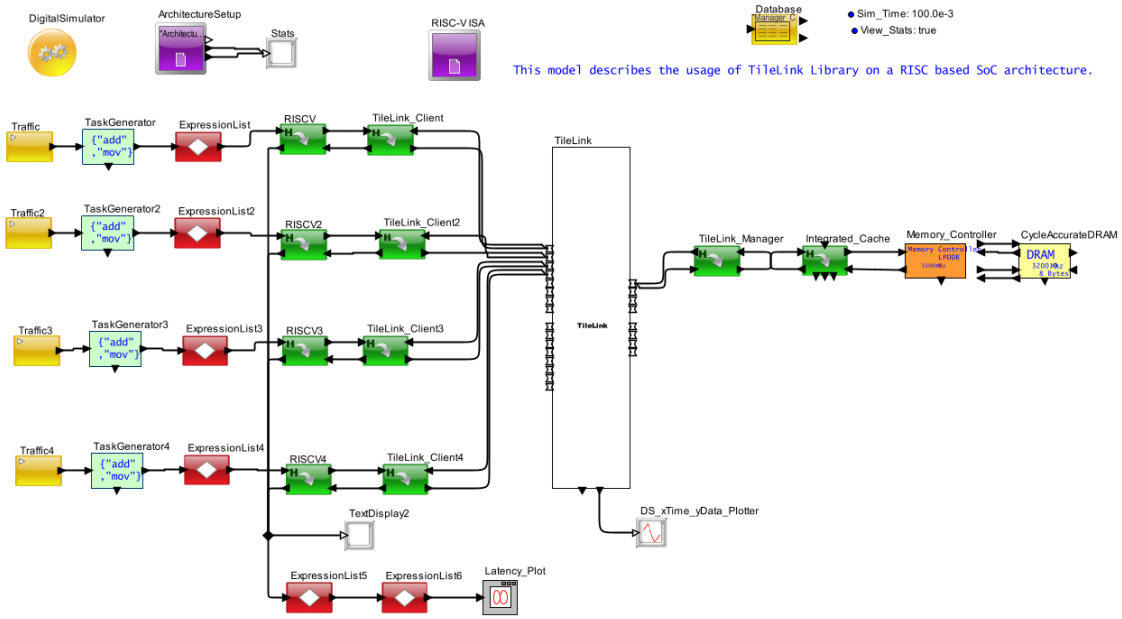
Manager_Speed
It defines the speed of the manager which is connected to a slave device.

Architecture_Name
It defines the name of the architecture setup block.

TileLink_Name
It defines the name of the TileLink bus in which the manager is connected.



The following image shows a multi core RISC V architecture using TileLink interconnect.



The following are the stats that can be observed in a TileLink design.

{DS_NAME = "TileLink_Packets per second",
TileLink_Channel_A_Packets per second = 18.00000018,
TileLink_Channel_C_Packets per second = 2.100000021,
TileLink_Channel_D_Packets per second = 20.100000201,
TileLink_Channel_E_Packets per second = 18.00000018}

{BLOCK = "TileLink_Topology.TileLink.TileLink_Channel_A",
DS_NAME = "TileLink_Channel_A_4",
Number_Entered = 100,
Number_Exited = 100,
Occupancy_Max = 1.0,
Occupancy_Mean = 0.5,
Occupancy_Min = 0.0,
Occupancy_StDev = 0.5,
Total_Delay_Max = 1.0000000000005E-9,
Total_Delay_Mean = 1.0E-9,
Total_Delay_Min = 9.999999999988E-10,
Total_Delay_StDev = 3.7682219008411E-17}

{BLOCK = "TileLink_Topology.TileLink.TileLink_Channel_C",
DS_NAME = "TileLink_Channel_C_4",

Number_Entered = 12,
Number_Exited = 12,
Occupancy_Max = 1.0,
Occupancy_Mean = 0.5,
Occupancy_Min = 0.0,
Occupancy_StDev = 0.5,
Total_Delay_Max = 1.0000000000005E-9,
Total_Delay_Mean = 9.9999999999993E-10,
Total_Delay_Min = 9.999999999988E-10,
Total_Delay_StDev = 1.8503717077086E-17}

{BLOCK = "TileLink_Topology.TileLink.TileLink_Channel_D",
DS_NAME = "TileLink_Channel_D_1",
Number_Entered = 9,
Number_Exited = 9,
Occupancy_Max = 1.0,
Occupancy_Mean = 0.5,
Occupancy_Min = 0.0,
Occupancy_StDev = 0.5,
Total_Delay_Max = 1.0E-9,
Total_Delay_Mean = 1.0E-9,
Total_Delay_Min = 9.9999999999997E-10,
Total_Delay_StDev = 0.0}

{BLOCK = "TileLink_Topology.TileLink.TileLink_Channel_E",
DS_NAME = "TileLink_Channel_E_3",
Number_Entered = 44,
Number_Exited = 44,
Occupancy_Max = 1.0,
Occupancy_Mean = 0.5,
Occupancy_Min = 0.0,
Occupancy_StDev = 0.5,
Total_Delay_Max = 1.0000000000001E-9,
Total_Delay_Mean = 1.0E-9,
Total_Delay_Min = 9.9999999999986E-10,
Total_Delay_StDev = 2.8547135612432E-17}

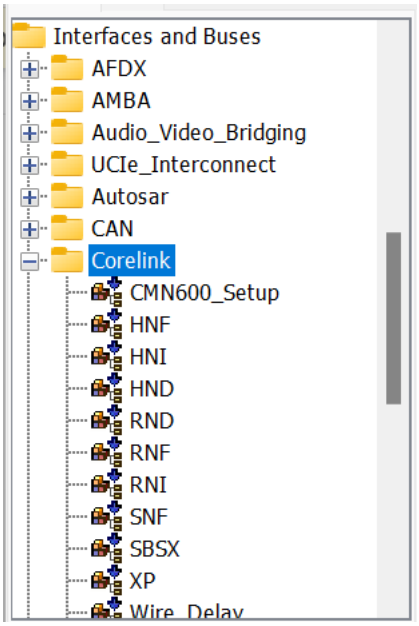
6 Network On Chip

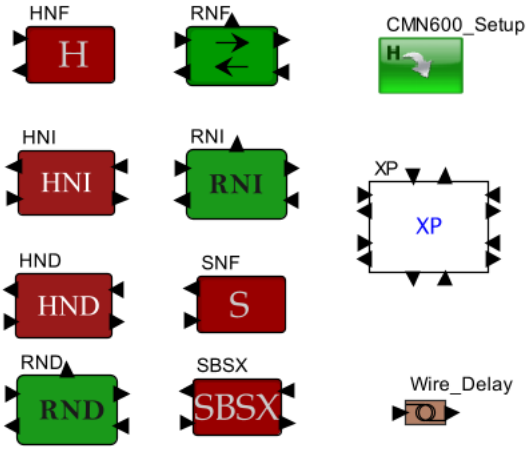
CMN600

Block Description

The Corelink NoC, CMN600 library modules can be used for configuring versatile use cases. The library modules provide a scalable configurable interconnect that is designed to meet the power and performance requirements for coherent mesh network systems that are used in high end networking and enterprise compute applications.

Crossbars are used between the Nodes(Cores) to transfer data and requests. The current design is based on the ARM Corelink Cache Coherent Network which is used as a cache coherency mechanism in a system with several IP Cores. The types of devices present in the NoC are Request Nodes, Home Nodes, and XP's. There are two types of request and home nodes which are modelled in this project. RNF (Fully Coherent Request Node), RNI (I/O Request Node), HNF (Fully Coherent Home Node), and HNI (I/O Home Node). The RNFs are fully coherent master devices that represent the processor cores in the NoC. The HNFs act as coherent region of the Memory which take in requests from the RNFs, handle snoops, and communicate with the cache and DRAM, and handles responses and acknowledgements. The RNIs and HNIs represent the I/O devices where the RNI sends data to the HNI, which acts as a sink.





Operation:

1. **ReadNoSnp – Read without Snoop**

HNF sends a response back to the Request Node. Sends a read request for the corresponding address to the Cache. The Cache sends data back to the HNF which in turn is sent to the Request Node.

2. **ReadShared – Read with Snoop**

HNF sends a response back to the Request Node. It checks the database to see if any other Request Node has written the data at the same address before. If already written, it sends a data snoop request to the other RNF. The other RNF, after receiving the snoop request

sends the data back to the HNF. Now the HNF forwards the data to the original RNF which requested the data.
If the HNF did not find a previous write on the same address by other RNFs, it sends a read request to the Cache and sends the data back to the RNF.

3. **WriteNoSnp – Write without Snoop**

HNF sends a response back to the Request Node. The RNF, after receiving the response from the HNF, sends the data to be written. The HNF gets the data and passes it on to the Cache. After writing the Cache sends a response to the HNF, which in turn will send an Acknowledgment back to the RNF.

4. **WriteSnp – Write with Snoop**

HNF sends a response back to the Request Node. The HNF checks the database to see if any previous reads have occurred on the same address by other request nodes. If Yes, the HNF sends notifications to the corresponding RNFs. The RNF, after receiving the response for data from the HNF, sends the data to be written. The HNF gets the data and passes it on to the Cache. After writing the Cache sends a response to the HNF, which in turn will send an Acknowledgment back to the RNF.

5. **ReadNotSharedDirty – Read clean or unique(dirty) data but not the Shared dirty**

HNF uses Snoop filter (database) to see is there any other RNFs shared the specific address blocks. If any RNF cached the block in clean or unique state then the HNF sends a response to the requestor and also sends a data request to the RNF. If no RNFs cached the specific block then it will get the data from the HNF cache slice which may get from the Main memory if it is a miss.

6. **ReadClean – Read clean data**

HNF uses Snoop filter (database) to see is there any other RNFs shared the specific address blocks and available in the clean state. If any RNF have clean data then the HNF sends a response to the requestor and also sends a data request to the RNF. The RNF will transfer the data to the requestor through the HNF.

7. **CleanUnique**

HNF uses snoop filter to see any RNF cached the specific block. If any other RNFs are cached that block, then HNF sends a request to remove that line or update that line to main memory. Also a response will be sent to the requestor to acknowledge the request.

Command Flow details:

ReadNoSnp --> Read No Snoop

1. RNF_1 sends Request to HNF_1
2. HNF_1 sends Response to RNF_1
3. HNF_1 reads Local Cache
4. HNF sends Data to RNF_1

ReadShared --> Read Snoop

1. RNF_1 sends Request to HNF_1
2. HNF_1 sends Response to RNF_1
3. HNF_1 reads Database
4. If Data already written by RNF_2 ?
    1. Yes = HNF sends Data Request to RNF_2
        1. RNF_2 sends Data to HNF_1
        2. HNF_1 sends Data to RNF_1
    2. No = HNF reads Cache --> HNF sends Data to RNF

WriteNoSnp --> Write No Snoop

1. RNF_1 sends Request to HNF_1
2. HNF_1 sends Response to RNF_1
3. RNF_1 sends Data to HNF
4. HNF_1 writes Data to Cache
5. HNF_1 sends Ack to RNF_1 after writing

WriteSnp --> Write Snoop

1. RNF_1 sends Request to HNF_1
2. HNF_1 sends Response to RNF_1
3. HNF_1 reads Database
4. If Data already read by RNF_2 ?
    1. Yes = HNF_1 Informs RNF_2
5. RNF_1 sends Data --> HNF_1 writes Data to Cache
6. HNF_1 sends Acknowledgement to RNF_1 after writing

ReadNotSharedDirty --> Read clean or unique but dirty

1. RNF_1 sends Request to HNF_1
2. HNF_1 sends Response to RNF_1
3. HNF_1 reads Database
4. If Data already read by RNF_2 ?
    1. Yes = HNF_1 Informs RNF_2
5. RNF_2 sends Data --> HNF_1 writes Data to Cache if it is dirty
6. HNF_1 sends data to RNF_1 after writing

ReadClean --> Read clean

1. RNF_1 sends Request to HNF_1
2. HNF_1 sends Response to RNF_1
3. HNF_1 reads Database --> If Data already read by RNF_2 ?
    1. Yes = HNF_1 Informs RNF_2
        1. RNF_2 sends Data
        2. HNF_1 writes Data to Cache if it is dirty
    2. No = HNF will get the data from the local cache slice
4. HNF_1 sends data to RNF_1

CleanUnique --> Clean other RNFs

1. RNF_1 sends Request to HNF_1
2. HNF_1 informs other RNF
3. HNF_1 sends Ack to RNF_1

Crossbar (XP)

XP blocks within the SoC guide the incoming requests and data to the right path by reading the destination field of the packets. The current design of the routers support two types of routing methods.

1. Incoming packets possess the path from source to destination

2. Incoming packets possess the destination address alone

4 channels are defined for the packets and according to the channel type each message is given a delay and sent out.
The XP also supports 2 devices at a time, where the device can be a home node or a request node. The routers also use the database “Forwarding Table” to find the destination of the packets. A delay block is added to the wires between two routers to account for the overall wire delay.
Upon Starting simulation each connected modules to a XP will send hello messages to each other modules so that we can create a record of where to find specific modules.

Parameters:
Router_Frequency
Used to specify the XP frequency. Delays are calculated based on this.

Ingress_Buffer_Size
Used to specify the Ingress Queue size.

VC_Buffer_Size
Used to specify the Egress Queue size.

Node_Name
Used to specify the XP module name. This has to be unique when compared to other XP's.

VLAN_Q
Max number of Virtual Channel available

Single_bit_error_ratio
Used to specify Error ratio for Single bit Corruption.

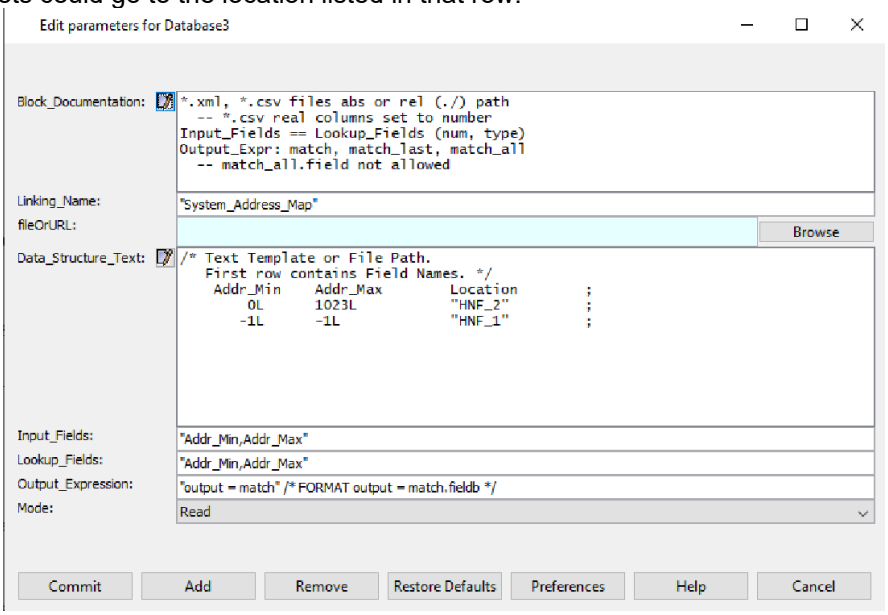
Double_bit_error_ratio
Used to specify Error ratio for Double bit Corruption.

Power_Manager_Name
Used to specify the name of the Power_Manager module. Power calculations for custom blocks are done using RegEx expressions which require using the Power_Manager name.

RNF

The RNFs are typically traffic sources that produce and pass on requests to the Home Nodes. Expression List blocks and Script blocks are used to add fields to the data structure. Some of these fields are Request Type, Request Size, Data Size, Memory Address, Priority, Source and Destination addresses. Currently four Request Types are being used which are, ReadNoSnp, ReadShared, WriteNoSnp, WriteSnp, ReadNotSharedDirty, ReadClean, and CleanUnique. Each request types are handled differently by the Home Node. The input port of the RNF receives responses, acknowledgements, and data from the home nodes.

The destination of the packet is obtained by reading a database called System_Address_Map. The -1,-1 entry means if any packet having address not listed in the database comes in, then those packets could go to the location listed in that row.



QoS Regulator Options has been added. User can choose one of the 3 options.

1. None
2. Latency Regulation
3. Bandwidth Regulation

Packets are fragmented according to Flit Size, which is a global parameter.

Parameters:

Frequency
Used to specify the frequency. Delay time between packet are calculated based on this.

Data_Bytes_Low
Minimum Size of the generated packet. This is before the NIU and flit fragmentation

Data_Bytes_High
Maximum Size of the generated packet. This is before the NIU and flit fragmentation

Source_Address
Name of this RNF

VLAN_Q
Max number of Virtual Channel available

Device_Threshold
Size of the buffer to store requests and data before sending to the network.

TrafficRate
Time between the generated packets

Address_Low
Used for Random address generation. Lower address value

Address_High
Used for Random address generation. Highest address value

Random_Address
Boolean for random number generation. True is for random numbers.False for sequential

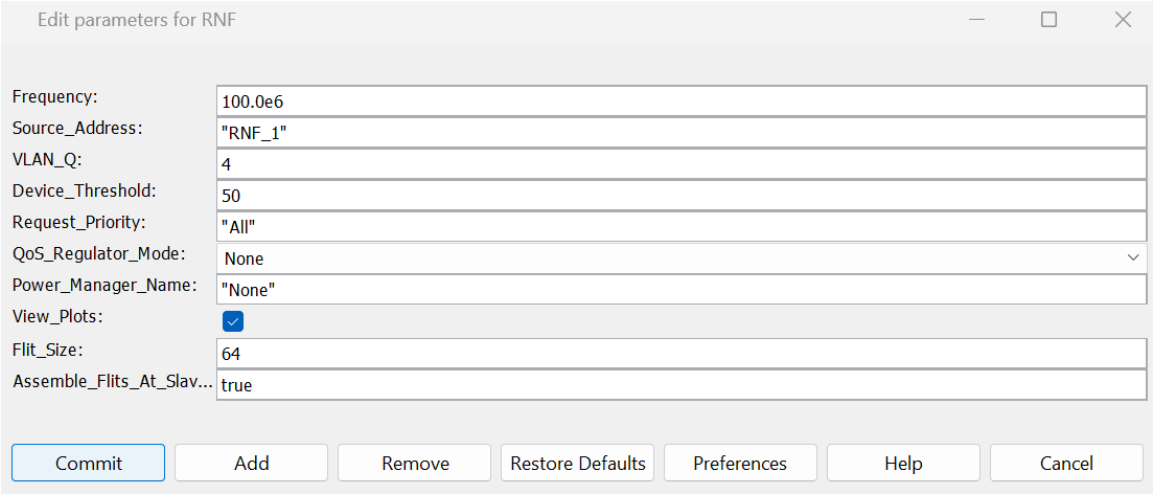
Request_Priority
There are four QoS Classes. if you want a mix of all 4, then set to "All". If a specific priority is requested then the value here is "HH", "H", "M" and "L"

QoS_Regulator_Mode
Pull down parameter. User can select "none","Latency" or "Bandwidth" regulation modes.

Power_Manager_Name
Used to specify the name of the Power_Manager module. Power calculations for custom blocks are done using RegEx expressions which require using the Power_Manager name.

View_Plots
True for viewing the plots.

Flit_Size
Used to specify the maximum packet size which can be accepted by the network.

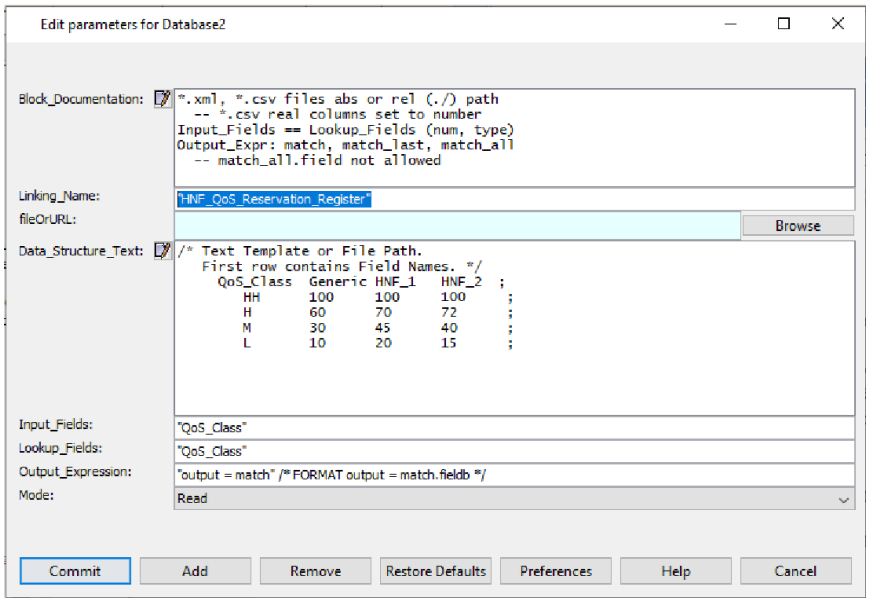


HNF:

HNFs act as smart memory interface blocks within the NoC. They receive and processes the requests received from the Request nodes. The HNF stores all the incoming requests in queues depending on their QoS class. It then uses a popping mechanism based on an algorithm that uses wait times and round robin arbitration on QoS classes to pop the right queue at the right time. The HNF also updates the database when a new request comes in to facilitate snoops. When the requests come out of the queue, a script is used to make the Home Node react differently to each type of requests.

The POCQ queue has been designed as per the specification.

For defining the number of resources available to POCQ, we have a parameter defined for HNF called Num_Resources. We also have a database at the top level called "HNF_QoS_Reservation_Register". The contents of the database look like the following



The values written under each column are in percentages. If the network is having many HNI modules, the entering values for each could be time consuming. In that case user could opt for a general configuration in which every HNF module follows. Under Generic, we have defined that HH class messages can use 100% of the resources, H class messages can use 60%, M class can use 30% and L class messages can use 10% of the total resources. We have a top level parameter called HNF_Qos_Register_Value_Selection, which is a pull down menu. User can select whether they want generic approach or not from there.

Parameters:

Frequency

Speed of processing for this device

Cache_Slice_Name
USed to specify the name of the cache slice.

Num_Resources
Used to specify the total number of POCQ resources available.

Source_Address
Name of this RNF

VLAN_Q
Max number of Virtual Channel available

Device_Threshold
Size of the buffer to store requests and data before sending to the network.

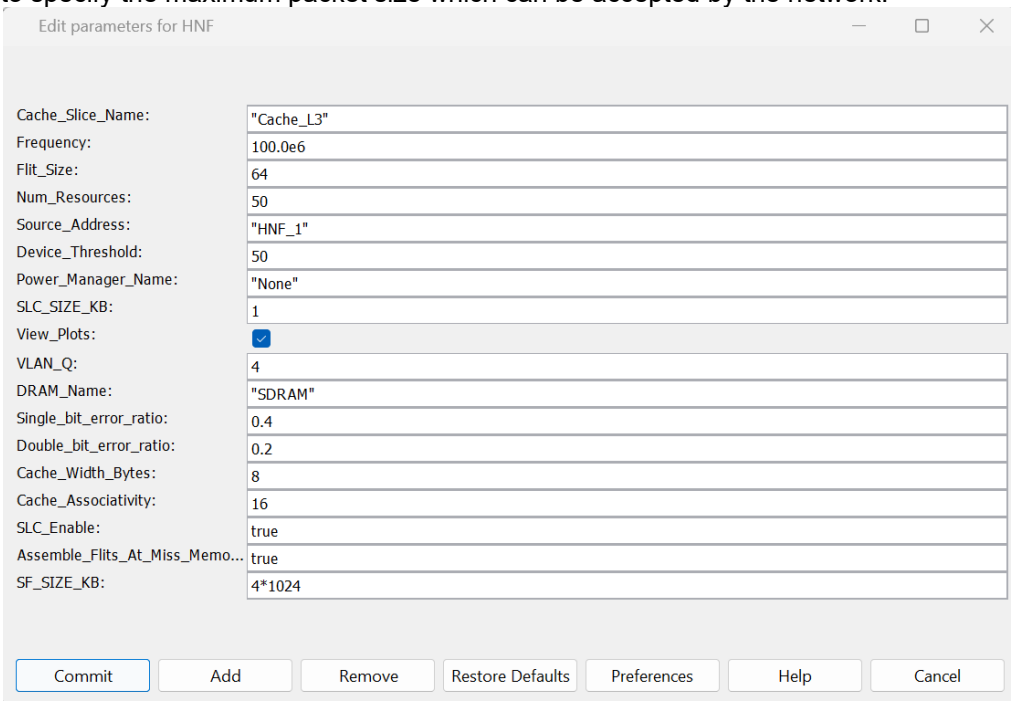
SLC_SIZE_KB
Used to specify the size of cache slice.

DRAM_Name
Used to specify the Main memory name.

Power_Manager_Name
Used to specify the name of the Power_Manager module. Power calculations for custom blocks are done using RegEx expressions which require using the Power_Manager name.

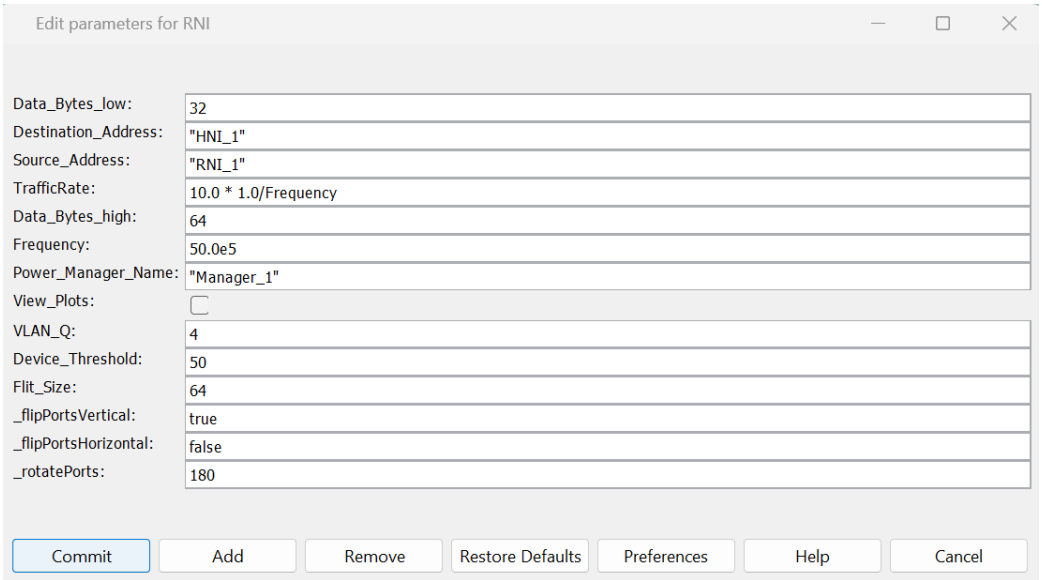
View_Plots
True for viewing the plots.

Flit_Size
Used to specify the maximum packet size which can be accepted by the network.



RNI

I/O request nodes are simple traffic generators that produce data packets to send them to a I/O Home Node. This block is the Reqest Node Interface implementation for the ARM AMBA CHI5 IP for Corelink CMN600. This block sends requests to an IO or Interface. This block does not send request to memory. Also it does not support Snoop request.



Parameters:

Frequency
Used to set the time between packet.

Data_Bytes_Low
Minimum Size of the generated packet. This is before the NIU and flit fragmentation

Data_Bytes_High
Maximum Size of the generated packet. This is before the NIU and flit fragmentation

Source_Address
Name of this RNF

VLAN_Q
Max number of Virtual Channel available

Device_Threshold
Size of the buffer to store requests and data before sending to the network.

TrafficRate
Time between the generated packets

Destination Address
Address of the HNI block. Required for Routing path

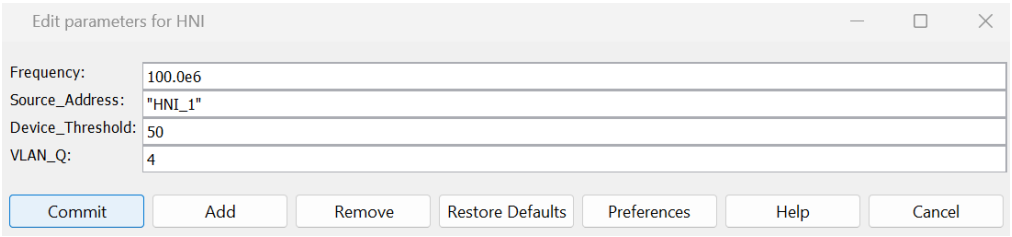
Power_Manager_Name
Used to specify the name of the Power_Manager module. Power calculations for custom blocks are done using RegEx expressions which require using the Power_Manager name.

View_Plots
True for viewing the plots.

Flit_Size
Used to specify the maximum packet size which can be accepted by the network.

HNI

I/O home nodes are sinks where the data received from the RNIs and RNFs are sent through the I/O device. This is a model of the Home Node- Interface. This simulates a sink and does not access memories or performance any snoop operation. Theblock conenects to the XP.



Parameters:

Frequency
Speed of processing the buffers

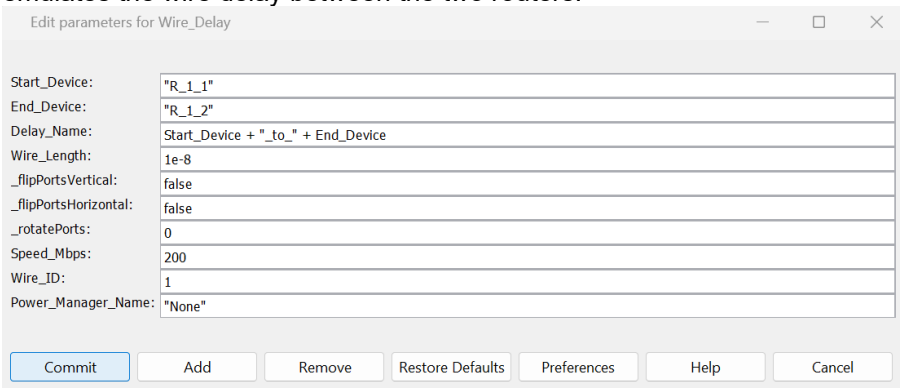
Source_Address
Name of this RNF

VLAN_Q
Max number of Virtual Channel available

Device_Threshold
Buffer Depth

Wire:

This block emulates the wire delay between the two routers.



Parameters:

Start_Device
Used to specify the XP name from which input is generated.

End_Device
Used to specify the XP name to which packet is sent.

Delay_Name
Unique name. No need to modify this paramater value.

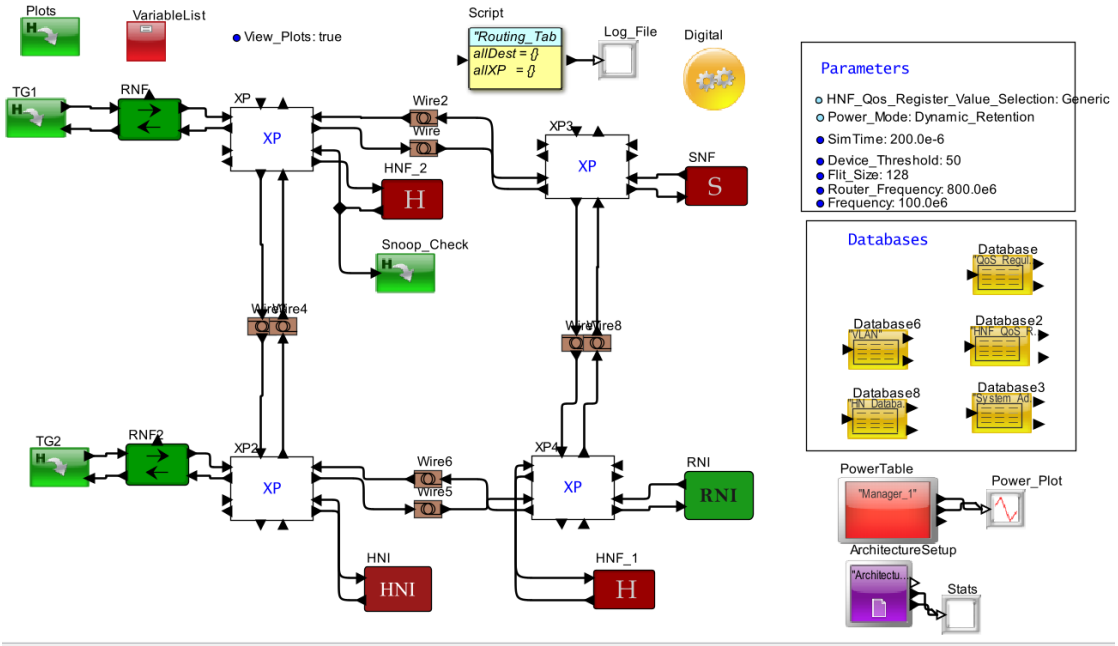
Wire_Length
Used to specify the wire length.

Speed_Mbps
Speed at which the data transfer happens between two routers

Wire_ID
Provide a unique wire ID. Each wire module should have a unique wire ID.

Power_Manager_Name
Power table name to enable power analysis.

The following image shows the 2x2 mesh architecture desinged using CoreLink CMN600 libraries.

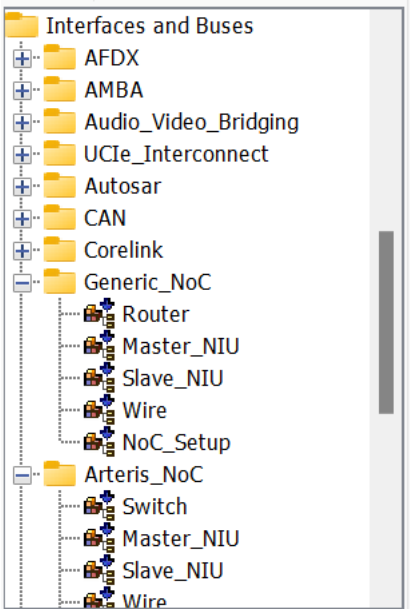


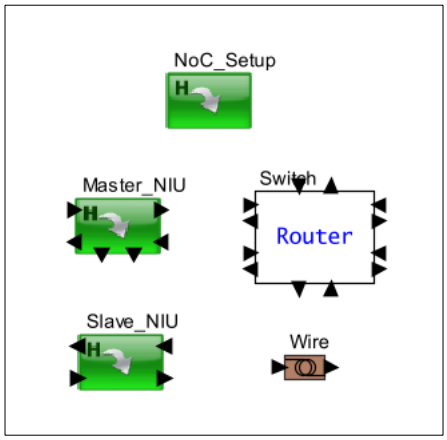
Arteris NoC

Block Description

Arteris NoC is designed to support the increasing complexity of modern SoCs, which often include hundreds of processing elements. It provides efficient communication between different devices and improves the overall performance. Its scalable feature allows users to design different SoC configurations. It supports high-bandwidth data transfer and low-latency communication. The QoS features allow users to prioritize packets and allocate bandwidth according to the application.

The key components of Arteris NoC include the Switch (Crossbar), Master NIU, and Slave NIU. The Processing elements and control devices interact with the network through the Master NIU, which performs Protocol conversion, packetization, serialization, fragmenting, and flow control. Similarly, the slave devices such as memory interact with the network through slave NIU. The Switch can be connected in different topologies such as mesh or loop and it handles clock synchronization, routing, data transfer between different ports through the crossbar, and QoS monitoring.





Features:

1. Scalable Network
2. Quality of service on bandwidth usage at each port
3. Packet prioritization
4. High bandwidth data transfer
5. Clock boundary synchronization
6. Virtual cut-through routing is used between switches
7. Store and forward routing is used at clock crossing

Switch:

The switch block acts as a crossbar to interconnect adjacent routers and the devices. The current version of the switch uses 2 dedicated ports for the devices which will be connected using either master NIU or Slave NIU. The adjacent ports can be connected through the directed in and out ports (NORTH, EAST, SOUTH, WEST).

The switch follows 4 stage pipelined operation for packets:

1. Routing computation
2. Virtual channel allocation
3. Switch allocation
4. Switch transfer

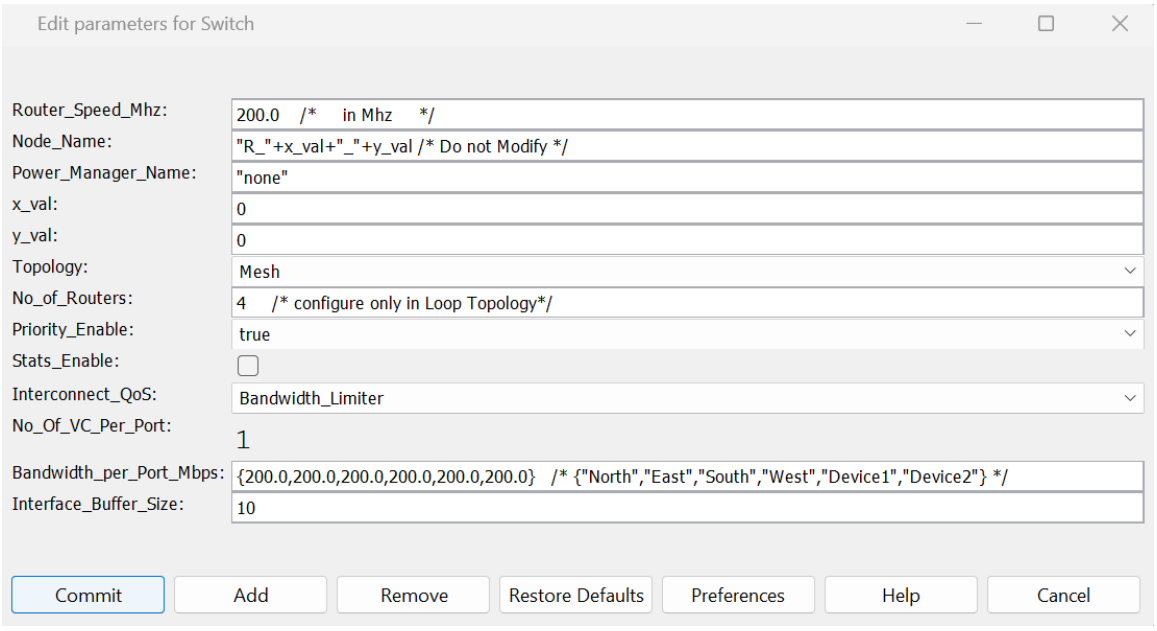
Switch operation on device ports:

The routing computations are performed for the head flit. The body flit and tail flits are bypassed to the next stage. Once the routing is computed, the packets are loaded into the input queue. The virtual channel allocation will enquire the downstreat router/switch based on the routing path. Once the virutal channel is allocated in the downstream router a reuqest will be send to the switch crossbar for arbitration. The switch uses round robin arbitration by default. Once the switch is allocated for the packet, the data will be transferred through the switch crossbar and diverted to the required output port.

Switch Operation on directional ports:

The packets might take multiple hops through the routers to reach the destination node. When the packets are moved from one router to another the directional ports are used to transfer the data. In that case the packets doesn't not need the routing computation. It will allocate the virtual channel in the downstream router, then once the allocation is successful the switch allocation and switch transfer will follow.

Parameters:



Router_Speed_Mhz
Speed of the router in Mhz. The crossbar data transfer use this parameter for delay claculation.

Node_Name
Used to specify the XP module name. This has to be unique when compared to other XP's.

Power_Manager_Name
Used to specify the Power Manager name.

x_val
Defines the x coordinate of the router to identify the position of the router in the network. It is used in Node_Name parameter.

y_val
Defines the y coordinate of the router to identify the position of the router in the network. It is used in Node_Name parameter.

Topology
Determines the NoC topology for router organization. This version supports Mesh and Loop topology.

No_of_Routers
This parameter defines the total number of routers in the network. It is used only in loop topology.

Priority_Enable
It enable or disable the priority reordering of queue in switch input buffer and virtual channels.

Stats_Enable
It enable or disabled the stats generation in the switch.

Interconnect_QoS
The switch maintains internal QoS for the bandwidth. It support Bandwidth limitter and bandwidth regulator. With Bandwidth limitter the it allows the packet only if the bandwidth is available, otherwise it will be block the flow. With bandwidth regulator the packets are delayed if the bandwidth is completely utilized.

No_Of_VC_Per_Port
It defines the number of virtual channels in each port of the switch.

Bandwidth_per_Port_Mbps
This parameter defines the bandwidth allocation of each ports in the switch

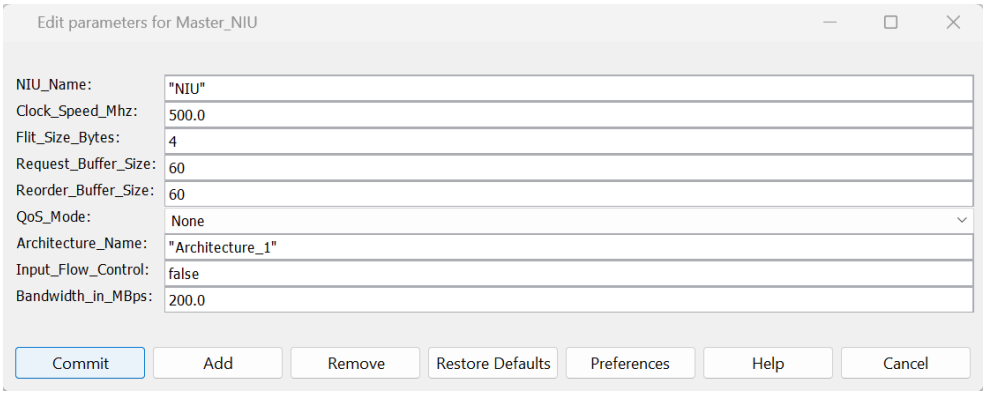
Interface_Buffer_Size
The size of the device interface buffer.

Master NIU:

The master NIU creates an interface to the switch and the device block. It buiffers the request coming from the device and add necessary fields to support the NoC protocol. Once the packets are ready to enter the network the NIU forwards the packet to Switch using a packet based flow control. On the return path the flits are reordered to buffer the entire packet. Once all the flits are recached the packet will be forwarded to the device.

Ports:

Device_in, Device_out : connects the device to the NIU
NW_In, NW_Out : connectes the NIU with the network
end_to_end_Latency : provides end to end latency for each packets
Throughput : provides the change in throughput during the simulation.



Parameter:

NIU_Name

It defines the unique name of the NIU in the network.

Clock_Speed_Mhz

It defines the speed at the which the NIU works with the device block.

Flit_Size_Bytes

It defines the size of the flits transferred in the network. The packets will be fragmented using the flit size and it should be consistent with the other NIUs in the same network.

Request_Buffer_Size

It defines the size of the request buffer that is connected to the device block.

Reorder_Buffer_Size

It defines the size of the reorder buffer which is connected a switch port, where the flits of the packets are received and buffered.

QoS_Mode

It defines the QoS type of the NIU. Supported QoS are “Fixed”, “Bandwidth_Limitter_Mode” and “Bandwidth_Regulator_Mode”.
The Fixed QoS allows packets to transfer in a fixed priority, where read is higher priority than write.
The Bandwidth limitter mode allows the request to transfer only if the request bandwidth is available, else it will be blocked.
The Bandwidth Regulator mode will allows the request to transfer only if the response bandwidth is available, else it will block the flow.

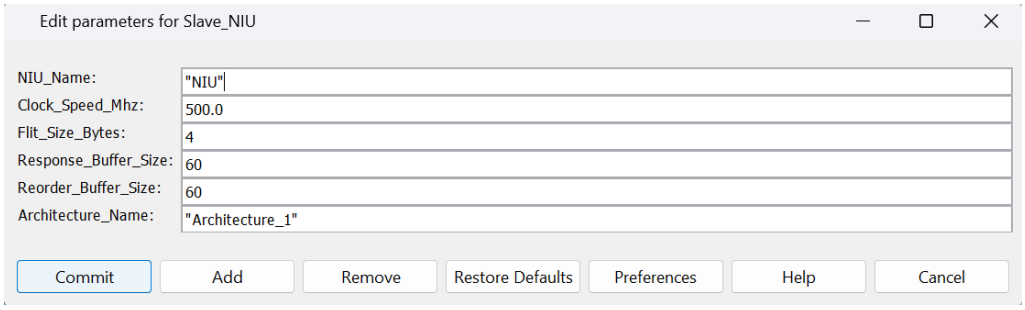
Architecture_Name
The name of the architecture setup block.

Input_Flow_Control
It enable or disbaled the flow control between the NIU and device.

Bandwidth_in_MBps
It defines the maximum bandwidth for request and response flow and it is used only the QoS is selected appropriately.

Slave NIU:
The slave NIU allows an interface to the switch and the slave device such as memory. It receive requests from the switch and buffer them, it will wait for the all the flits to be received and once it can be sent to the deivce it will forward that. On the return path the response will be forwarded to the switch using packet based flow control.

Ports:
Device_in, Device_out : connects the device to the NIU
NW_In, NW_Out : connectes the NIU with the network



Parameter:
NIU_Name
It defines the unique name of the NIU in the network.

Clock_Speed_Mhz
It defines the speed at the which the NIU works with the device block.

Flit_Size_Bytes
It defines the size of the flits transferred in the network. The packets will be fragmented using the flit size and it should be consistent with the other NIUs in the same network.

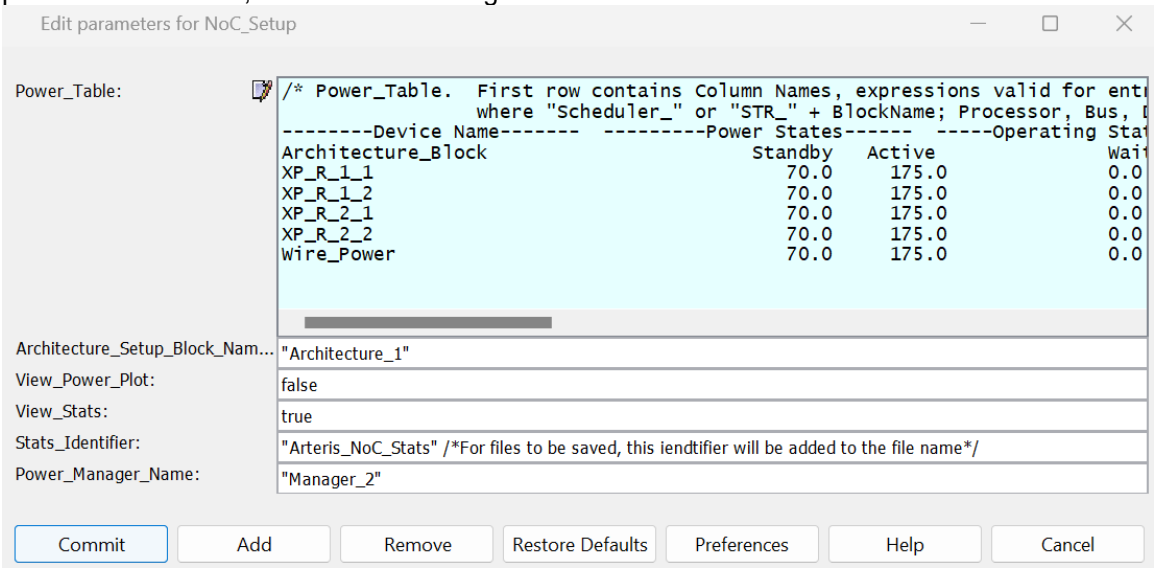
Response_Buffer_Size
It defines the size of the response buffer that is connected to the device block.

Reorder_Buffer_Size

It defines the size of the reorder buffer which is connected a switch port, where the flits of the packets are received and buffered.

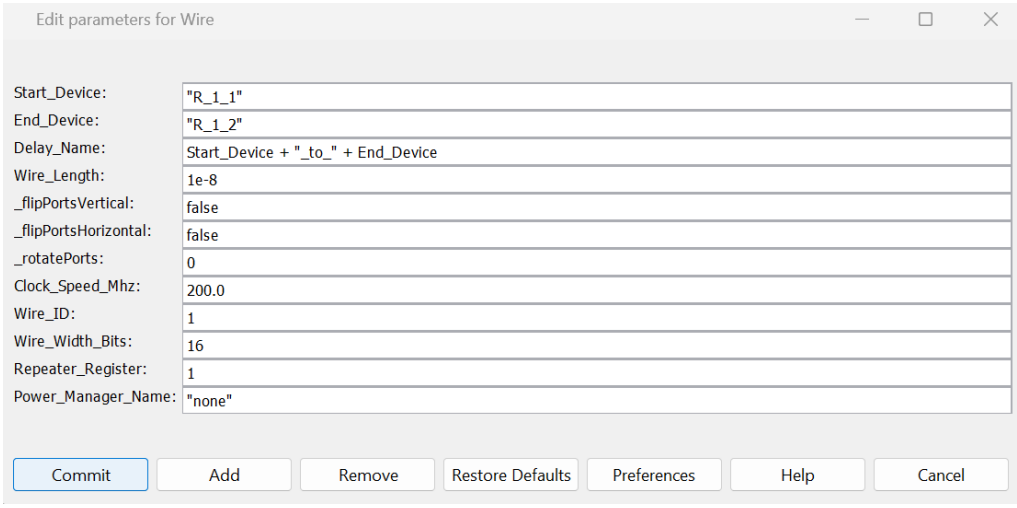
Architecture_Name
The name of the architecture setup block.

NoC Setup
The NoC Setup populates the routing table and observe the stats from the network. It should be present in the model, otherwise user will get an error.



Wire:
The wire block emulates the transaction of flits in the wire. It performs the delay according the flit size and wire length.

Ports:
Delay_in : connects to a switch directional port
Delay_out : connects to a switch directional port



Parameters:

Start_Device
Used to specify the XP name from which input is generated.

End_Device
Used to specify the XP name to which packet is sent.

Delay_Name
Unique name. No need to modify this paramater value.

Wire_Length
Used to specify the wire length.

Clock_Speed_Mhz
Speed at which the data transfer happens between two routers

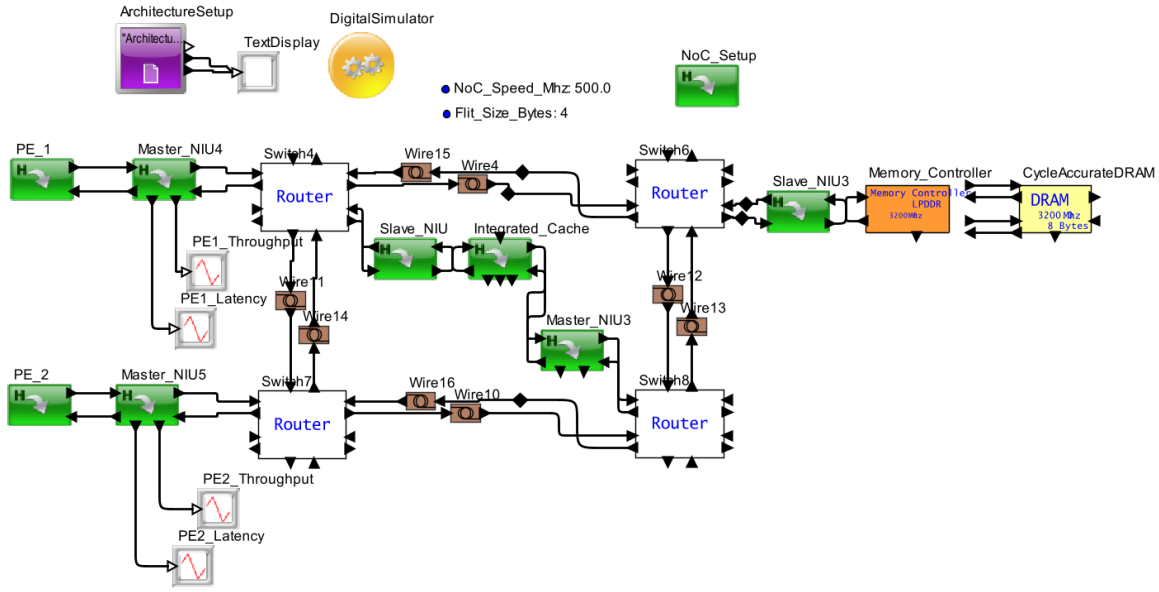
Wire_ID
Provide a unique wire ID. Each wire module should have a unique wire ID.

Power_Manager_Name
Power table name to enable power analysis.

Wire_Width_Bytes
It defines the width of the wire in bytes.

Repeater_Register
It defines the number of repeater register in that wire connection. It has an impact on wire transfer delay.

The following image shows a 2x2 mesh network with master and slave device connection.



The following are the stats that can be observed from the network.

NIU Stats:

NIU_Core_1_NIU_00_Flits_Initiated = 1278,
NIU_Core_1_NIU_01_Request_Throughput_MBps = 26.016000013008,
NIU_Core_1_NIU_02_Response_Throughput_MBps = 49.436000024718,
NIU_Core_1_NIU_03_Read_Request_Initiated = 782,
NIU_Core_1_NIU_04_Read_Response_received = 772,
NIU_Core_1_NIU_05_Write_Request_Initiated = 31,
NIU_Core_1_NIU_06_Write_Response_received = 0,
NIU_Core_1_NIU_07_Total_Read_Request_Bytes = 50048,
NIU_Core_1_NIU_08_Total_Write_Request_Bytes = 1984,
NIU_Core_1_NIU_09_Total_Request_Bytes = 52032,
NIU_Core_1_NIU_10_Total_Response_Bytes = 98872,
NIU_Core_1_NIU_11_Minimum_End_to_End_Latency = 2.787368E-6,
NIU_Core_1_NIU_12_Maximum_End_to_End_Latency = 2.7586022E-5,
NIU_Core_1_NIU_13_Request_Buffer_overflow = 0,
NIU_Core_1_NIU_14_Packets_Waiting_in_Request_Buffer = 0,
NIU_Core_1_NIU_15_Packets_Waiting_in_ROB_Buffer = 0}

Router Utilization and throughput:

{Arteris_NoC_R_1_1_Throughput_MBps = 60.4610443115234,
Arteris_NoC_R_1_1_Utilization_Pct = 98.0741310000001,
Arteris_NoC_R_1_1_to_R_2_1_Throughput_Bps = 2.3936000107712E7,
Arteris_NoC_R_1_2_Throughput_MBps = 49.3698120117188,
Arteris_NoC_R_1_2_Utilization_Pct = 97.56179055,
Arteris_NoC_R_1_2_to_R_2_2_Throughput_Bps = 4.8704000219168E7,
Arteris_NoC_R_2_1_Throughput_MBps = 34.4352722167969,
Arteris_NoC_R_2_1_Utilization_Pct = 63.8816450000002,
Arteris_NoC_R_2_1_to_R_1_1_Throughput_Bps = 1.2172000054774E7,
Arteris_NoC_R_2_2_Throughput_MBps = 49.3679046630859,
Arteris_NoC_R_2_2_Utilization_Pct = 97.55732255,
Arteris_NoC_R_2_2_to_R_1_2_Throughput_Bps = 3.062000013779E6,

DS_Name = "Arteris_NoC_Stats"}

Wire Throughput:

********** Wire_Throughput **********
R_2_1 to R_1_1 = 1.2172000054774E7 Bytes_per_sec
R_1_1 to R_2_1 = 2.3936000107712E7 Bytes_per_sec
R_2_2 to R_1_2 = 3.062000013779E6 Bytes_per_sec
R_1_2 to R_2_2 = 4.8704000219168E7 Bytes_per_sec

7 CoreConnect Bus

7.1.1 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.

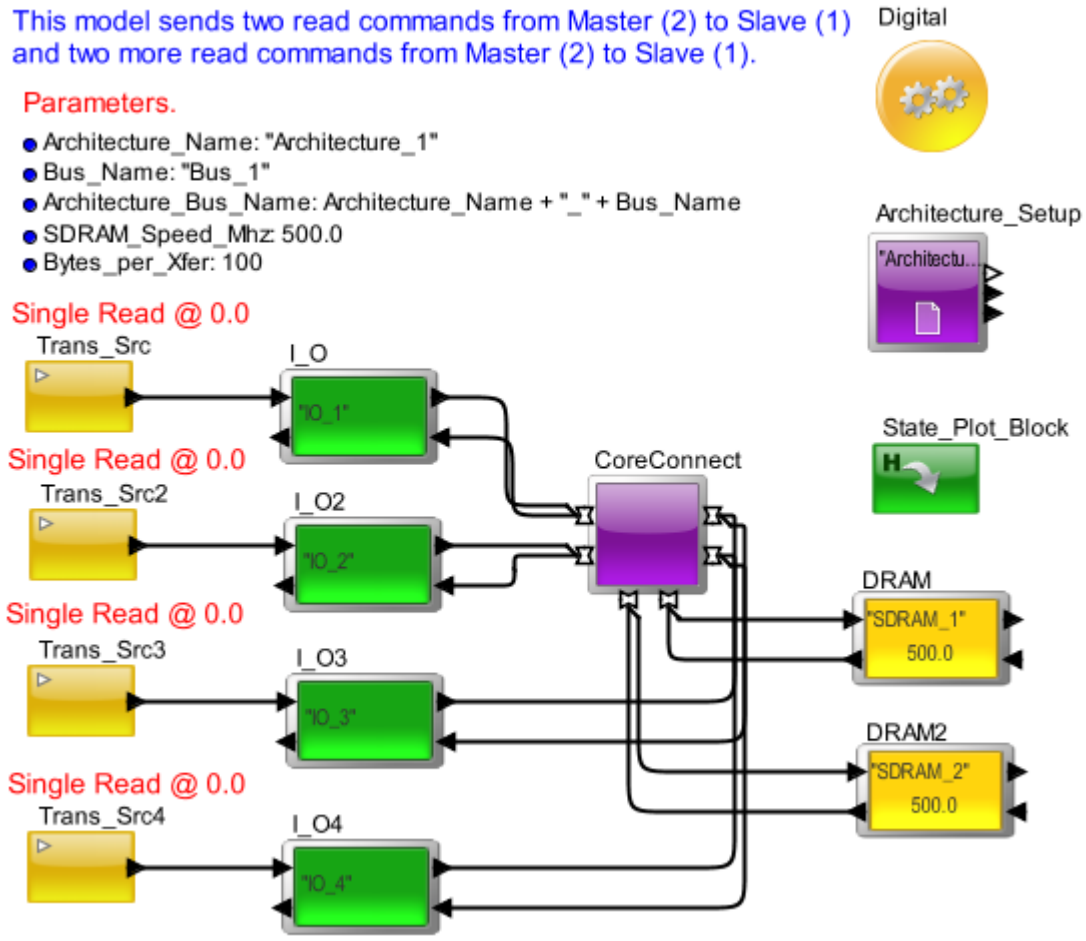


Figure 7-1 VisualSim model of a system using the CoreConnect bus

7.1.2 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.

7.1.3 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.

7.1.4 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:

CoreConnect Bus Commands	A_Command field
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”:

Data Structure Field	Data Type	Sample Data
A_Bytes	int	64
A_Command	string	“Read_Memory”

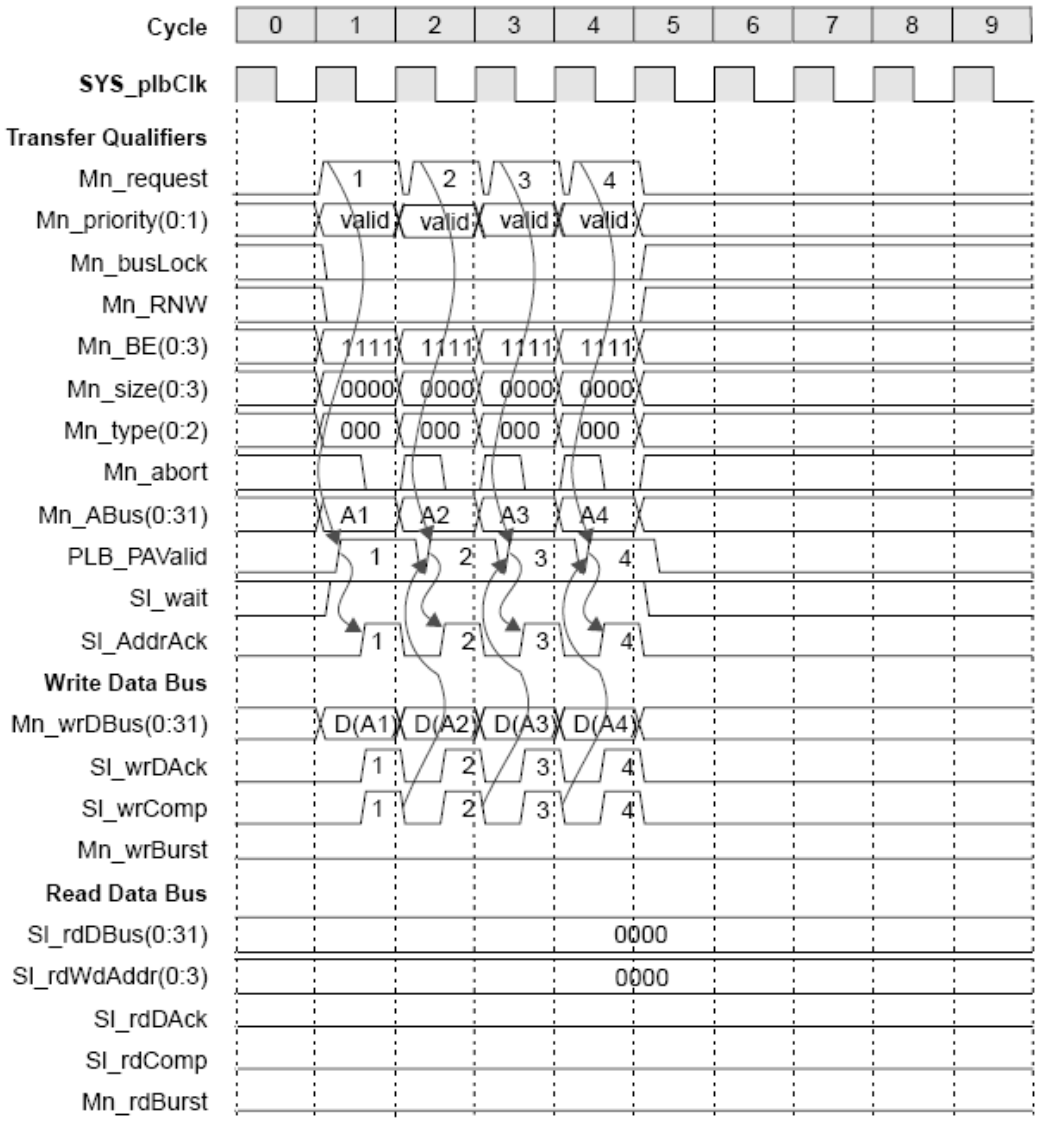
A_First_Word	boolean	true
A_Priority	int	1
A_Source	string	“IO_1”
A_Destination	string	“SDRAM_1”

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.

7.1.5 Timing Diagrams

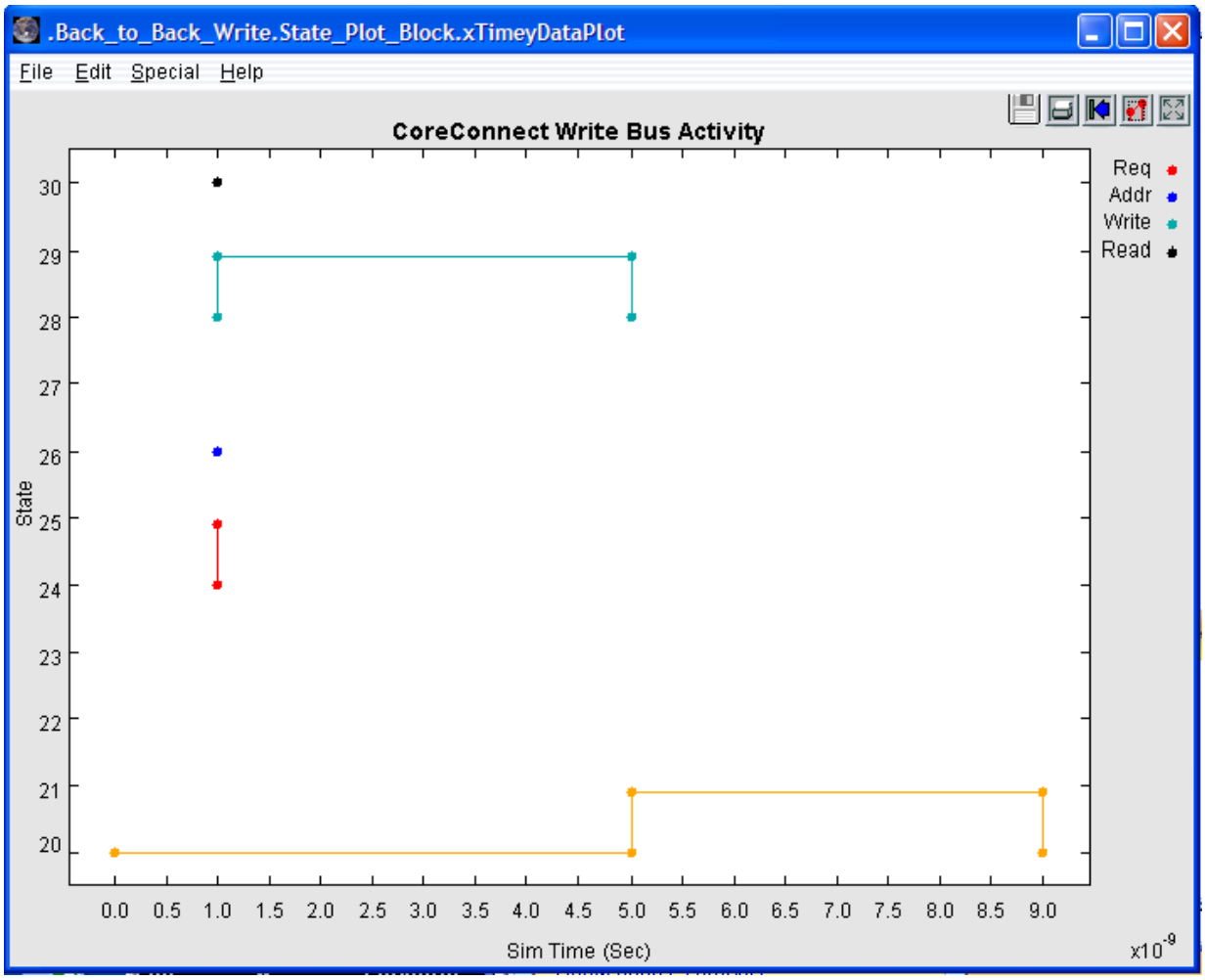
7.1.5.1 Back-to-Back Write Transfers (IBM)

IBM CoreConnect, 4 Cycles for 4 words



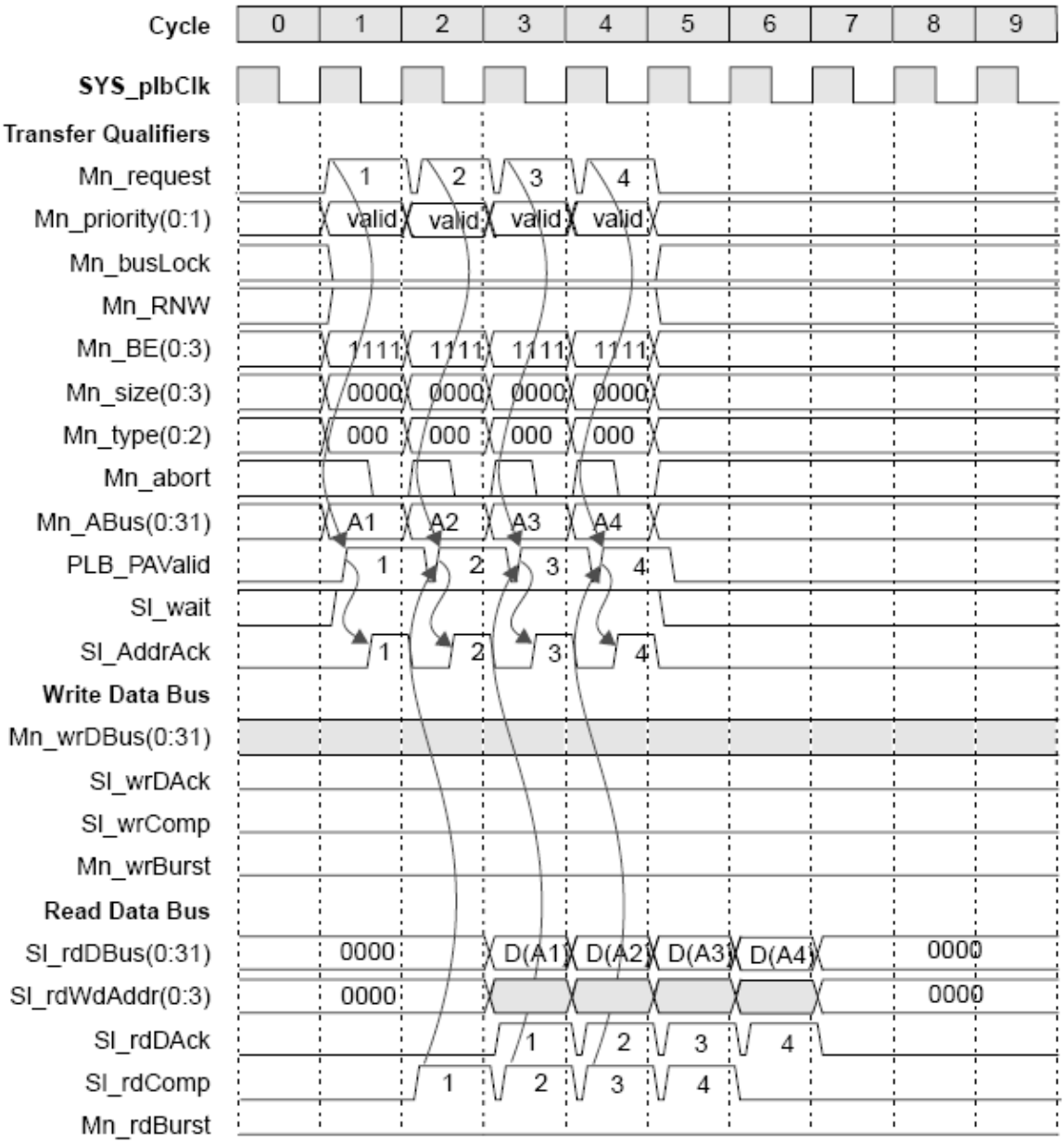
7.1.5.2 Back-to-Back Write Transfers (VisualSim)

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.



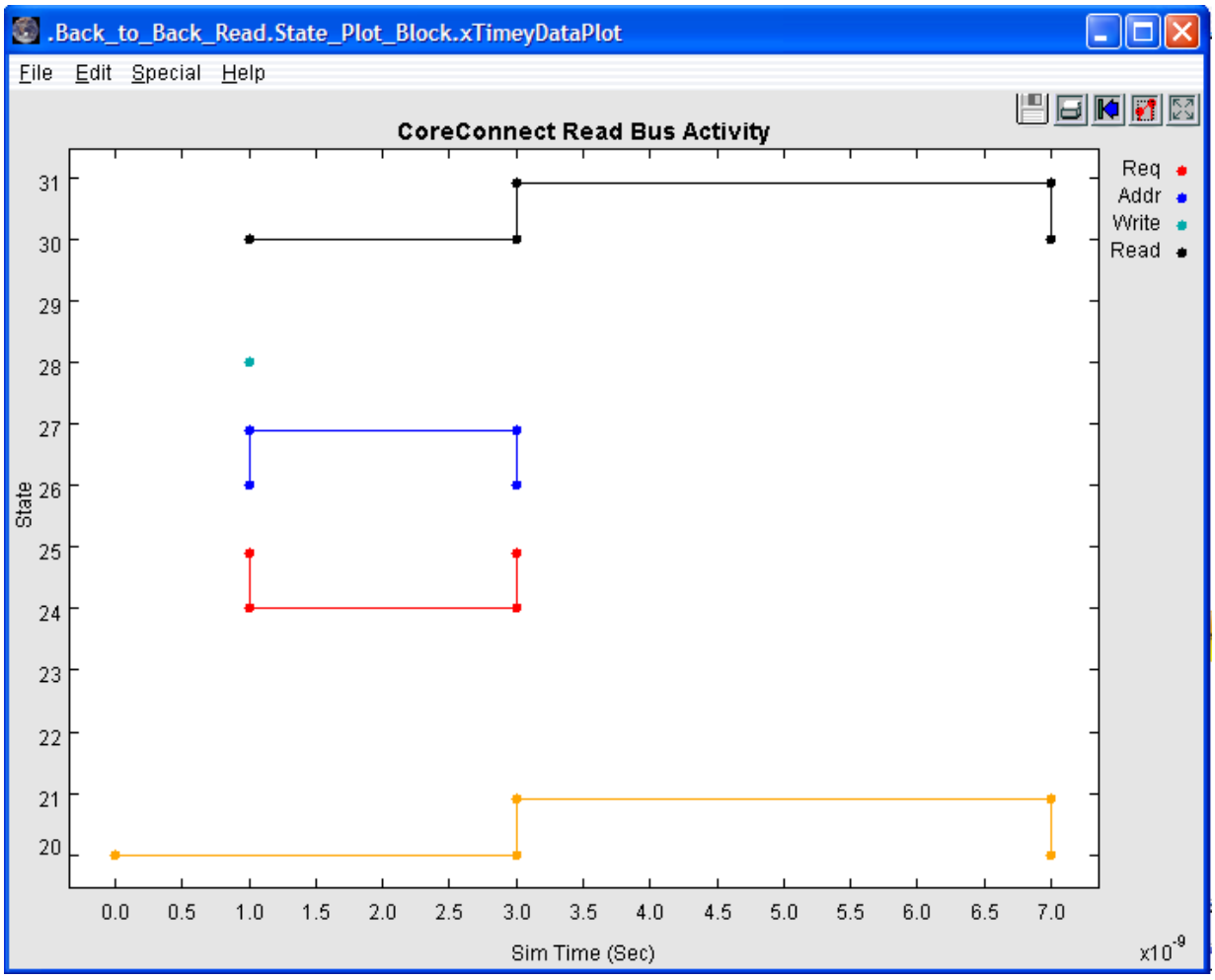
7.1.5.3 Back-to-Back Read Transfers (IBM)

IBM CoreConnect, 6 Cycles for 4 words



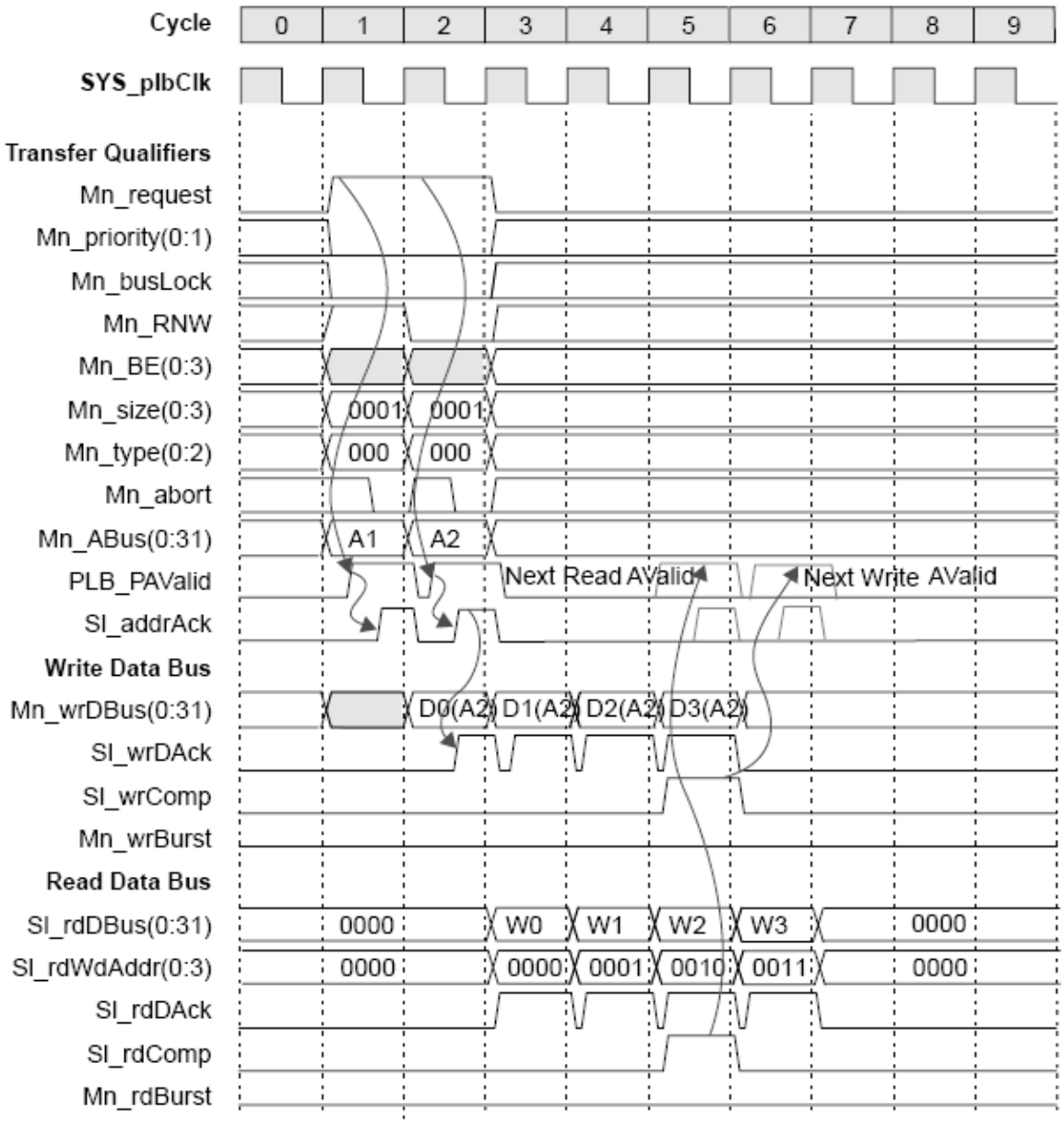
7.1.5.4 Back-to-Back Read Transfers (VisualSim)

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.



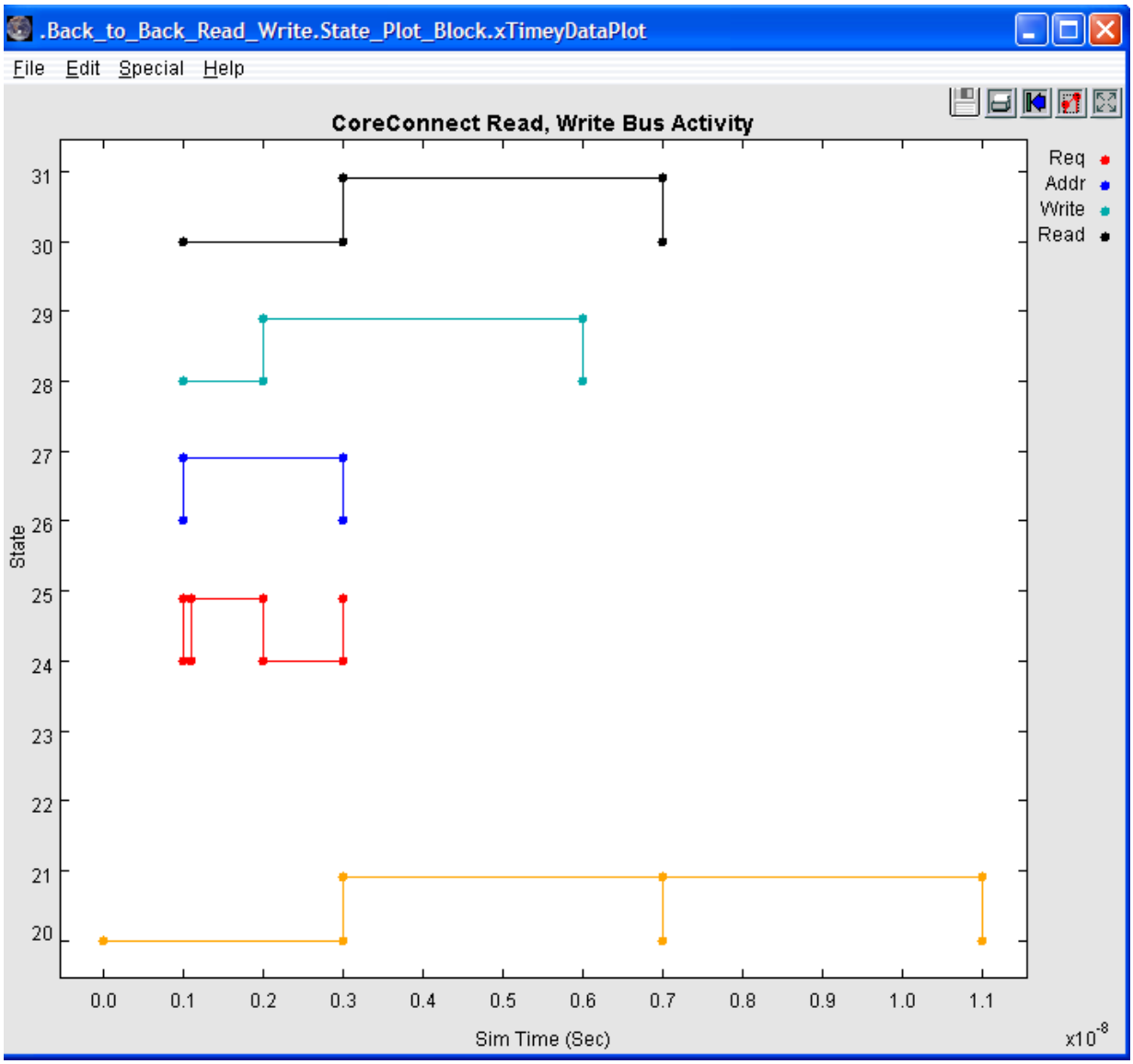
7.1.5.5 Four-word Line Read Followed By Four-word Line Write Transfers (IBM)

IBM CoreConnect, 6 Cycles for 4 word Read followed by 4 word Write



7.1.5.6 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.



7.1.5.7 Single Read Transfer (Xilinx)

Xilinx CoreConnect, 6 Cycles for 1 word

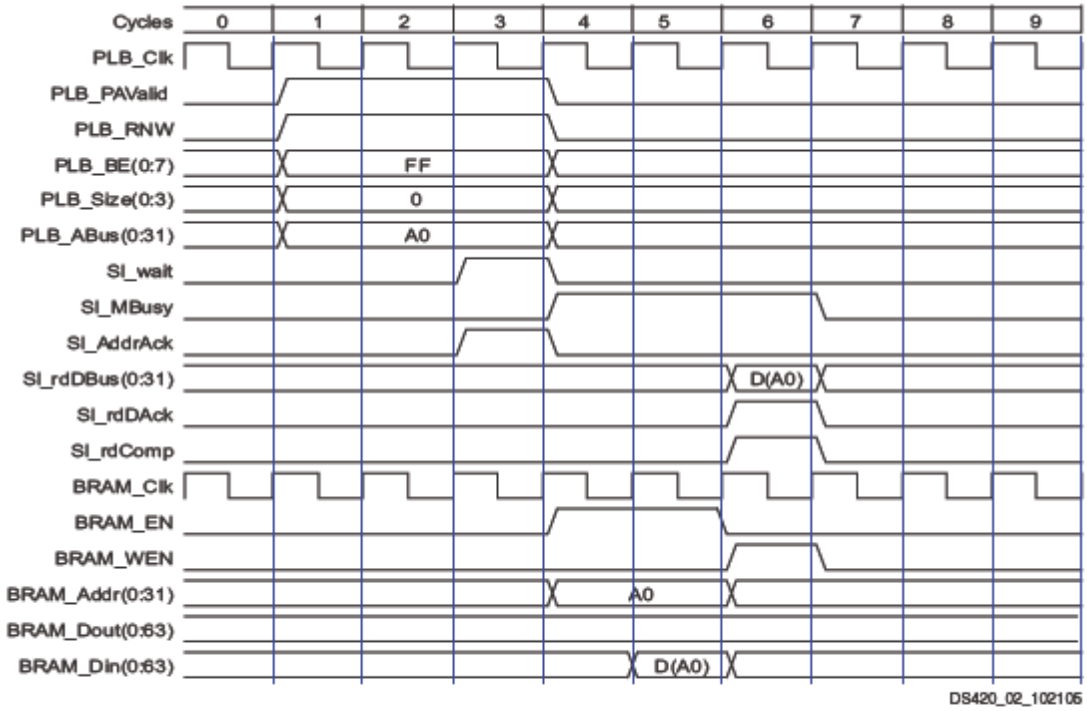
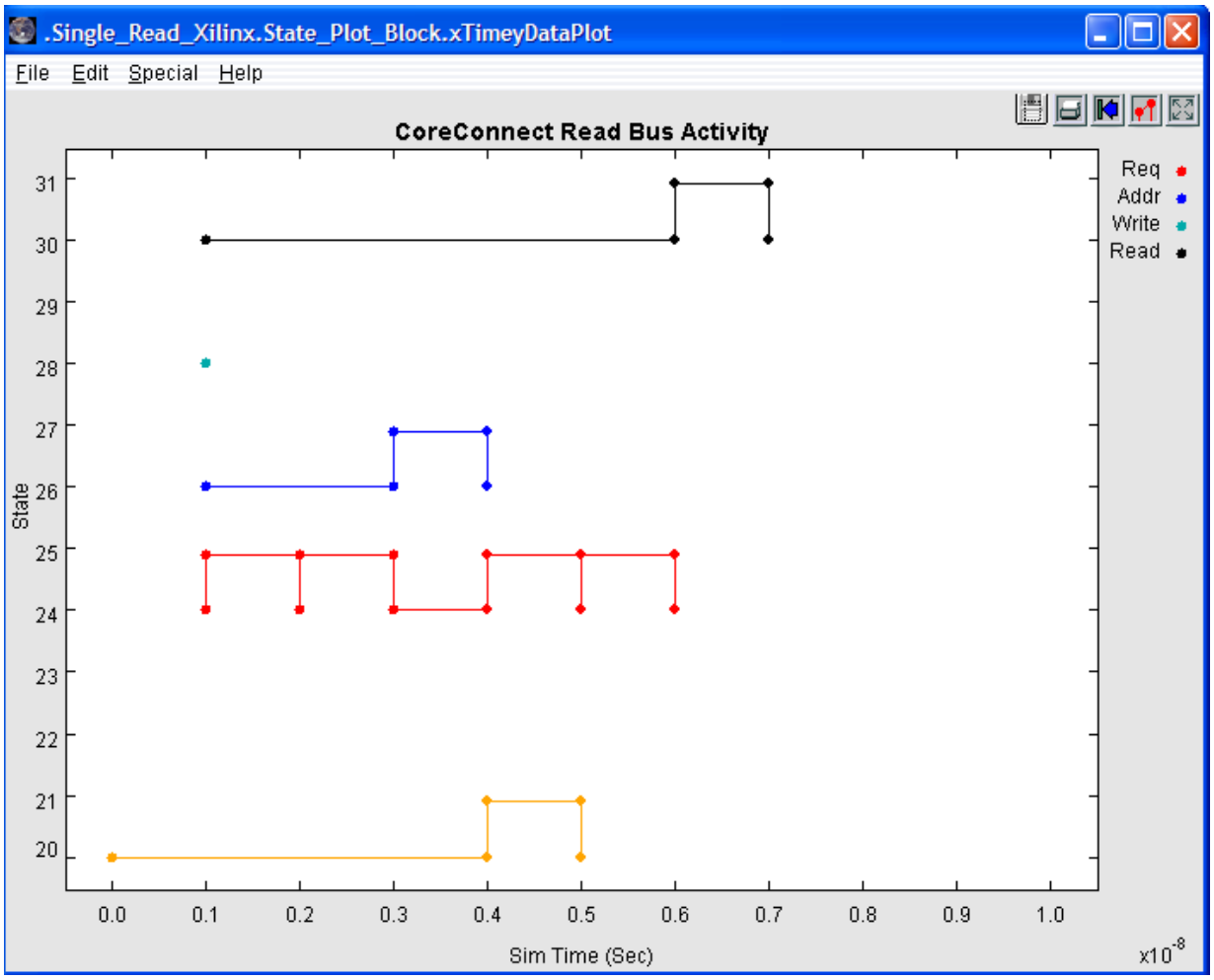


Figure 2: **Single Read Transfer**

7.1.5.8 Single Read Transfer (VisualSim)

Models Xilinx CoreConnect, 6 Cycles for 1 word
Address_Bytes = 4
Request_Clock_Multiplier = 0.99



7.1.5.9 Single Write Transfer (Xilinx)

Xilinx CoreConnect, 6 Cycles for 1 word

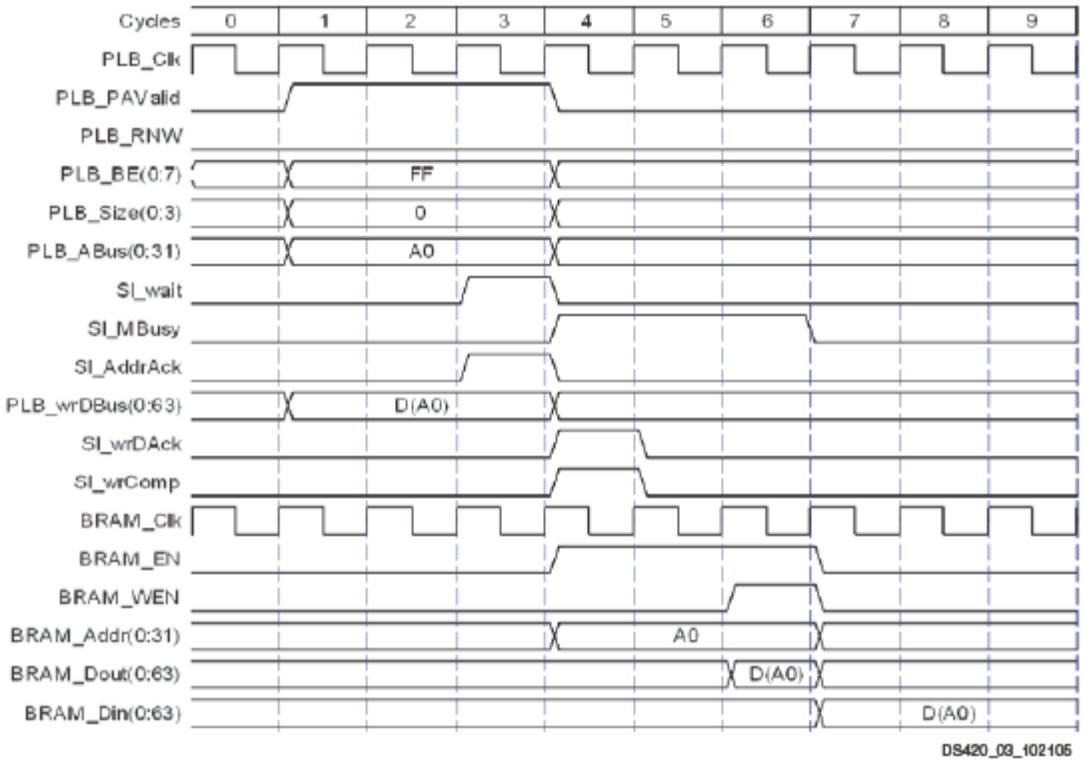
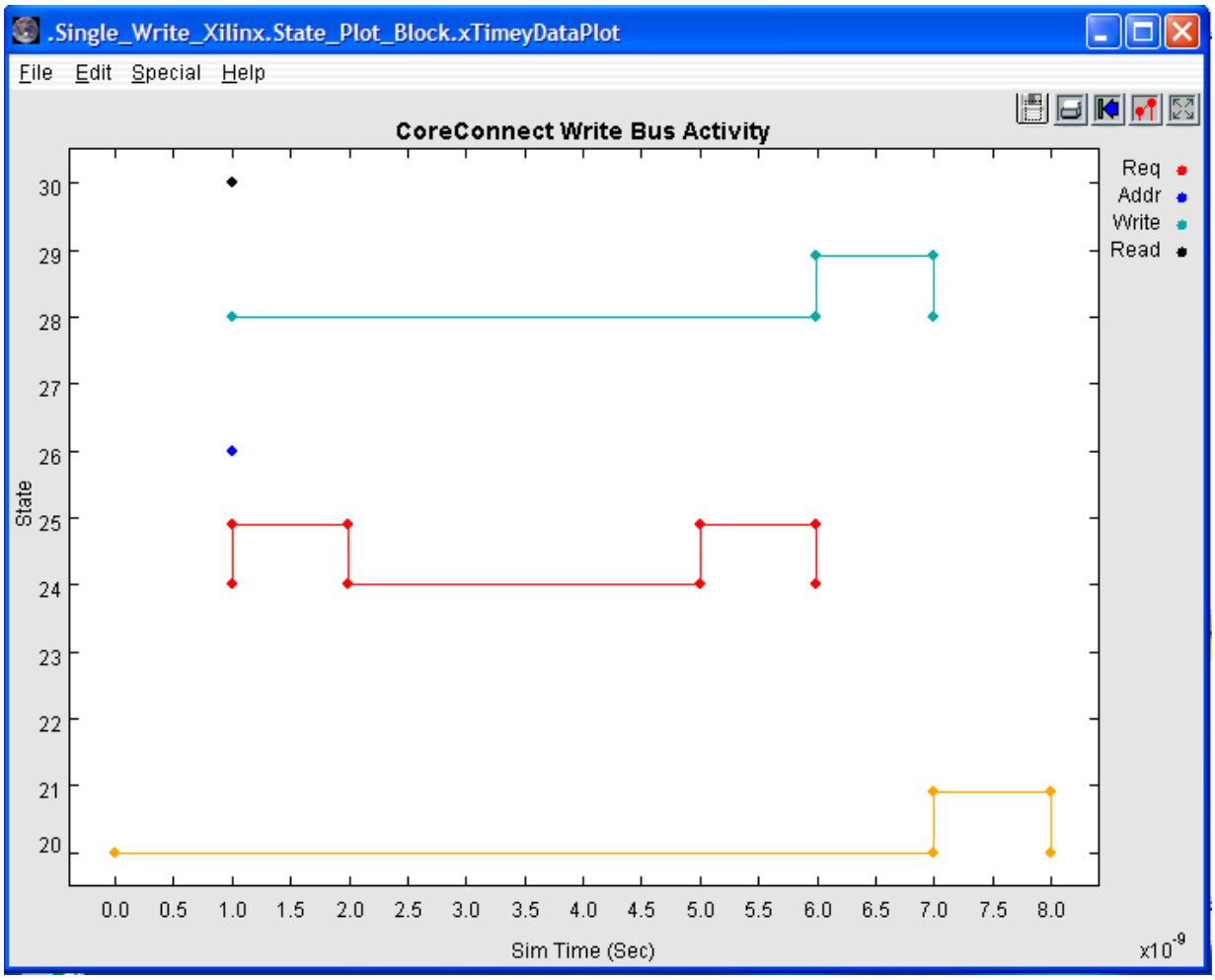


Figure 3: **Single Write Transfer**

7.1.5.10 Single Write Transfer (VisualSim)

Models Xilinx CoreConnect, 6 Cycles for 1 word
Address_Bytes = 4
Request_Clock_Multiplier = 0.99



7.1.6 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.

7.1.7 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_MBs*** – 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       = 0.3333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Mean      = 0.3333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_Min       = 0.3333333333333,
Bus_1_Node_1_Address_Buffer_Occupancy_in_Words_StDev     = 0.0,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Mean         = 0.6666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_Min  = 0.6666666666667,
Bus_1_Node_1_Read_Buffer_Occupancy_in_Words_StDev        = 0.0,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Max       = 0.6666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Mean      = 0.6666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_Min       = 0.6666666666667,
Bus_1_Node_1_Request_Buffer_Occupancy_in_Words_StDev     = 0.0,
```

Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_3_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_3_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_3_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_4_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_4_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_4_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_5_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_5_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_5_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_6_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,
Bus_1_Node_6_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_6_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_6_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_6_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_6_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_6_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_6_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_6_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_6_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_6_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_6_Request_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_7_Address_Buffer_Occupancy_in_Words_Max = 0.3333333333333,

Bus_1_Node_7_Address_Buffer_Occupancy_in_Words_Mean = 0.3333333333333,
Bus_1_Node_7_Address_Buffer_Occupancy_in_Words_Min = 0.3333333333333,
Bus_1_Node_7_Address_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
Bus_1_Node_7_Read_Buffer_Occupancy_in_Words_StDev = 0.0,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_Max = 0.6666666666667,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_Mean = 0.6666666666667,
Bus_1_Node_7_Request_Buffer_Occupancy_in_Words_Min = 0.6666666666667,
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,

7.1.8 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.

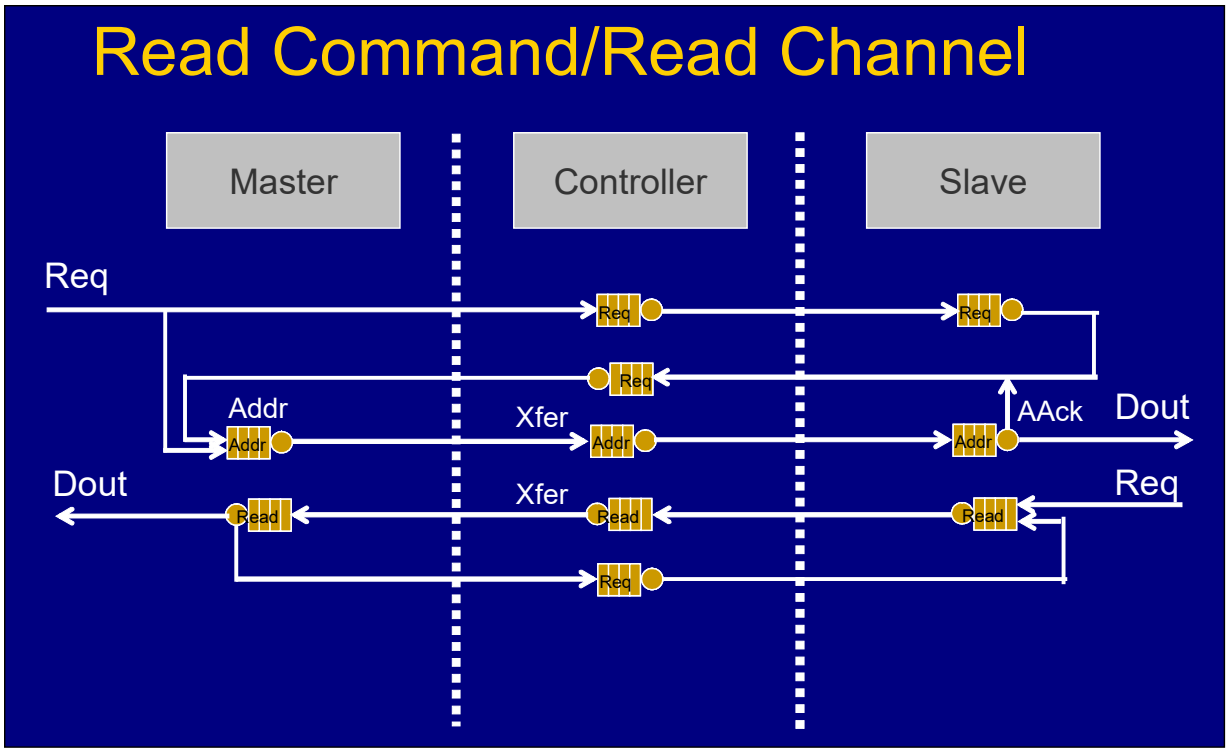


Figure 7-2 Read Command/Read Channel

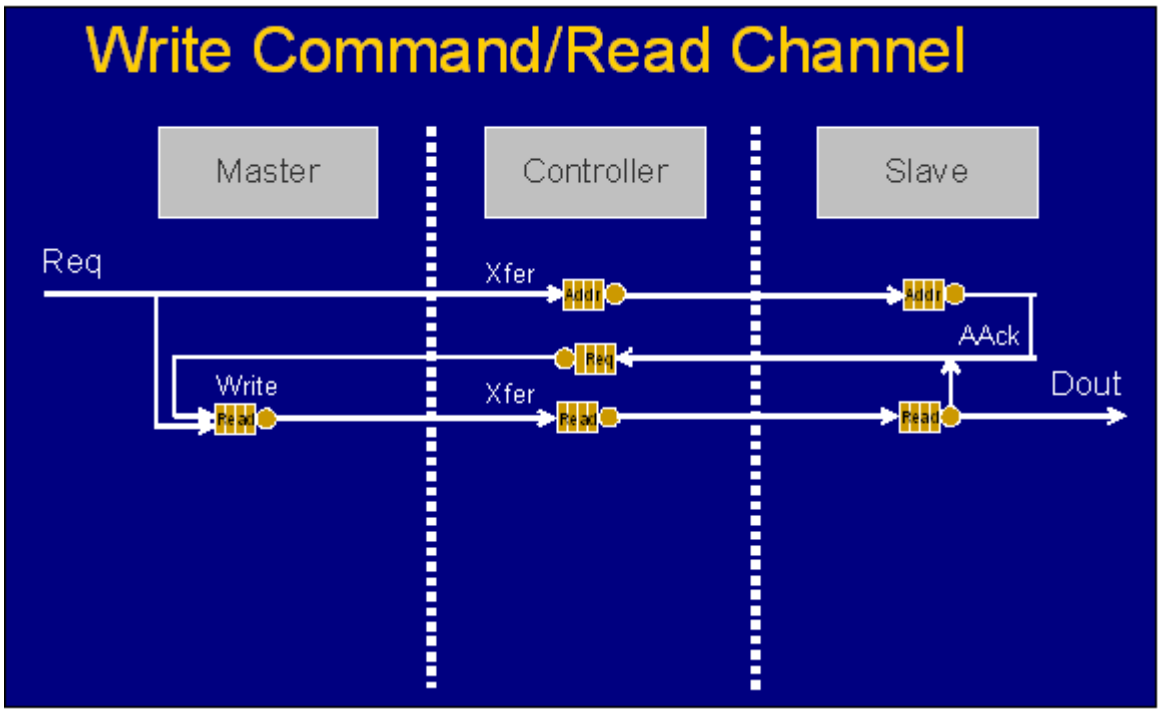


Figure 7-3 Write Channel/ Write Command

8 SpaceWire

8.1.1 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.

8.1.1.1 Sample Model

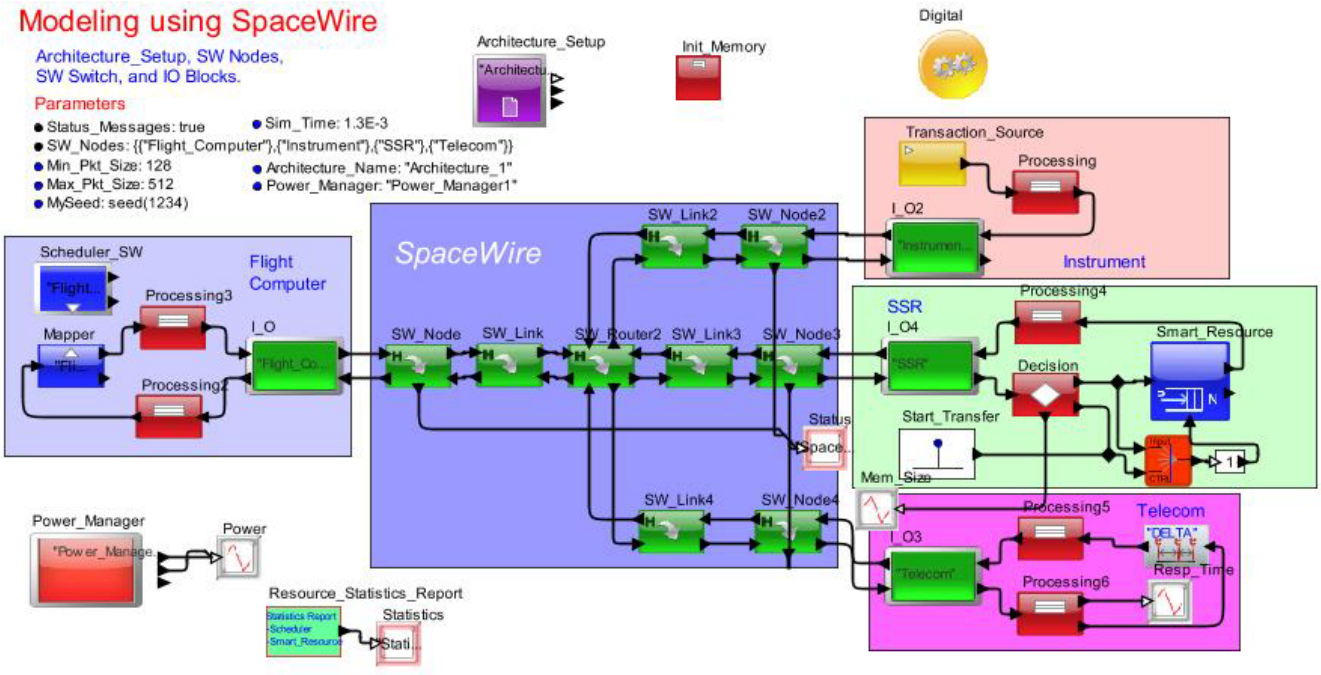


Fig.1 SpaceWire Model

8.1.2 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.

8.1.3 Model Parameter

The Parameter for a system that uses implementation of SpaceWire Bus

6. Sim_Time = 1.3E-3
7. Architecture_Name = "Architecture_1"
8. Power_Manager = "Power_Manager1"
9. Min_Pkt_Size = 1024
10. Max_Pkt_Size = 4096
11. 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 ;

8.1.3.1 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.

8.1.4 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 occurance such as Buffer Full Error, Bit Error and Timeout Error resend the Token by setting A_Packet = "EEP" and calculate the Statistics.

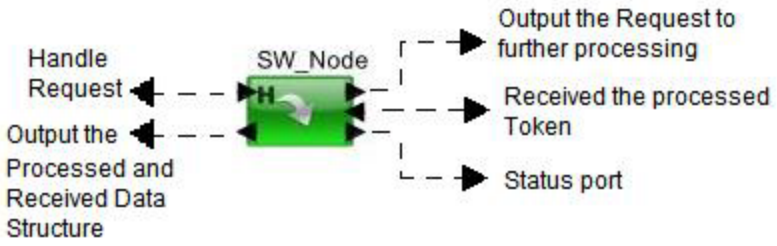


Fig. 2 SpaceWire Node Block

8.1.4.1 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.

8.1.5 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.

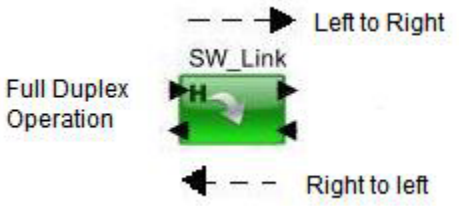


Fig. 3 SpaceWire Link Block

8.1.5.1 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

8.1.6 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.

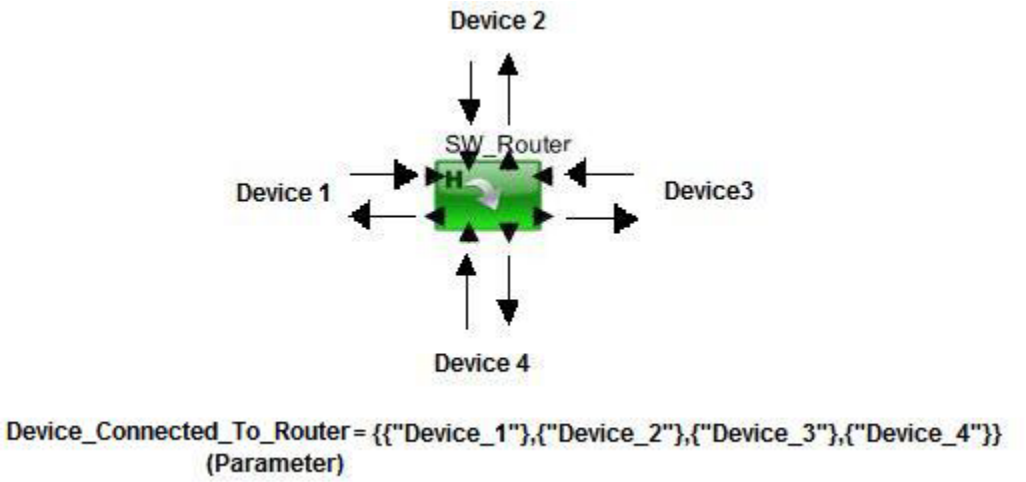
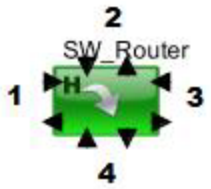


Fig.4 SpaceWire Router Block

8.1.6.1 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



8.1.6.2 Block Diagram

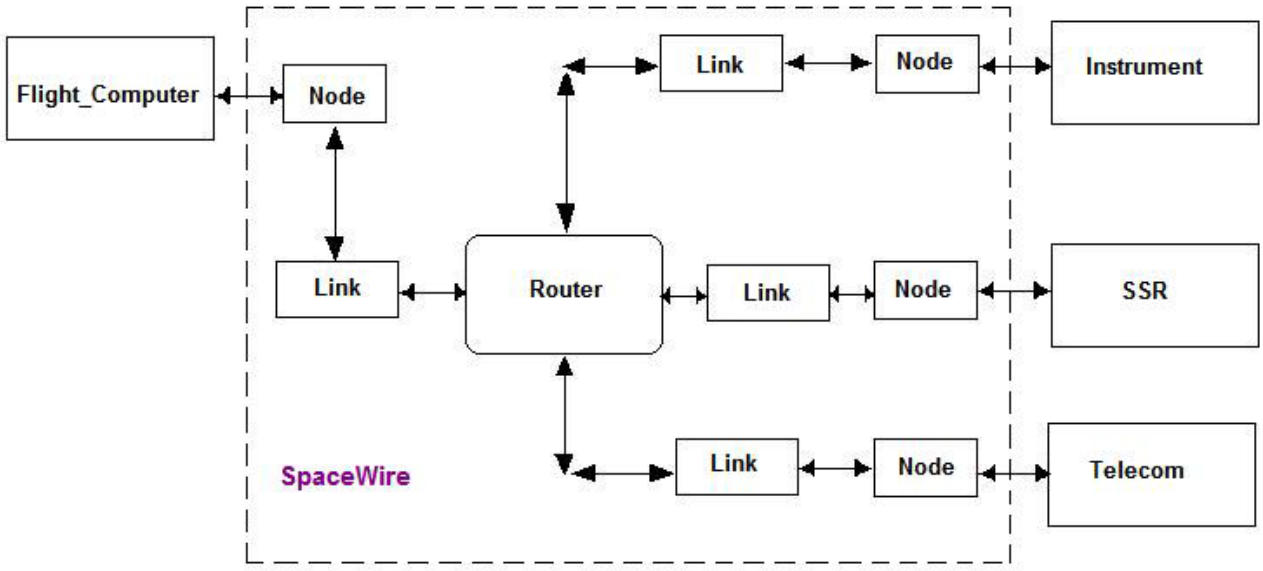


Fig.5 SpaceWire Model Block Diagram

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.

8.1.7 SpaceWire Description

The SpaceWire standard covers the following normative protocol levels

8.1.7.1 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

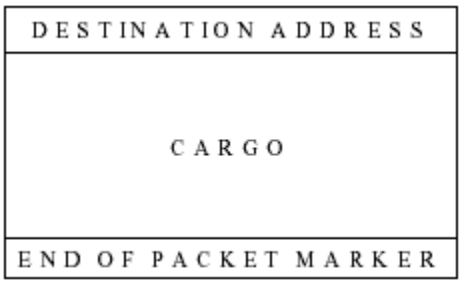


Fig.6 Packet Format

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.

8.1.7.2 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.

8.1.7.3 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.

8.1.7.4 ERROR Scheme

The error occurrence and recovery in the SpaceWire are as follows.

a) Buffer Overflow Error

The Token in the output Queue is geater 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.

8.1.8 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 = 6.9150000000000244E-6,
Total_Delay_Mean = 3.580000000000031E-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 = 7.983999999999943E-6,
Total_Delay_Mean = 4.985384615384614E-6,
Total_Delay_Min = 2.144000000000039E-6,
Total_Delay_StDev = 1.974573509489743E-6,
Utilization_Mean = 9.970769230769228

9 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 tested for various fault and error condition 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.

9.1 Library

The AVB library is available as feature in VisualSim. The AVB library is located in Interfaces and Buses → Audio_Video_Bridging.

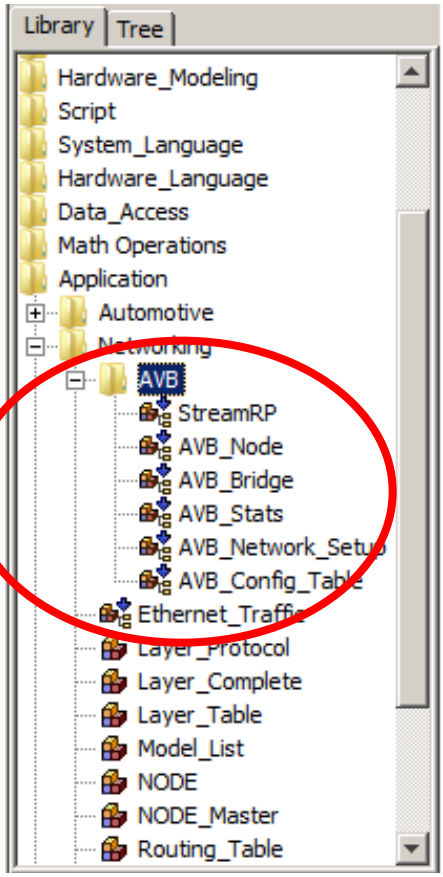


Figure 4 AVB Modeling Library

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 form 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.

7. Ethernet_Traffic- This is a traffic generator that can handle both Ethernet traffic and AVB talker request message.

9.2 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:

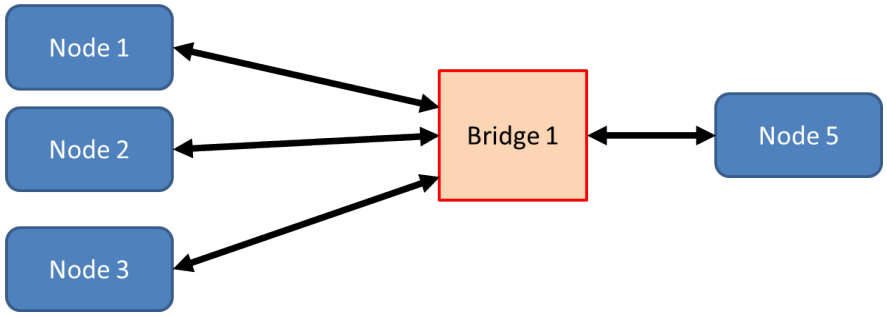
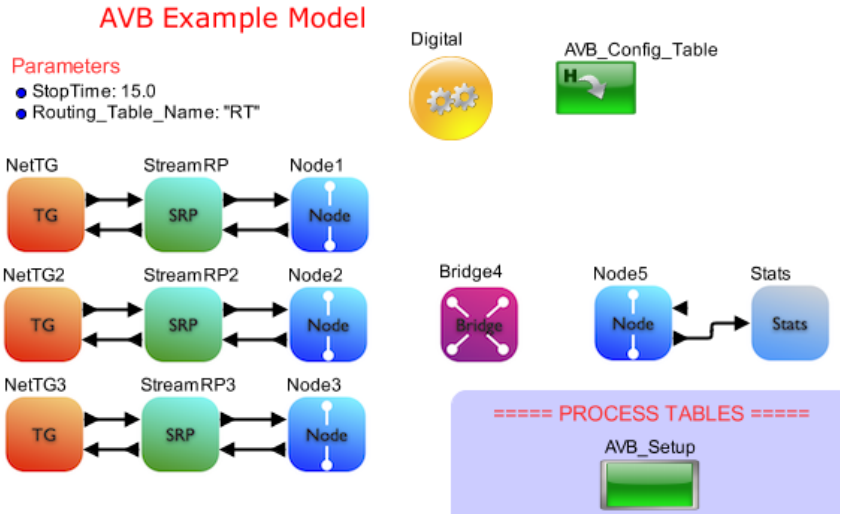


Figure 5 Block Diagram of a AVB Network

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.



10 Fibre Channel

10.1 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.

10.2 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.

10.3 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

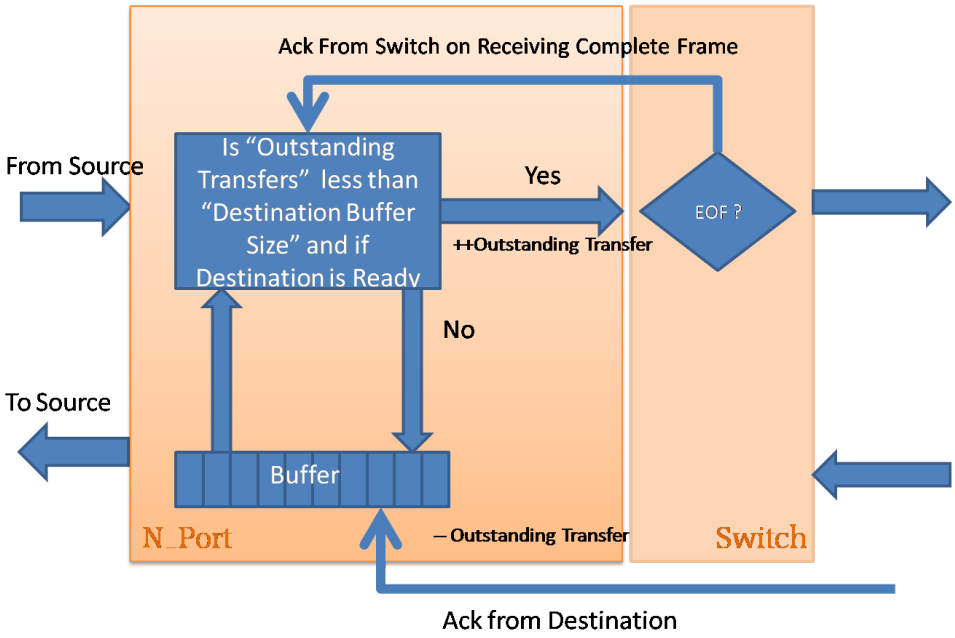
Each library components are built to Fibre Channel Specification and meets performance requirements.

10.4 FC_N_Port

FC_N_Port interfaces to the device on one side and the Fibre Chanel 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.

10.4.1 Flow Diagram



10.4.2 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 */
```

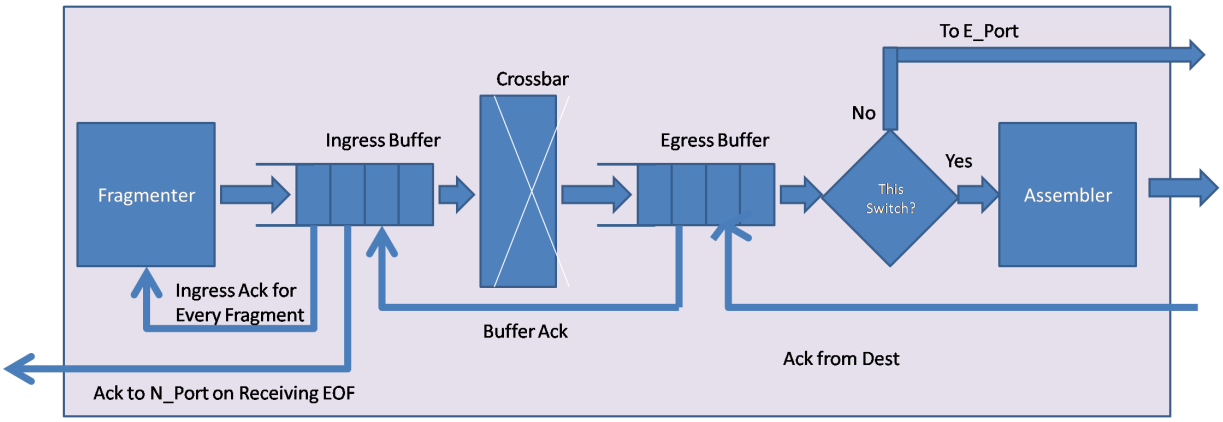
10.4.3 Parameters

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***

10.5 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 Swicthes . 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

10.5.1 Flow Diagram



10.5.2 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.

10.5.3 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.

10.5.4 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***

Egress_Buffer_Size	***Integer***	***Size of Egress Buffer at each port***
Architecture_Name	***String***	***Name of Arch_Setup***
Fragment_Size	***Integer***	***Size of Frame Fragment***
Overhead_Bytes	***Integer***	***Overhead Bytes***
Enable_Debug	***Boolean***	***Enable or Disable Debug Message***
Switch_ID	***Integer***	***Unique Identifier for Fabric Switch***

10.6 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.

10.6.1 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

10.7 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.

10.7.1 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 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 ***Switch_IDx Connected_Switches E_Port_Address ;*** ***2 {{1,3}} {{4,1}} ;*** ***1 {{2},{3}} {{1,1},{4,1}} ;*** ***3 {{1}} {{1,1}} ;***

10.8 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

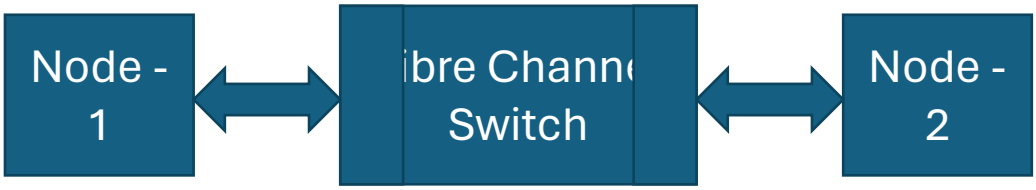


Figure 1.0: System Block Diagram

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

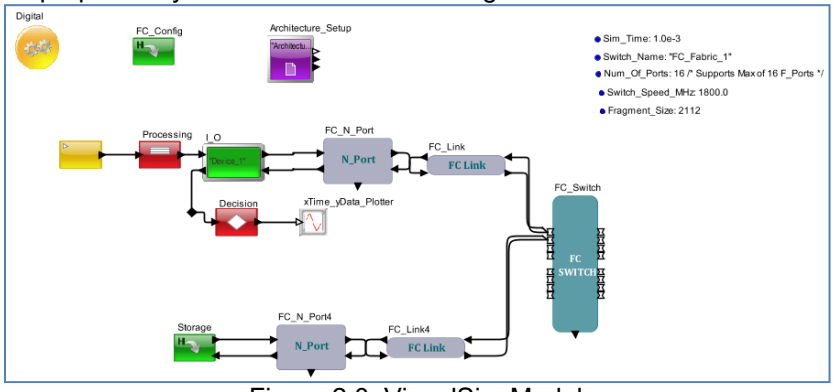


Figure 2.0: VisualSim Model

10.8.1 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

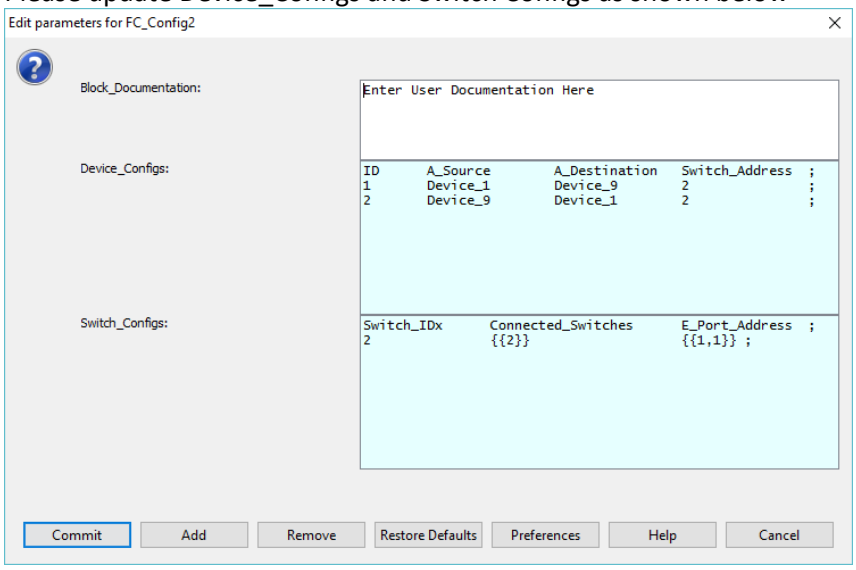
10.8.2 Construction Steps

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

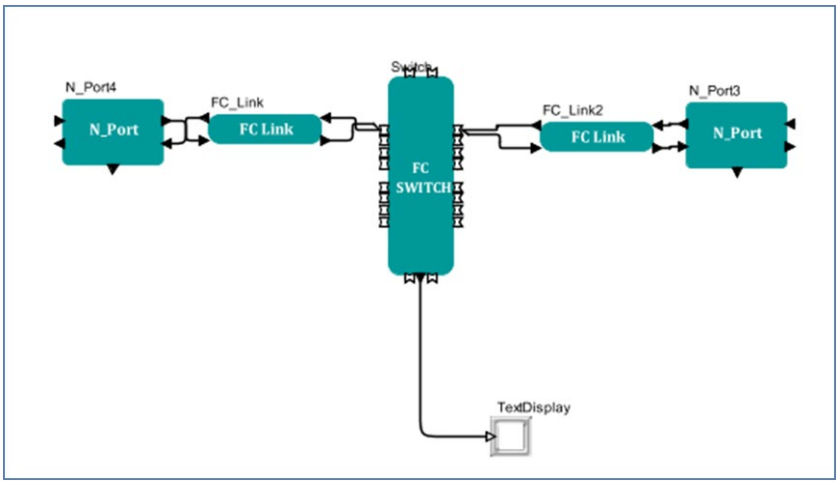
10.8.2.1 Step1

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

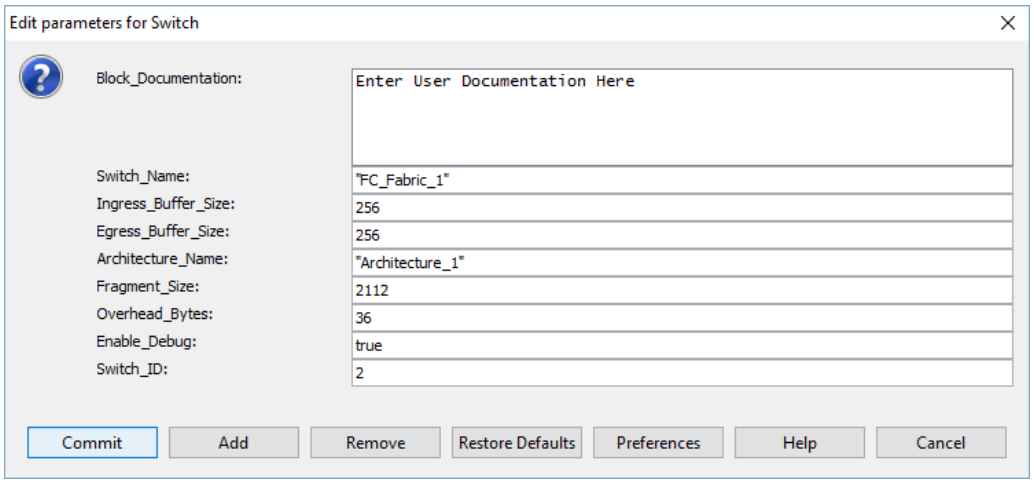


10.8.2.2 Step2

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



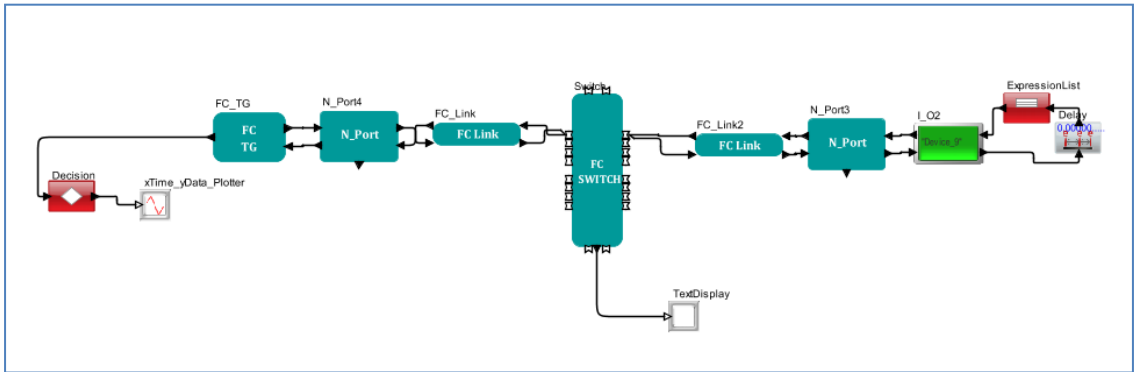
10.8.2.3 Step3

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	1.0e-3	"Write"	false	1000.0	;
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	;

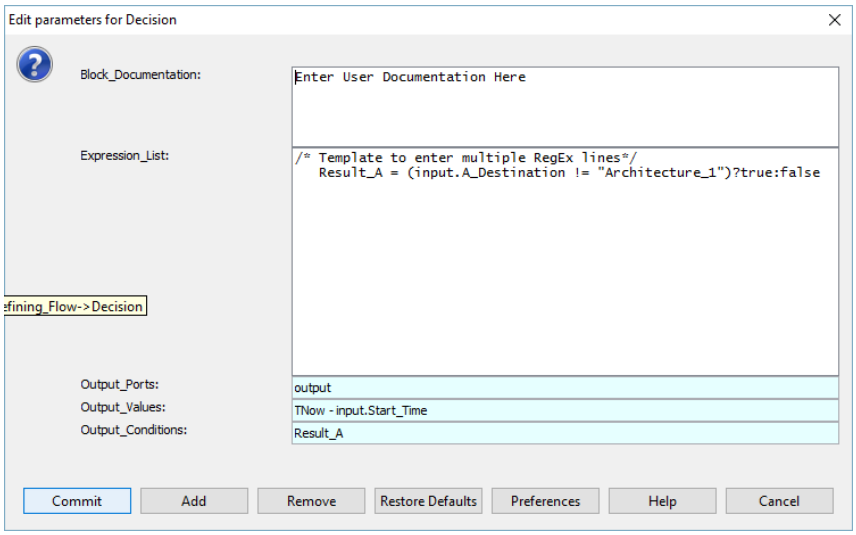
3. Instantiate Delay block from the library pane Delay and Utilities → Delay. Double click on the block and enter Delay_Value as 5.0e-8
4. Instantiate Expression_List or Processing block from the library pane. Define the expression as below under Expression
   input.A_Command = (input.A_Command == "Read")?"Write":"Read"
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



10.8.2.4 Step4

In this step we will configure model for capturing statistics and end-to-end latency

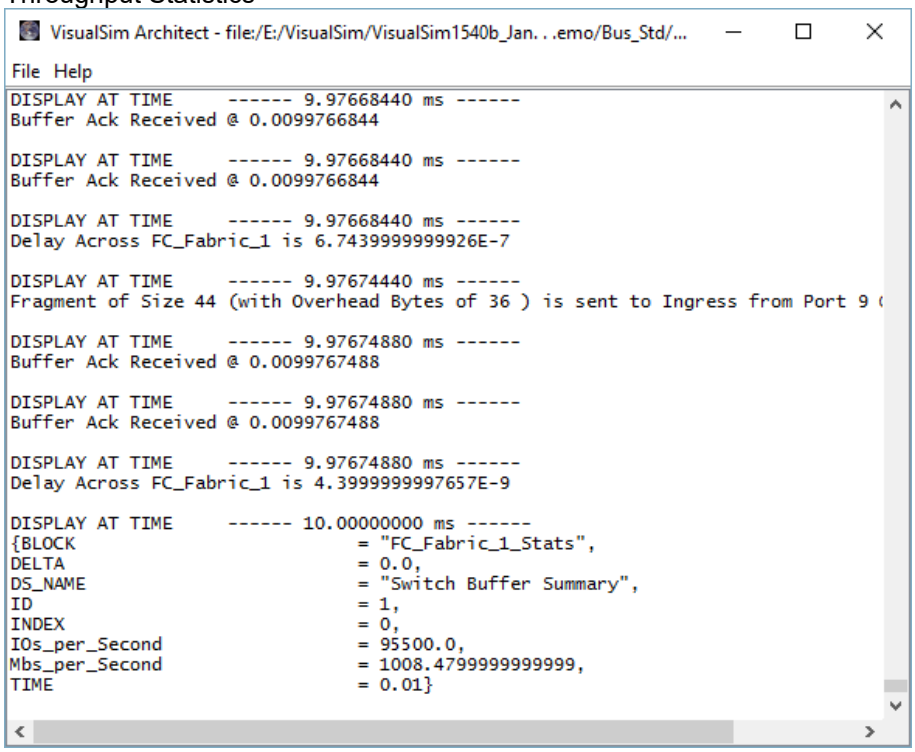
1. Connect data_out port of FC_TG block to a Decision block and configure the Decision block as below



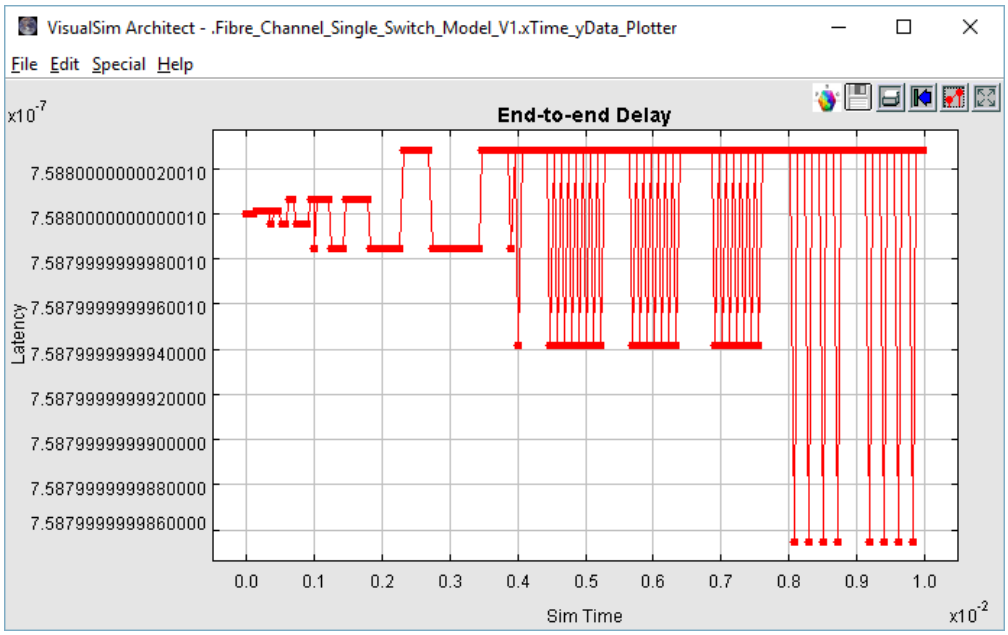
2. Instantiate xTime_yData Plotter from the library pane Results →Plotter → xTime_yData plotter
3. Connect output port of Decision block to input port of plotter

10.8.2.5 Reports

Throughput Statistics



10.8.2.6 End-to-End Latency

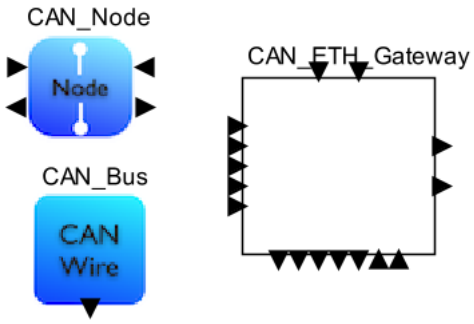


10.8.2.7 Analysis

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

11 CAN

Block Description
The Controller Area Network (CAN) is a serial communications protocol which efficiently supports distributed realtime control with a very high level of security. What we have in visualsim is an extensive library dedicated to all the protocols implemented in an Automobile. CAN is one of them. It provides a very flexible network. Nodes can be added without much difficulty.



The following properties of the CAN has been implemented in VisualSim :

1. Prioritization of Messages
2. Configuration flexibility
3. Multicast reception with time synchronization
4. System wide data consistency
5. Multi master
6. Error detection and signalling
7. Automatic retransmission of corrupted messages as soon as the bus is idle again
8. Distinction between temporary errors and permanent faliures of nodes and autonomous switching off of defect nodes.
9. Both 11 bit and 29 bit can co-exist in the bus at the same time

The CAN_Bus module alone wont be functional. We need to add CAN_Nodes (atleast 2) . The CAN_Nodes communicate to each other with the help of arbitration algorithms and access controls defined in CAN_Bus.

CAN_Bus Parameter:

CAN_Bus_Name
The parameter is used to specify the name of the CAN Bus. This name has to be matched in the Nodes attached to it

CAN_Type
User can configure either the Classical_CAN, CAN_FD or CAN_XL using this pull down parameter

Arbitration_Bit_Rate_Mbps
This parameter can be used to define the Arbitration bit rates

Data_Bit_Rate_Mbps
This parameter can be used to define the data transfer bit rates

Signal_DB
User can modify this parameter and the Signal database will be updated. The signals are generated based on the info specified here

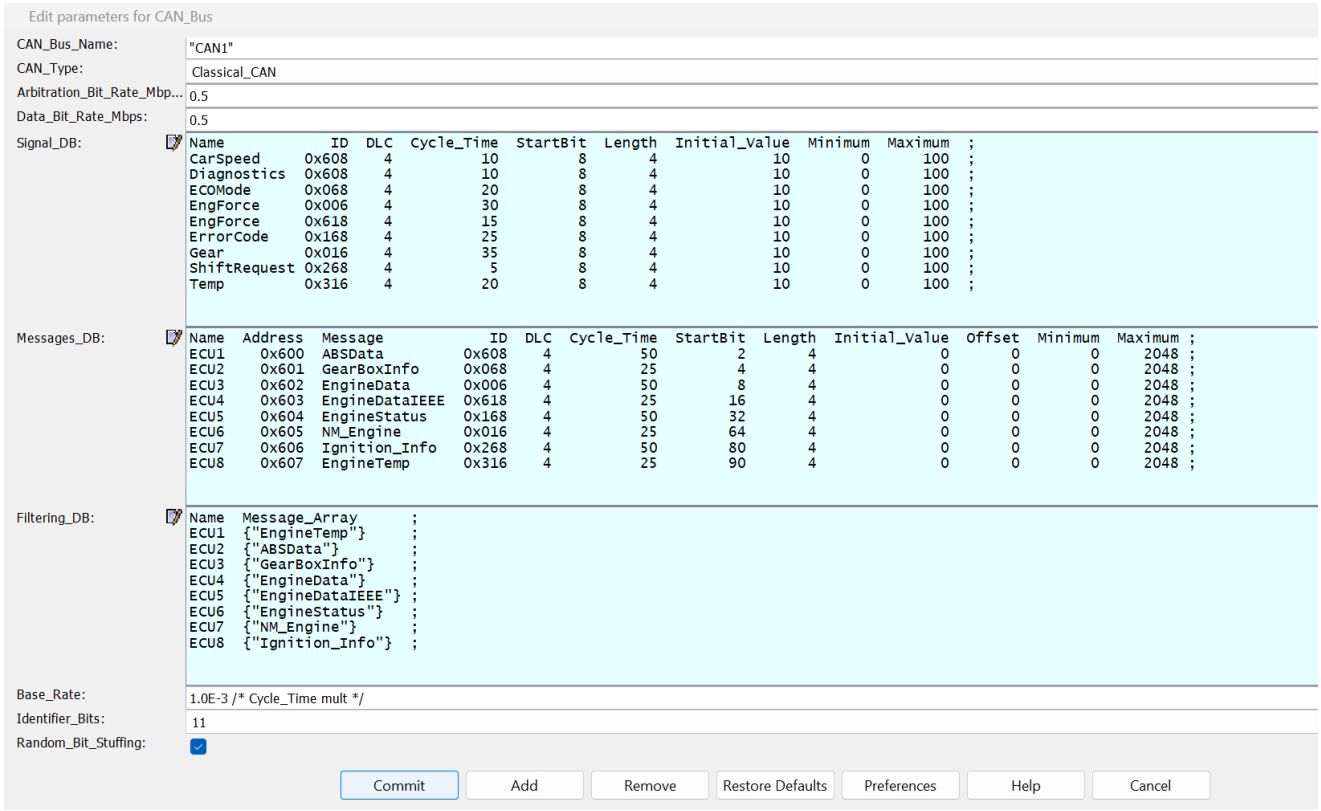
Messages_DB
User can modify this parameter and the Message database will be updated. The Messages are generated based on the info specified here

Filtering_DB
This parameter is used for updating the filter database. This table provides info on which messagesare accepted in each nodes

Base_Rate
This parameter is used to accept the Base Rate

Identifier_Bits
This is a pull down parameter. User can choose between 11 bit and 29 bit. Based on this value, the frames generated based on database (Messages_DB) will have the selected Frame format (Standard or Extended).

Random_Bit_Stuffing
If this paramter is enabled, then Bit stuffing will be enabled



Signal_DB

Name	ID	DLC	Cycle_Time	StartBit	Length	Initial_Value	Minimum	Maximum
CarSpeed	0x608	4	10	8	4	10	0	2048

Notes:

1. An Entry for every Message ID specified in Messages_DB has to be specified in Signal_DB. Otherwise an error will be generated.
   Example:

   CAN1_Sensor_ECU1 Signal ID Column:
   {"0x6108", "0x618", "0x068", "0x006", "0x168", "0x016", "0x268", "0x316"}
    does not contain Message ID ("0x608"), suggest editing Signal or Message entries.

2. Signal Length values cannot be greater than Signal DLC value .

3. Multiple signals can be linked with a single Message. Useful in defining Signal Routing vs Message Routing. When having multiple signals linked , we have to make sure sum of the Signal Lengths are less than 64 Bytes. Else exception will be thrown. Same with Signal DLC values as well.

4. Sum of Signal Lengths cannot be greater than Signal DLC's.

5. Sum of Signal DLC's cannot be greater than he MEssage DLC that the signals are linked to.

Message DB

Name	Address	Message	ID	DLC	Cycle_Time	StartBit	Length	Initial_Value	Offset	Minimum	Maximum
ECU1	0x600	ABSData	0x608	4	50	2	4	0	0	0	2048

Notes:

1. All ECU mentioned in the column "Name" should be linked to a CAN_Node. If the CAN_Bus cannot find the Node with the said name, then it will throw an error.
   Example: I added a column with Name specified as ECU9 and I dont have any CAN_Node with the name "ECU9", then it will throw the following error:
   CAN Wire (CAN1) cannot find a referenced Node: CAN1_ECU9 sets Error_Flag.
   {CAN Wire simple means CAN_Bus}

2. The Address specified in the Address Column has to be unique. If not, CAN_Bus will throw an Error/Warning.

The Messages_DB is used in CAN_Node. This table is used for generating traffic packets synthetically from the CAN_Node according to the column value. The CAN_Node can also be configued to accept External inputs. which means, the CAN_Node wont be using this database for traffic generation. But, make sure the incoming packet Address is specified in the Address column of Messages_DB as well as the Message of the incoming packet should also match with Message column value.

If a CAN_Node opts for generating traffic from Database (Please Refer CAN_Node documentation for more details on CAN_Node parameter configurations), then the following cases needs to be checked by the user:
   a. Length column value cannot be greater than DLC column value for the same Row
   b. Address Column value and ID column value for the same Row cannot be same
   c. Message DLC value should be greater than or equal to the sum of the signal DLC's. Which menas if

Message DLC value is 4, then we cannot have 2 signals assigned with sum of thier DLC greater than 4.
d. Message DLC column values cannot be greater than 64 Bytes.

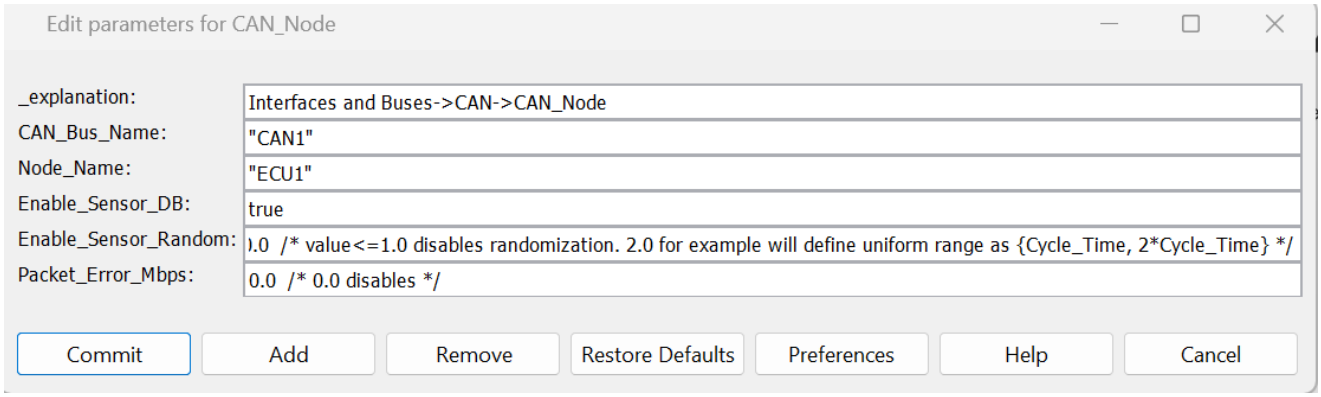
Filtering DB

Name	Message_Array
ECU1	{“ABSData”,”EngineData”}
ECU2	{“none”}
ECU3	{“ALL”}

Notes:

1. Messages are accepted on a CAN_Node according to the Message_Array specified for it.
2. The Array can contain multiple Message Names.
3. If the Message_Array contains "ALL", then all messages are accepted on that CAN_Node.
4. If the Message_Array contains "none", then no messages are accepted on that CAN_Node.

CAN_Node Parameter:



CAN_Bus_Name
The parameter is used to specify the name of the CAN Bus. This name has to be matched with the CAN_Bus_Name provide in the CAN_Bus module.

Node_Name
This parameter specifes the Node Name and this has to be unique in a CAN_Bus network.

Enable_Sensor_DB
This parameter is used to specify whether we are generating traffic from Database or is this node getting packets from External source. True means the traffic is generated from the database.

Enable_Sensor_Random
true- to randomize the sensor data rates that are listed in the Signal table. The Signal table is in the CAN_Bus block. [This Parameter not used anymore]

Packet_Error_Mbps
A value of 0.0 disables this error generation.If according to the value specified a failure was detected then Error counters are incremented and when reaching threshold, appropriate actions will be taken.

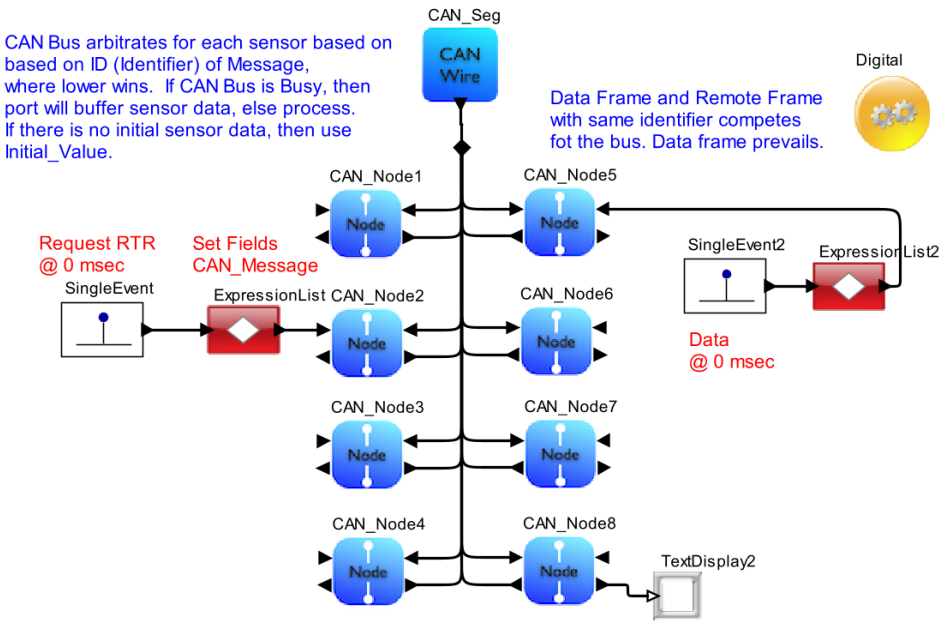
CAN Node ports

node_in : Data from External Traffic generator is fed in here
node_out : Data Accepted by this Node is sent out through this port. This acceptance is based on the entries specified in Filterring_DB.
rx : Data sent from CAN_Bus is received here
tx : Data to CAN_Bus is sent on here

Required Input Fields at CAN Node:

IDE	Required for specifying the Identifier.0 for 11 bits and 1 for 29 Bits.	0
DLC	It should be less than 64 Bytes	8
ID	USed to specify the Message ID	"0x108"
Message	Specify the Message name	"ABSData"
Length	Specify the Length in Bytes	8
Error_Flag	Specify the State of the Error_Flag. If True, then it is an Error Frame.	false
Overload_Flag	Specify the State of the Overload_Flag. If True, then it is an Overload Frame.	false
RTR_Bit	Specify the State of the Remote_Bit. If True, then it is a Remote Frame. Else Data frame.	false
A_Source	To Specify the Source which generated this traffic	"Brake_ECU"
Address	The Address Mentioned on this field should also be specified in the Messages_DB and Signal_DB	"0x608"

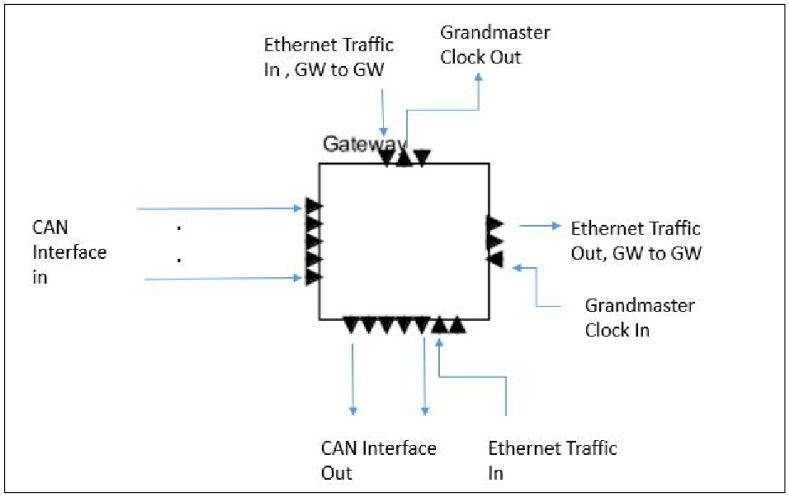
The following image shows the simple structure of CAN Nodes and CAN bus.



CAN ETH Gateway

The Gateway block by default provides five CAN Bus interfaces and two Ethernet Interfaces. We use an IEEE 802.1 compliant sheduler in the interfaces. All the standards supported within are as follows:

IEEE 802.1Qbv	: Time Aware Shaping
IEEE 802.1Qbu & IEEE 802.3br	: Preemption
IEEE 802.1Qca	: Path control and Reservation
IEEE 802.1Qcc	: Stream Reservation Protocol
IEEE 802.1Qci	: Per Stream Filtering and Policing
IEEE 802.1Qcb	: Frame Replication and elimination
IEEE 802.1Qch	: Cyclic Frequency and Forwarding
IEEE 802.1As	: Enhanced Generic Precision Time Protocol



When the packets come into the CAN Bus interface, user can opt for signal routing or Message Routing by modifying a table called Classification_Table. In Classification_Table user can mention Hash_ID assosciated with each Message or with the signal that has to be transmitted. Hash_ID is a unique ID assigned to a Message/Signal structure that will be transmitted through the network. The destinations of these frames can be mentioned from a database called Forwarding_Table. If the destination mentioned for a Hash_ID is a local CAN interfcae, then it will be send to that interface. But if the packets are to be sent to another Gateway, then we specify one of the Ethernet Interfaces as the Task_Destination.The TSN library makes use of a couple of database blocks which can be easily obtained by instantiating a module named DB from VisualSim\actor\arch\TSN. The following are the databases used in building the Gateway:

1. Class DB - explained below.
2. Frwd_DB - refer TSN section (*TSN Switch Config DB*)
3. GCL DB - refer TSN section (*TSN Switch Config DB*)
4. Priority DB - refer TSN section (*TSN Switch Config DB*)
5. Avg Class to Priority DB - refer TSN section (*TSN Switch Config DB*)

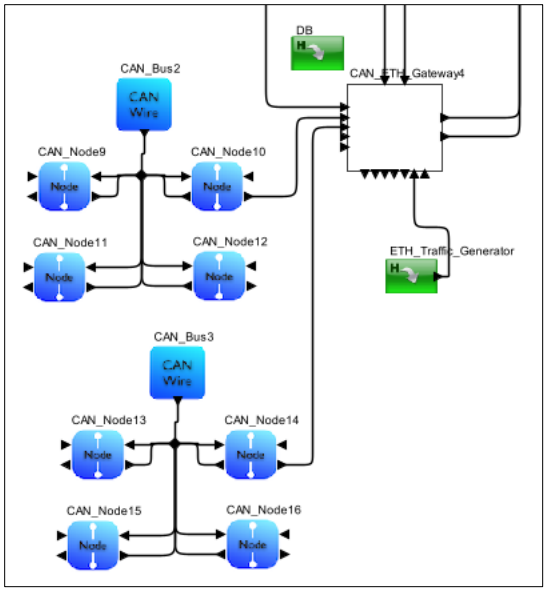
Classification Table

This table is used to define the required Routing mechanism. It can be Signal Routing or Message Routing. For Each Row entered in the Classification Table, we assign a unique ID called Hash_ID. This ID will be unique in the enitre network. The following table will be how a classification table is supposed to look like :

Gateway	Source	ID	Routing
GW1	CAN1	1	{"Ignition_Info","ABSData"}
GW1	CAN1	2	{"EngineData"}

A CAN bus named, "CAN1" is connected to the Gateway named "GW1" . The ID colum corresponds to the Hash_ID . The routing Column corresponds to the messages that have to be encapsulated so as to send thoughtout the network. For Hash ID 1, we have defined Message Routing, as two signals are put together and send out in a single frame. The Hash_ID two is designed to follow signal routing as the routing column only contains one signal, which is EngineData. So when EngineData comes into the CAN Node Interface module, we just adds the Hash_ID field, Timestamp and sends it out. For Message Routing, multiple signals have to be arrived. So when all the required signals have arrived at the CAN Node interface module, the Hash_ID and Timestamp is added and is send out.

The following image shows the connection between CAN and CAN ETH gateway.



The CAN ETH Gateway uses internal sub modules with default configuration. The following subsections are built in this Gateway block.

1. CAN Interfaces
2. Block DB
3. Grandmaster clock
4. ETH Interface

CAN Interface

The CAN_Interface module provided in the library is responsible the CAN bus interface to the gateway. If the user wants to connect multiple CAN bus to the gateway and route the packets in between, then multiple CAN

Node modules has to be instantiated. The CAN Interface module will support Message/Signal Routing. For this purpose, it looks into Classification Table.

The following are the parameters provided for a CAN Interface Module :
GW_Name
Provide the Gateway name in which the CAN Node is placed
Default: GW_Name (top parameter)

CAN_Bus_Name
"CAN1"
Provide the CAN Bus name from whichc connections are added
Eg: "CAN1"

Max_Bytes
Provide the maximum byte size that can be sent through
Default: 30

Bit_Rate_Mhz
Provide the CAN Bit Rate in Mhz
Default: 1.0

Block_DB
This module is used for mainitaning connections between the other modules. This is a vital block and it requires the Gateway name, the top level parameter is linked by default.

Grandmaster Clock
This module acts as the Granmaster clock and is sent across the entire network. The redundancy is implemented.

The following are the paramters supported by this module:

Grandmaster_Name
Provide the naame of the Grandmaster
Default: "GM1"

Grandmaster_Enable
Set to true if Grandmaster is enabled
Default: true

GW_Name
Provide the Gateway name in which the CAN Node is placed
Default: GW_Name (top level parameter)

Bit_Rate_Mhz
Provide the CAN Bit Rate in Mhz
Default: 1.0

Overhead
Provide the overhead bytes
Default: 12

Delay_grandmaster
Defines the internal delay of grandmaster clock
Default: (Overhead+2)/(Bit_Rate_Mhz*1.0e6)

Ethernet Interface

The ETH_Interface Module provided in the library is responsible for the Ethernet interface. This module implements all the IEEE802.1Q standards which are necessary for the automotive driving network.
The following are the parmeters supported by this module:

GW_Name
Provide the gateway name in which Ethernet Node is placed
Default: GW_Name (top level parameter)

ETH_Name
Provide the Ethernet Interface Name
Eg: "ETH1"

Max_Bytes
Provide the Max Bytes that can be sent out through Ethernet
Default: 30

Speed_MBps
Provide the speed in MBps
Default: 300.0

Overhead
Provide the overhead bytes in sending the grandmaster clock through the network
Default: 12

Idle_Slop_A_MBps
Provide the Idle Slope for Class A in MBps
Default: 100.0

Idle_Slope_B_MBps
Provide the Idle Slope for Class B in MBps
Default: 100.0

Send_Slope_A_MBps
Provide the Send Slope for Class A in MBps
Default: 2000.0

Send_Slope_B_MBps
Provide the Send Slope for Class B in MBps
Default: 100.0

Offset
Provide the offset by which the transmitting should be shifted
Default: 0.0

Guardband
Provide the time period for the Guardband
Default: 50.0e-7

Percent_Class_A_B
Provide the Class A and B bw percentage combined
Default: 30

Class_B_BW_Percentage
Provide the Class B bandwidth alone

Default: 30

Plot_Enable
Provide true if buffer occupancy, send slope, idle slope plots have to be made visible
Default: Plot_Enable (top level parameter)

hiCredit
Provide the hiCredit value;Credit for Class A and B cannot go above it
Default: 8000.0

loCredit
Provide the loCredit value; credit cannot go below it. User is nottified if it happens
Default: -70000.0

Use_General_Priority
Drop down menu. Select required choice and Bandwidth for priority classes and AVB class assigning will be based on this parameter. Mentioned about it in Tables section.
Default: GW_Specific

Send_CDT_low_BW
Select if CDT can be sent even if the bandwidth is very low
Default: true

CDT_BW_Percentage
Provide the allotted CDT bandwidth percentage (percenatge with the total)
Default: 10.0

Borrow_BW
Select true if bandwidth can be borrowed from classes which are not using it
Default: true

Utilization_Check_Period
Provide the period through which utilization for the link has to be measured
Default: 50.0e-3

View_Link_Utilization
Select this paramter if the user wants to see the utilization percentage
Default: true

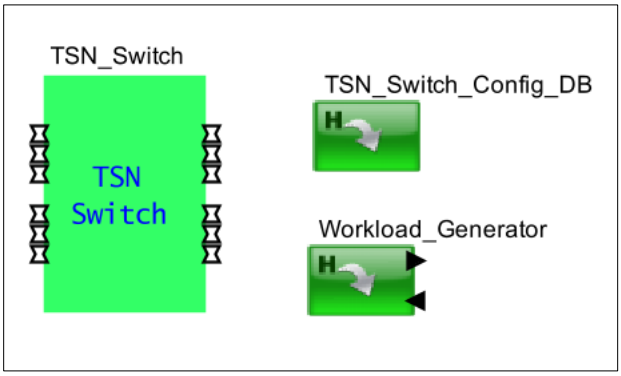
Bit_Rate_Mhz
Provide the CAN Bit Rate in Mhz
Default: 1.0

12 TSN

Block Description

This module represents the Protocol variant of TSN Ether Switch. This module is built with various configurable parameters which will help user to easily map the settings of a target EtherSwitch. By default, the TSN_Switch provides 12 interfaces to the user. This means, 12 connections can be made to this switch from other masters, networks, ECUs, Slaves, Sinks etc.. If more than 12 interfaces are required, then the additional interfaces can be instantiated as per users requirement

For the user to simulate models using TSN_Switch, user must instantiate TSN_Switch_Config_DB (Location : Interfaces and Buses -> TSN_Switch)



Input fields required:

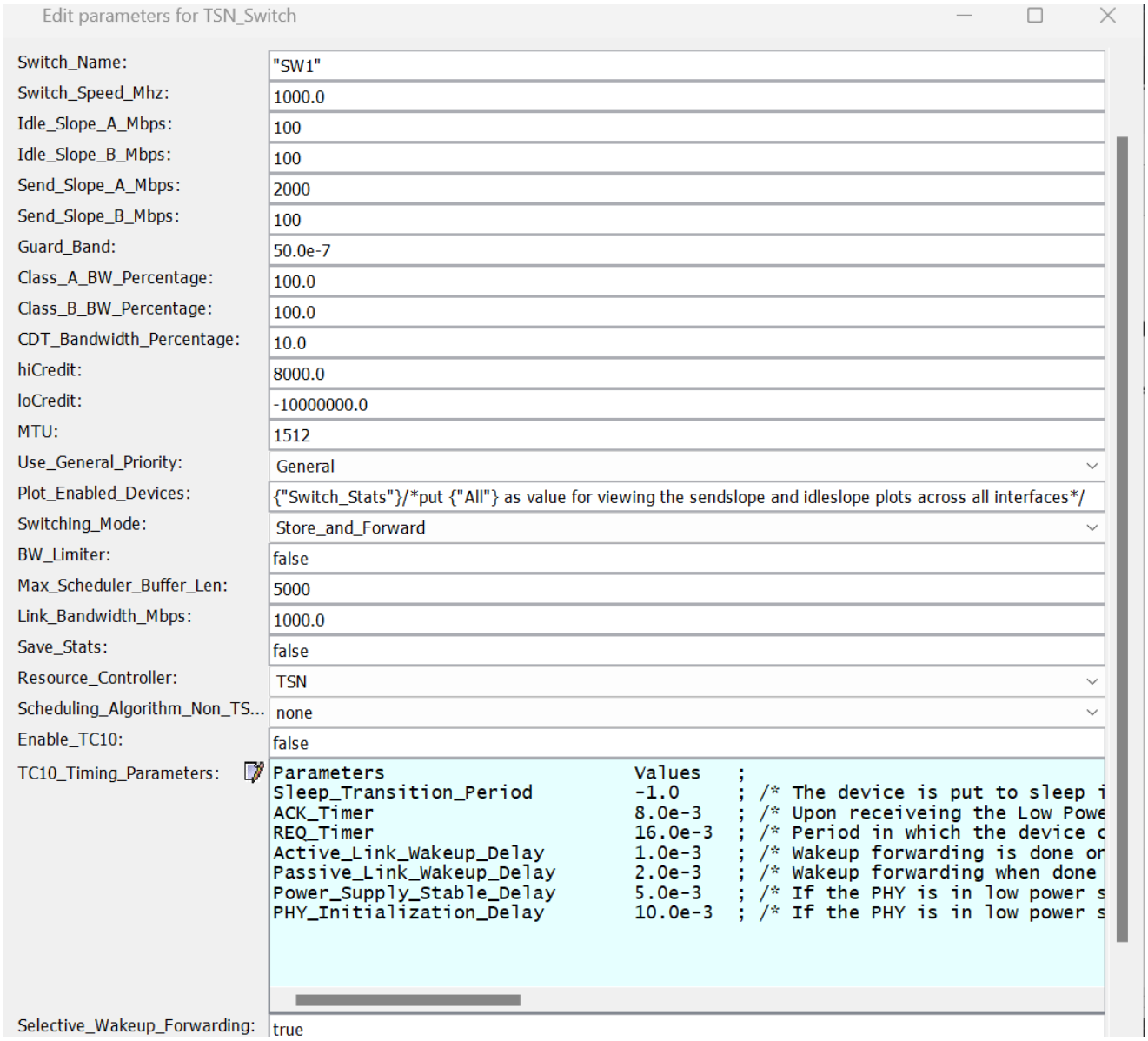
Use the Data Structure template defined by the name "Task_Class". Along with the fields defined within "Task_Class", add the following:

Data Structure Field Name	Description	Value (Examples)
Task_Size	Size of the packet in bytes	1512
Destination_Port	Incoming port number.	15 or irand(1,1023)
Message	Used to define the message value. It will be used in Gateway module, if used in model (for packing).	"ETH_Transfer"
Hash_Id	Unique Identifier. It will be used while doing Forwarding Table lookup.	"0x102"
Transport_Protocol	Define the Transport layer protocol type assosciated with the stream of message. Possible values - "TCP" or "UDP"	"UDP"

As long as the incoming data structure contains all the required inputs as specified above, it will work well. Else, an exception will be thrown.

TSN_Switch can be connected to the Gateway module directly to any one of the ports.
Worloads which generate frames in the required data structure format can be connected directly to any one of the port.

TSN Switch Parameters:



Switch_Name
Unique name for the switch
Eg: "SW1"

Switch_Speed_Mhz
Switch speed in Mhz
Eg: 1500.0

Link_Bandwidth_Mbps
Total Bandwidth for each of the link.
Eg: 100.0

Idle_Slope_A_Mbps
Idle slope rate for Class A. Credit rebuild rate

Eg: 100.0

Idle_Slope_B_Mbps
Idle slope rate for Class B. Credit rebuild rate
Eg: 100.0

Send_Slope_A_Mbps
Send slope rate for Class A. Credit use rate
Eg: 100.0

Send_Slope_B_Mbps
Send slope rate for Class B. Credit use rate
Eg: 100.0

Guard_Band
Guardband period in seconds
Eg: 50.0e-7

Class_A_BW_Percentage
Class A BW percentage. Class A and a periority level shares the total BW of a priority level. This percentage parameter specifies how much BW can Class A take from that Priority Level
Eg: 100.0

Class_B_BW_Percentage
Class B BW percentage. Class B and a periority level shares the total BW of a priority level. This percentage parameter specifies how much BW can Class B take from that Priority Level
Eg: 60.0

CDT_Bandwidth_Percentage
CDT BW percentage.
Eg: 10.0

hiCredit
max credit value. The credit value (for classes A and B) is capped at hiCredit
Eg: 8000.0

loCredit
low threshold limit for the credit. if the credit goes below, then an error is thrown.
Eg: -10000000.0

MTU
Max Transmission Unit in Bytes. If frame size is higher than MTU, frame is fragmented internally.
Eg: 1512

Use_General_Priority
General means the TSN_Switch will use the genral column value from the database. Switch_Specific wil use parameter values specific to each switch.
Eg: General

Plot_Enabled_Devices
Since we have 12 interfaces, user can decide which all of the device interface plots (SendSlope and IdleSlope) should be turned on. User can specify "All", "Switch_Stats" or specific interface names like "ETH0","ETH1" etc..
Eg: {"All"}

Switching_Mode
Supports two options - Cut_Through and Store_and_Forward. In Store and Forward, entire frame is is reassembled before sending it out. In Cut through, each fragments are sent out.
Eg: Store_and_Forward

BW_Limiter
Set it to be true if the bandwidth per classes needs to be limited. If it is false, then all classes enjoy full bandwidth at different instances of time.
Eg: true

Max_Scheduler_Buffer_Len
Used to specify the max buffer size of the scheduler buffer at each interface.
Eg: 5000

Save_Stats
Set this parameter to be true if the sats generated across this TSN Switch needs to be saved.
Eg: false

Resource_Controller
Use this parameter inorder to select between TSN (Time Aware Shaper) and Scheduler algorithms which are listed under a parameter called "Scheduling_Algorithm_Non_TSN"
Eg: TSN

Scheduling_Algorithm_Non_TSN
This parameter lists the various scheduling algorithms that can be selected when the switch is configured to use non_TSN scheduling algorithm.
Options:
FIFO
FCFS
LIFO
Round Robin
Weighted Round Robin
Weighted Fair Queueing
Deficit Round Robin
Strictly priority
Round Robin Priority

Eg: none

Enable_TC10
Set this parameter to be true if TC10 Sleep/Wakeup needs to be enabled
Eg: false

TC10_Timing_Parameters
This parameter defines a list of timing values used by the TC10 standard. Configure them as required

Parameters **Values** ;
Sleep_Transition_Period -1.0; */* The device is put to sleep if the device is idle for this much amount of time */*
ACK_Timer 8.0e-3 ; */* Upon receiveing the Low Power Signal request, the receiving device has to respond with an ACK if it is idle. The period at which the receiving device waits to confirm whether its idle is set by this value */*
REQ_Timer 16.0e-3 ; */* Period in which the device can move to sleep state if no activity is observed across the device */*
Active_Link_Wakeup_Delay 1.0e-3 ; */* Wakeup forwarding is done on top of active links as well. In such cases, the time it takes to send the wakeup signals across an active link is set using this value */*
Passive_Link_Wakeup_Delay 2.0e-3 ; */* Wakeup forwarding when done on top of passive links will take a certain amount of time to complete. It is set by this value */*
Power_Supply_Stable_Delay 5.0e-3 ; */* If the PHY is in low power sleep state, then once the wakeup is detected, Power supply needs to be brought back on. It will take a cerain amount of time to get the power value stable */*
PHY_Initialization_Delay 10.0e-3 ; */* If the PHY is in low power sleep state, then once the wakeup is detected, PHY needs to be initialized. It will take a cerain amount of time to get the initialization completed */*

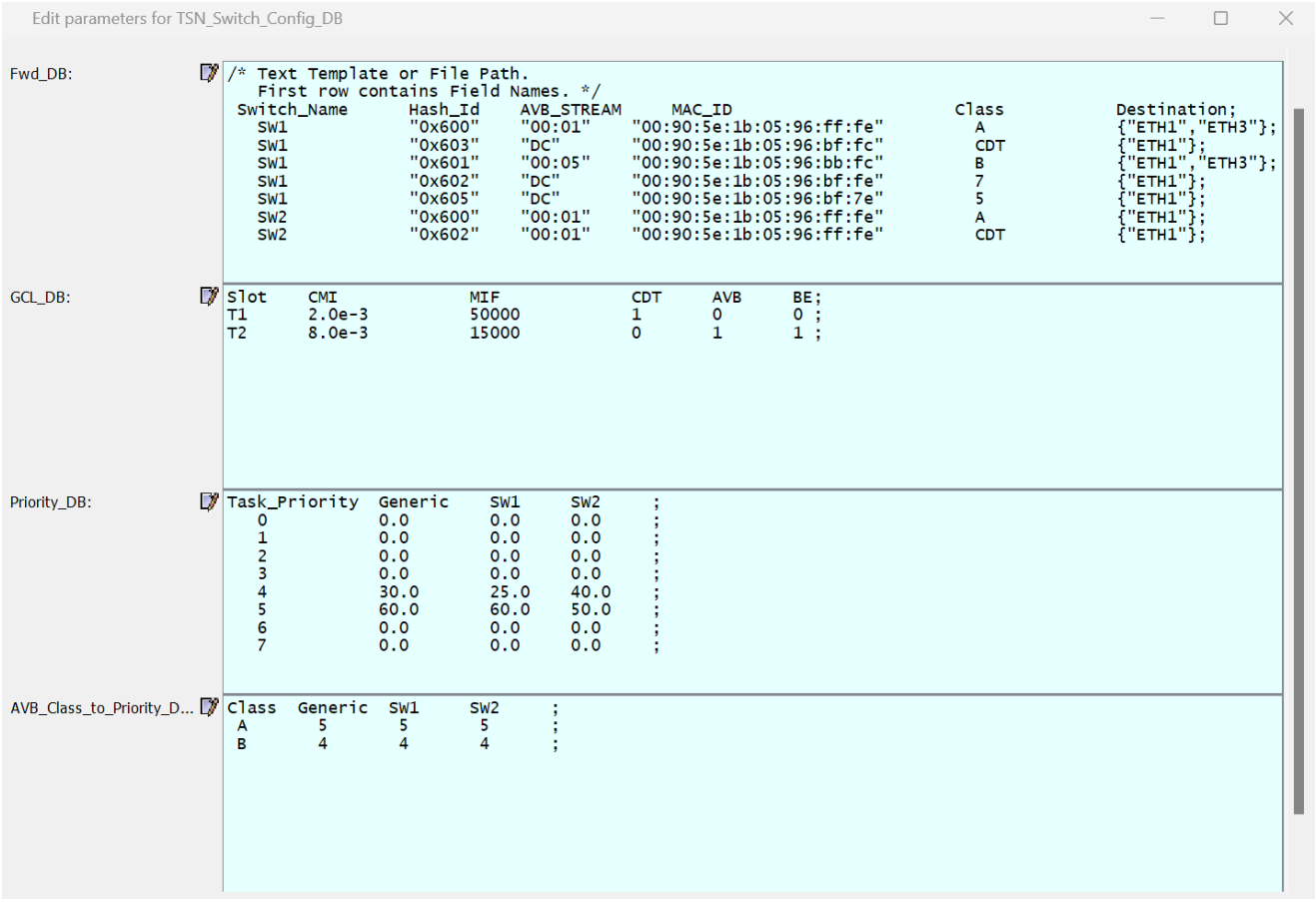
Selective_Wakeup_Forwarding
Switches are defined with two options: one is to do selective wakeup forwarding and other is to do a broadcast. So if this parameter is set to a value of false, then wakeup signals are broadcasted to all ports of the switch.
Eg: true

Power_Manager
This parameter is used to define the Power Manager name. The defined power manager must contain a state called "Low_Power_Sleep" and define the ports in the following format:
"Power_"+Switch_Name+"_"+Port_Name. Port_Names are ETH0,ETH1....,ETH11
Eg: “none”

TSN Switch Config DB

This module provides a list of tables for the user to apply their target settings - including setting of packet destinations, bandwidth allocation across priority classes, Time Aware Shaper settings etc..

Parameter:



Forwarding Table (Fwd_DB)

This table is used to define the forwarding of the packets within each TSN_Switch. If a packet having a particular Hash_Id needs to be sent to multiple switch interfaces, then user just have to specify the

corresponding port names in Destination column. The following Table will be how a forwarding tabe is supposed to look like:

Switch_Name	Hash_Id	AVB_STREAM	MAC_ID	Class	Destination
SW1	"0x001"	"00:01"	"00:90:5e:1b:05:96:ff:fe"	"CDT"	{"ETH0"}
SW1	"0x002"	"DC"	"00:90:5e:1b:05:96:ff:fa"	"5"	{"ETH1","ETH10"}
SW1	"0x106"	"00:02"	"00:90:5e:1b:05:96:ff:3a"	"A"	{"ETH6"}
SW2	"0x503"	"DC"	"00:90:5e:1b:05:96:ff:fe"	"7"	{"ETH6"}

Lets examine this row by row. In the first row, we could see that Hash_Id is defined as a Class CDT and is given the destination {"ETH0"}. So when a packet having Hash_Id ="0x001" comes into the TSN_Switch we accept the packet and is send out on the "ETH0" port .If we look at the row 2, we can see the Hash_Id "0x002". We are sending the packet to multiple destinations - ETH1 and ETH10. Set the AVB_STREAM as "DC" (dont care) or give a unique identifer.

Gate Control List (GCL_DB)

This database is used to define the metrics for the Time Aware Shapper. The user can define how many slots needs to be there in a period. The Guardband period is a parameter provided in the TSN_Switch. So the slot period provided in the Gate Control List + the Guradband period will be the entire period time and this will be repeated till the end of the simulation. The following Table will be how a Gate Control List is supposed to look like:

Slot	CMI	MIF	CDT	AVB	BE
T1	2.0e-3	50000	1	0	0
T2	8.0e-3	15000	0	1	1

Slot corresponds to the slot name. The column CMI corresponds to Class Measurement Interval for the corresponding Time Slot. The Coumn MIF corresponds to the Maximum Interval Frame for the corresponding time slot. The other three columns say which of the traffic classes can be sent on that time slot. Lets see an example:
The CDT Frames can be sent from 0.0 to 2 milliseconds. Then AVB classes and BE classes can be sent out, which will be for 8 milliseconds. So now a total of 10 milliseconds are over. Now comes the guardband where all the packets can be sent out if available. We look in the sequence of CDT,AVB and BE. This is to effectively utilize the bandwidth and not to waste it by making the link idle. Lets say guardband is of time 50 microseconds. Then an entire period will be completed by 10.050 milliseconds and from 10.050 milliseconds to 12.050 milliseconds, CDT frames can be sent out again and this process repeats till the end of the simulation.

During Time Slot “T2”(8 msec), both AVB and BE frames can go through. Class A and B frames have higher priority than BE. But the Class A and B frames are processed through Credit Based Shaper (CBS) and thus could send frames through if they have non-negative credit. When the Class A or B frames are not being sent and put to wait in the buffer, their credits start building up. During that time, BE frames go through.

Priority BW table (Priority_DB)

This database is used to define the bandwidth allocated in percentage for each priority (7 -> 0) in each TSN_Switch. The following will be how priority BW table is supposed to look like :

Priority	Generic	SW1	SW2
7	0.0	10.0	20.0
6	0.0	15.0	5.0
5	30.0	20.0	10.0
4	20.0	20.0	30.0
3	0.0	5.0	5.0
2	0.0	5.0	10.0
1	0.0	0.0	5.0
0	0.0	0.0	5.0

The user can speecify the bandwidth assosciated wil each priority classes. If the network designed is huge and wants to have same badwidth specifications across every TSN_Switch, then the user can select Generic option in the TSN_Switches and then the bandwidth allocalted under the genric column will be only used for all TSN_Switches. Else, the bandwidth specifed for corresponding TSN_Switch will be used. Each entry is a percentage of the total Link_Bandwidth_Mbps value set in the TSN_Switch. Please make sure that the sum of all the percentages + CDT_Bandwidth_Percentage (Set in TSN_Switch) doesnot go above 100%.

Class to Priority table (AVB_Class_to_Priority_DB)

This database is used to define the the priority classes to which AVB classes are assigned in each TSN_Switch. The following will be how the class to priority table is supposed to look like:

Class	Generic	SW1	SW2
A	5	5	5
B	4	4	4

The user can either specify the priority classes assigned for the AVB classes for the entire network by using genric or can go for TSN_Switch specific priority classes.

Workload Generation:

It generates frames according to the profile set by the user. This module can be connected to any one of the TSN_Switch ports.

Parameter:

Enable_Traffic_Generation
True/False - check or uncheck the box. If true, then traffic will be generated by this workload generator. Else it will not generate any packet.

Start_Time
The time at which the workload generator can start generating frames

Traffic_Gen_Rate
Specify the rate at which the frames must be generated.
Traffic_Gen_Rate_Unit
Provides two options - Sec or Mbps. If it is set as Sec, then Traffic_Gen_Rate parameter value will be considered in Seconds. So if Traffic_Gen_Rate = 10.0e-3 and the Unit is Sec, then the frame will be generated every 10 msec. If it is Mbps, then the Workload generator will make sure that the frames are generated to match the corresponding rate.

Hash_Id
A Unique Identifer added to the frames generated. It will used at the TSN_Switch

Transport_Protocol
Transport protocol for the corresponding frame. Two options - TCP or UDP

Frame_Size_Bytes
Size of the Frame in Bytes

Frame_Class
Class of the Frame. Class A or Class B or Class CDT or any of the Best Effort Classes (0 to 7)

Device_Name
A unique name for the workload generator.

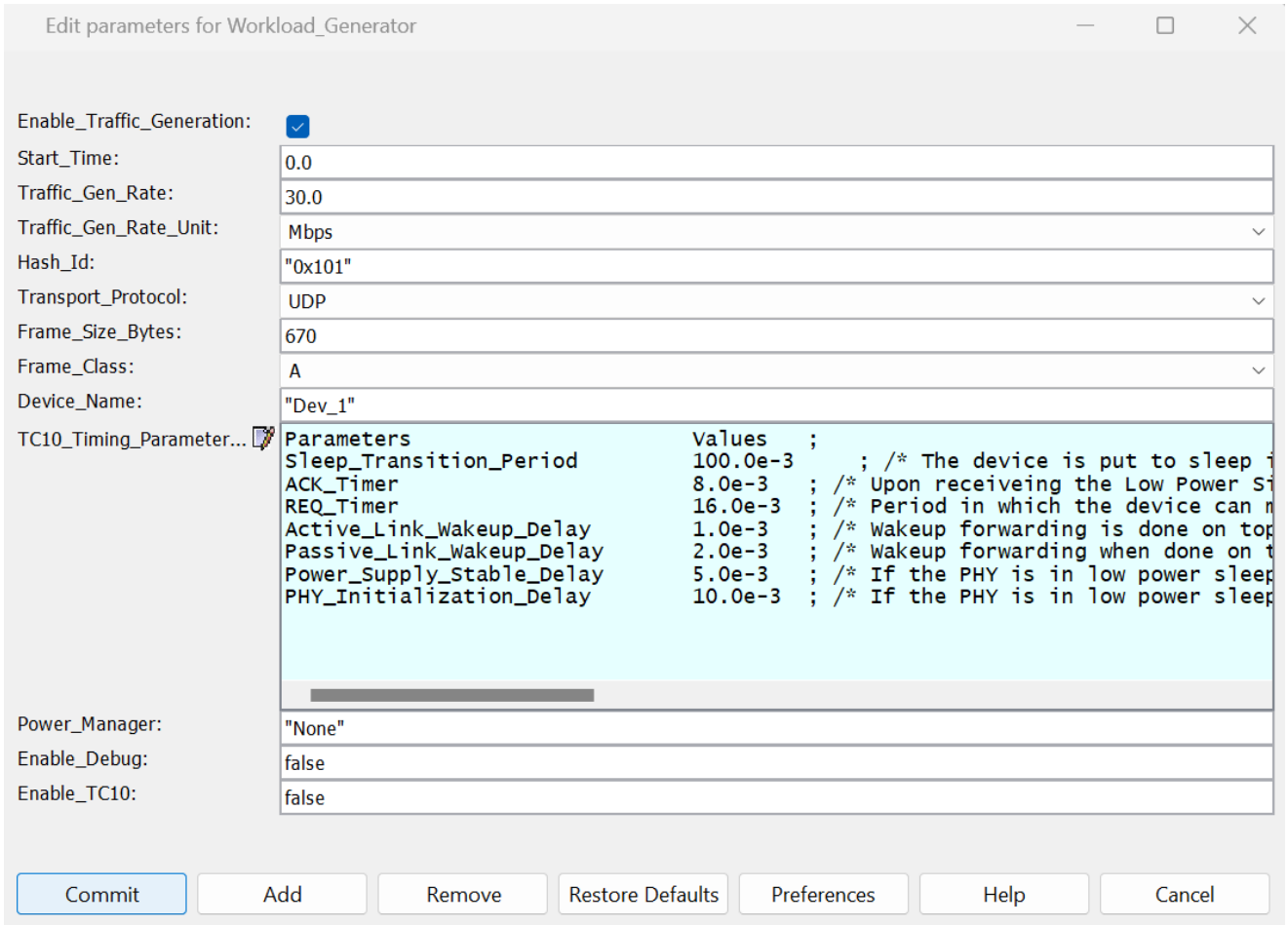
TC10_Timing_Parameters
This parameter defines a list of timing values used by the TC10 standard. Configure them as required

Parameters Values ;
Sleep_Transition_Period 100.0e-3; */* The device is put to sleep if the device is idle for this much amount of time */*
ACK_Timer 8.0e-3 ; */* Upon receiveing the Low Power Signal request, the receiving device has to respond with an ACK if it is idle. The period at which the receiving device waits to confirm whether its idle is set by this value */*
REQ_Timer 16.0e-3 ; */* Period in which the device can move to sleep state if no activity is observed across the device */*
Active_Link_Wakeup_Delay 1.0e-3 *; /* Wakeup forwarding is done on top of active links as well. In such cases, the time it takes to send the wakeup signals across an active link is set using this value */*
Passive_Link_Wakeup_Delay 2.0e-3 ; */* Wakeup forwarding when done on top of passive links will take a certain amount of time to complete. It is set by this value */*
Power_Supply_Stable_Delay 5.0e-3 *; /* If the PHY is in low power sleep state, then once the wakeup is detected, Power supply needs to be brought back on. It will take a cerain amount of time to get the power value stable */*
PHY_Initialization_Delay 10.0e-3 ; */* If the PHY is in low power sleep state, then once the wakeup is detected, PHY needs to be initialized. It will take a cerain amount of time to get the initialization completed */*

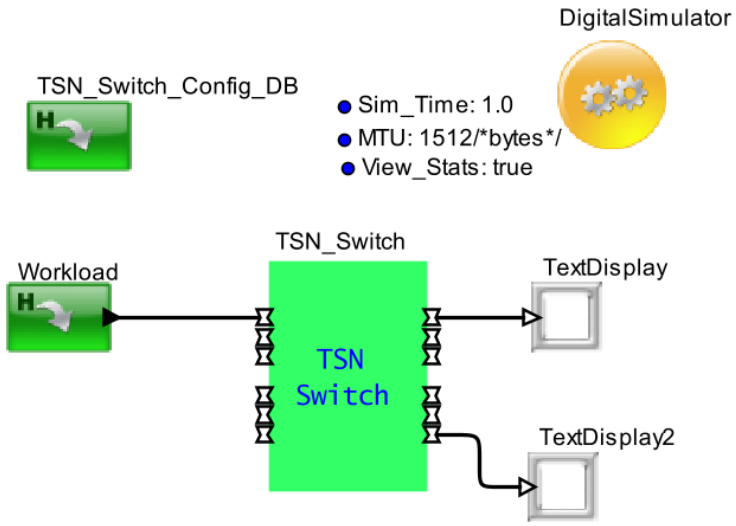
Power_Manager
This parameter is used to define the Power Manager name and enable power analysis to this block.

Enable_Debug
This enable or disables the timing diagram plot for TC10.

Enable_TC10
Enable or disables the TC10 TC10 Sleep/Wakeup functionality.



The following image shows a simple demo model of TSN to observe the data transfer and the impact of the TSN parameter changes.



13 TEthernet

13.1 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.

13.2 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 TTE-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.

13.3 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 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.

13.4 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.

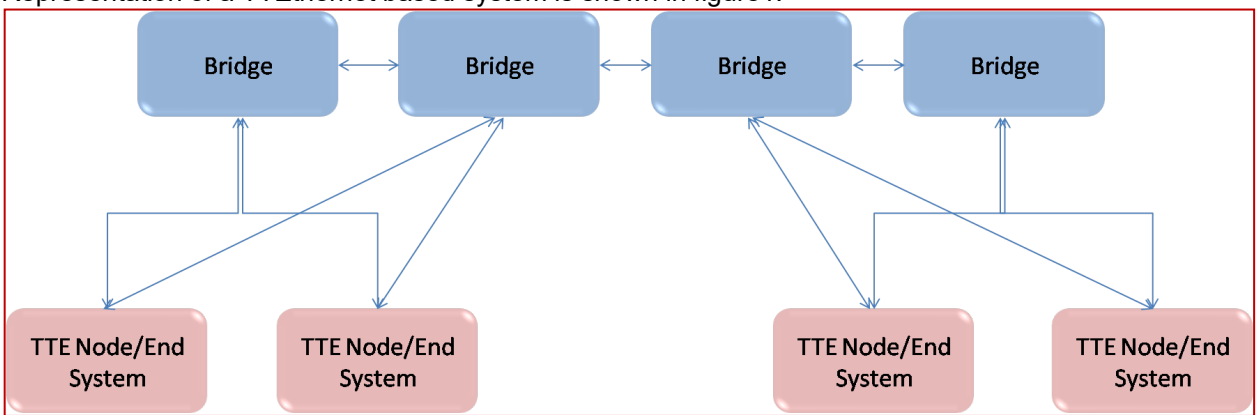


Figure 1: Block Diagram of a TTEthernet based System

VisualSim model of the proposed system architecture as defined in figure 1 is shown in figure 2 below.

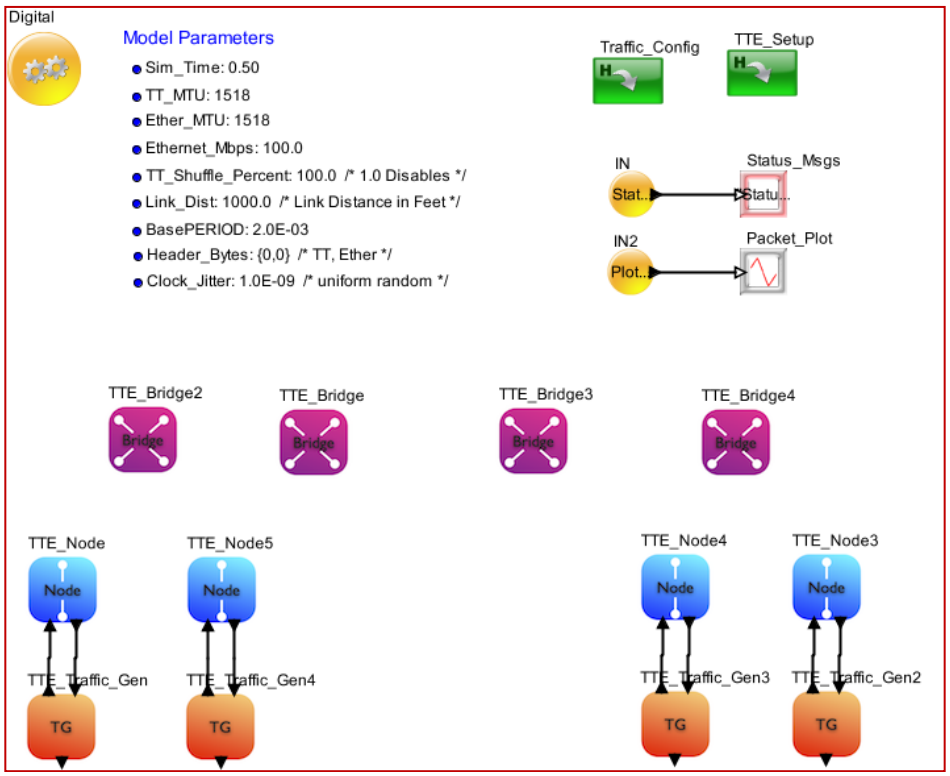


Figure 2: VisualSim Model

Please note that TTE_Node and TTE_Bridge blocks are connected virtually using named mapping.

13.5 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 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 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

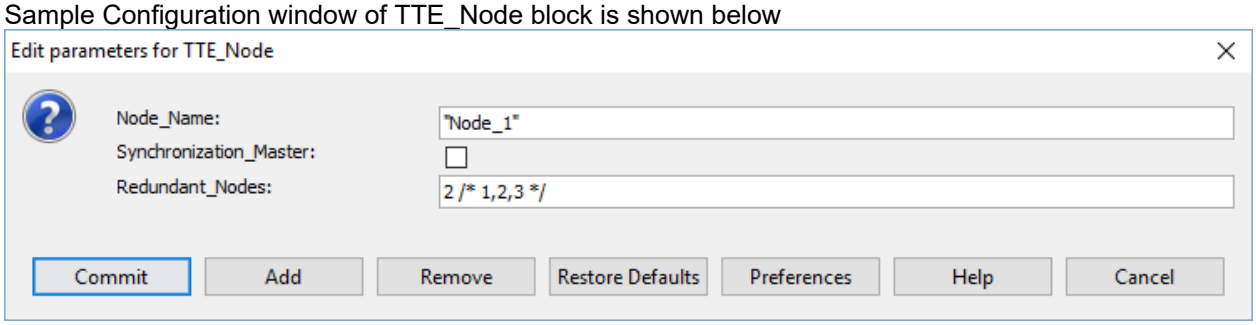
BasePERIOD:- Base period determines the communication cycle for the slots

Header_Bytes:- TTEthernet and Ethernet header bytes. First array position is for TTEthernet and the second position is for Ethernet.

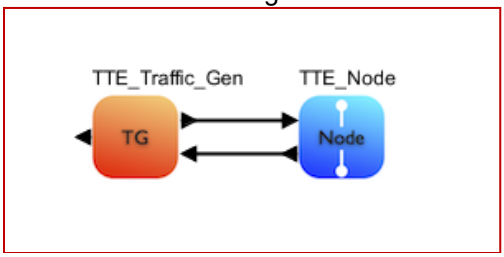
Clock_Jitter:- Clock Jitter value

13.6 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.

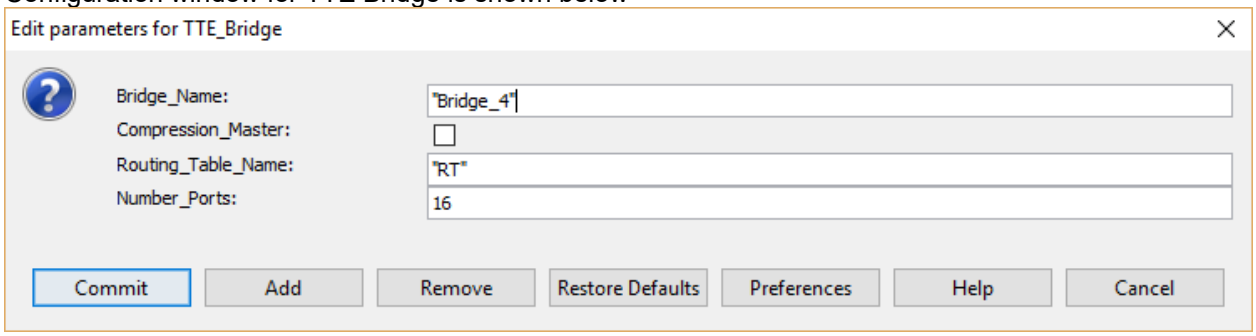


13.7 TTE Bridge

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 µseconds (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

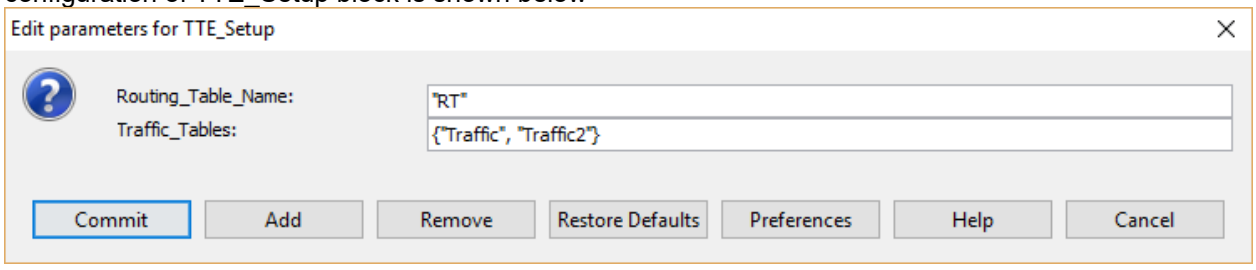


13.8 TTE Config

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

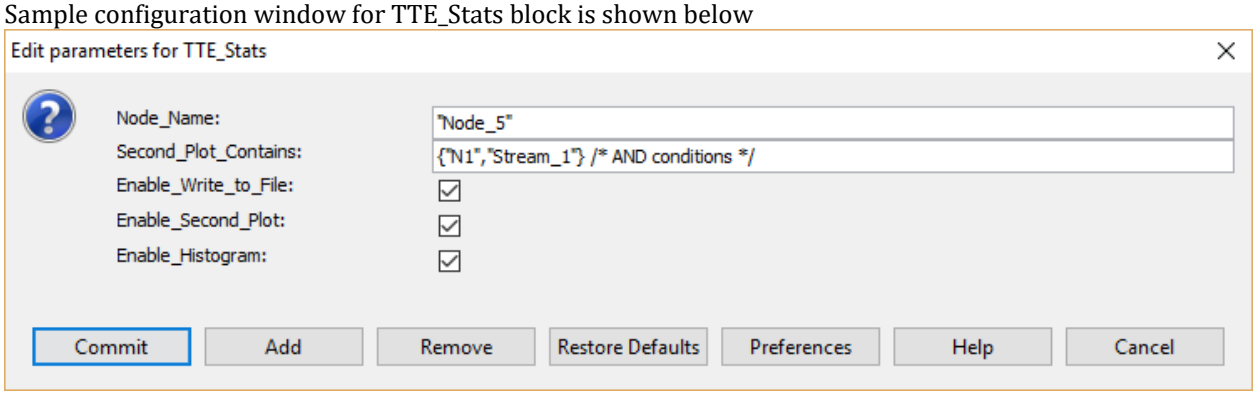
13.9 TTE_Setup

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



13.10 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



13.11 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 destinated 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

13.12 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:

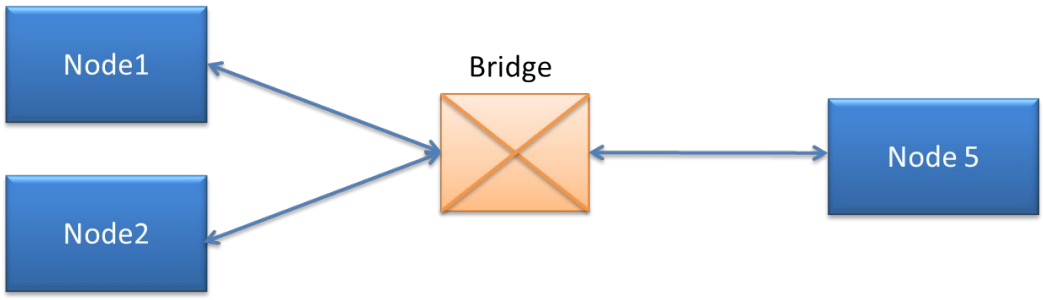


Figure 1.0 Block Diagram of Simple TTEthernet Network

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.

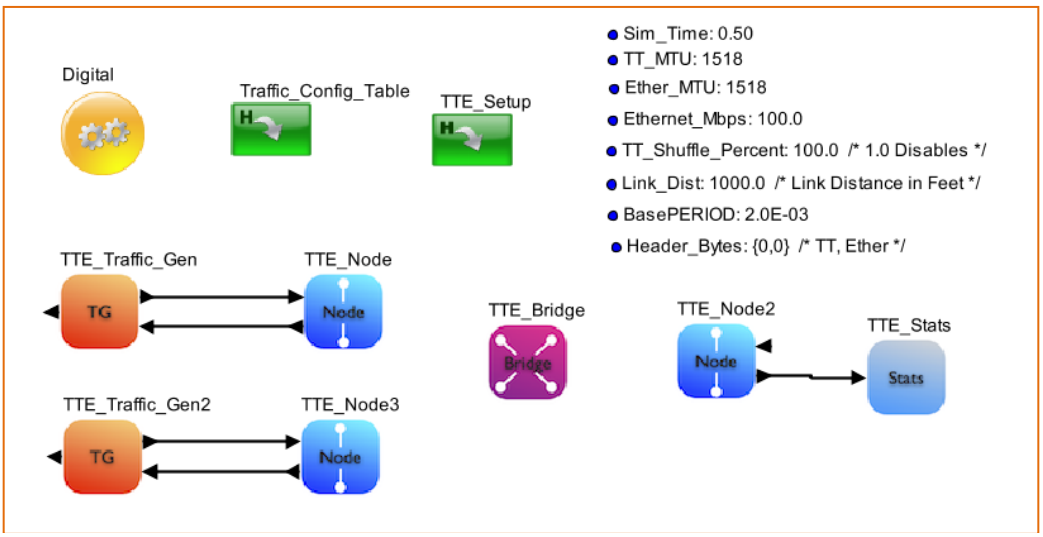


Figure 2.0 VisualSim Model

13.12.1 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.

13.12.2 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.

13.12.2.1 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.

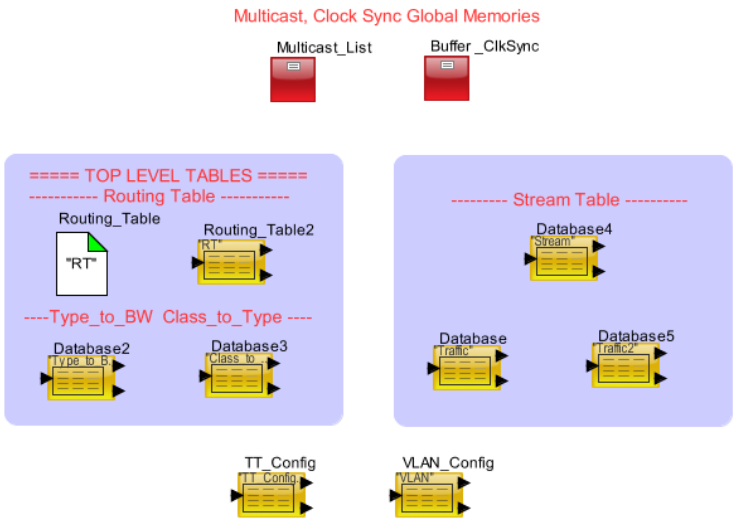


Figure 3: TTE_Config_Tables

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

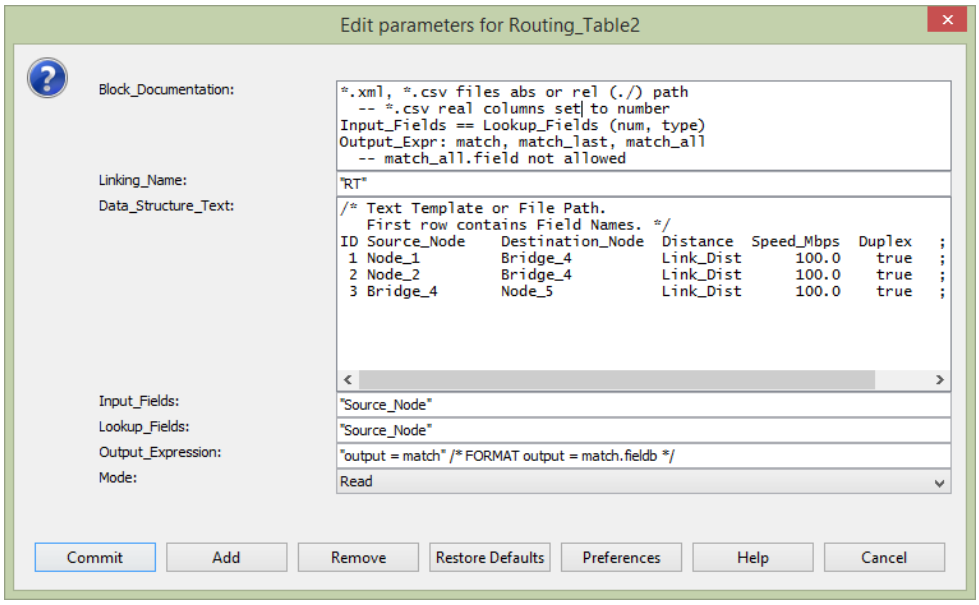


Figure 4: Routing Table Configuration

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
Protocol VLAN Type ;
 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 tailers
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 ;

13.12.2.2 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.
a. Instantiate the **Interfaces and Buses ->TimeTriggeredEthernet->TTE_Setup block.**
b. Link the Routing_Table_Name parameter to the top-level. This ensures that all blocks will use the same name. Example name is "RT"
c. In the parameter 'Traffic_Tables', list all the traffic tables names as strings in array.

13.12.2.3 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

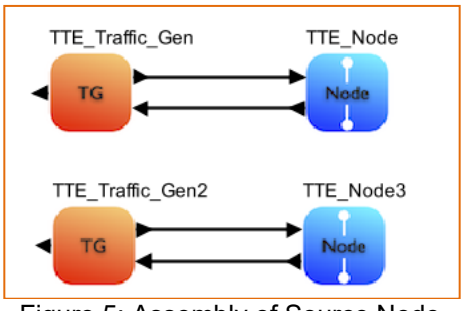


Figure 5: Assembly of Source Node

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

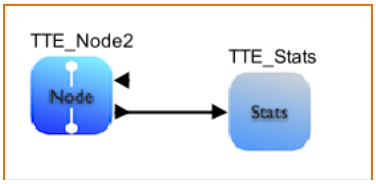


Figure 6: Assembly of Client Node/ Destination only node

13.12.2.4 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.

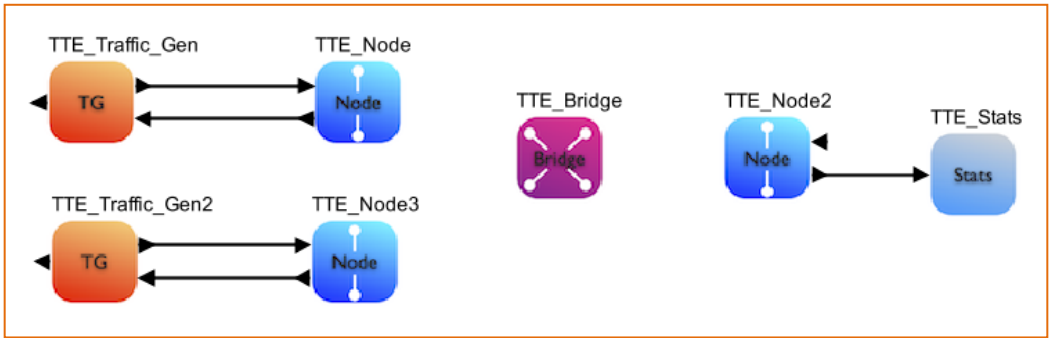


Figure 7 Single Bridge Network

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.

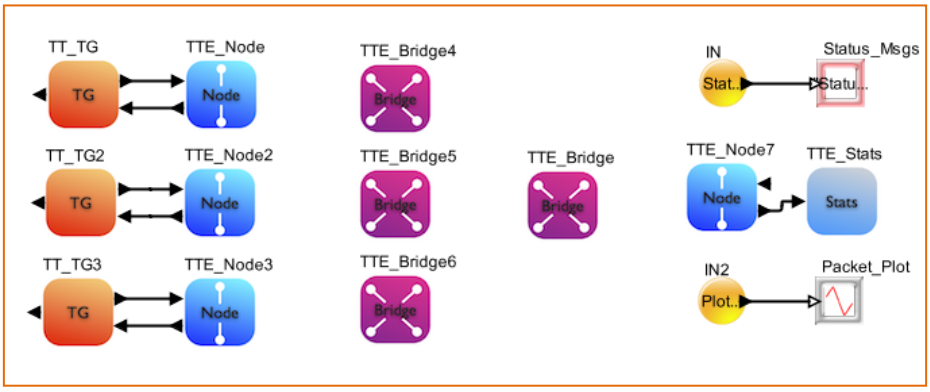


Figure8 Multiple Bridge Network

The following is the sequence:

a. Add as many **Interfaces and Buses ->TimeTriggeredEthernet ->TTE_Bridge** blocks as you have defined in the Link Configuration table

b. 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

c. For the parameter, add the **Routing_Table_Name**. For now it is just “RT”.

13.12.2.5 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 3different 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
N1_to_N5_TT1_1 **Latency** **Minimum: 2.5061367090 ms** **Mean: 2.5061367090 ms** **Std Dev: 29.1 ps** **Maximum: 2.5061367090 ms** **Throughput** **Mbps: 0.0328661952051**	**N2_to_N5_RC2_3** **Latency** **Minimum: 24.5134240 us** **Mean: 494.2624945 us** **Std Dev: 571.1953459 us** **Maximum: 2.0779767120 ms** **Throughput** **Mbps: 0.5011401721222**
Ethernet	
N1_to_N5_Ether_4 **Latency** **Minimum: 14.2734240 us** **Mean: 519.6828018 us** **Std Dev: 617.3499543 us** **Maximum: 2.8099000580 ms** **Throughput** **Mbps: 0.3010222764019**	

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.

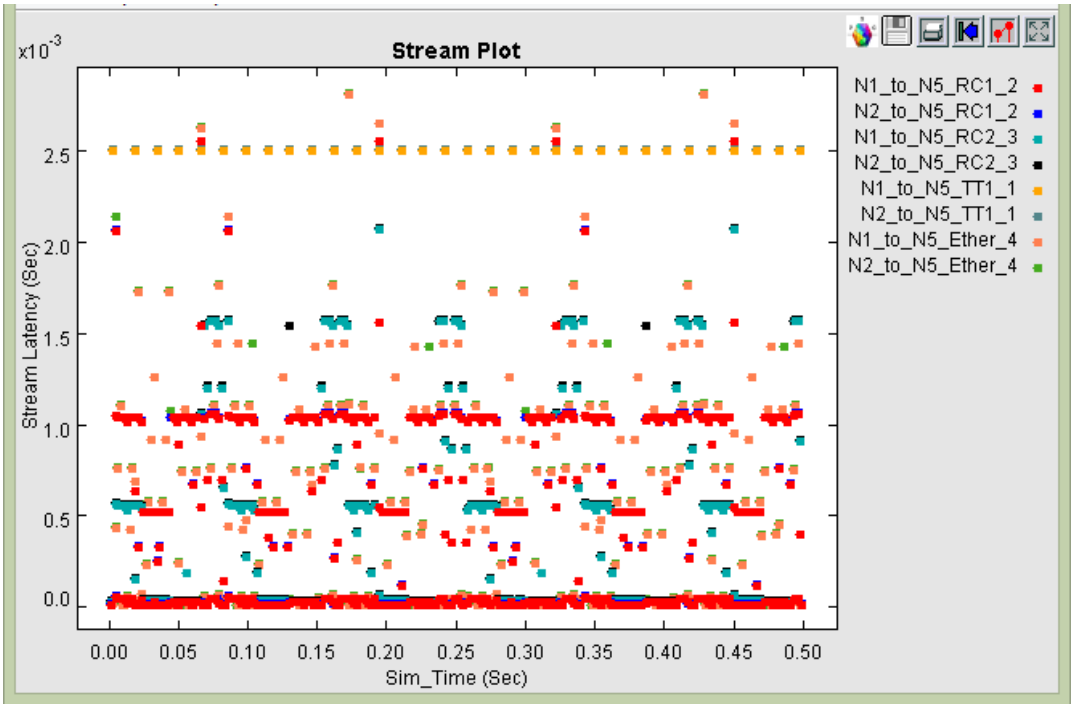


Figure 9: Stream Latency Plot

3. Histogram of the Latency for all the streams arriving at the Node to which this TTE_Stats block is connected.

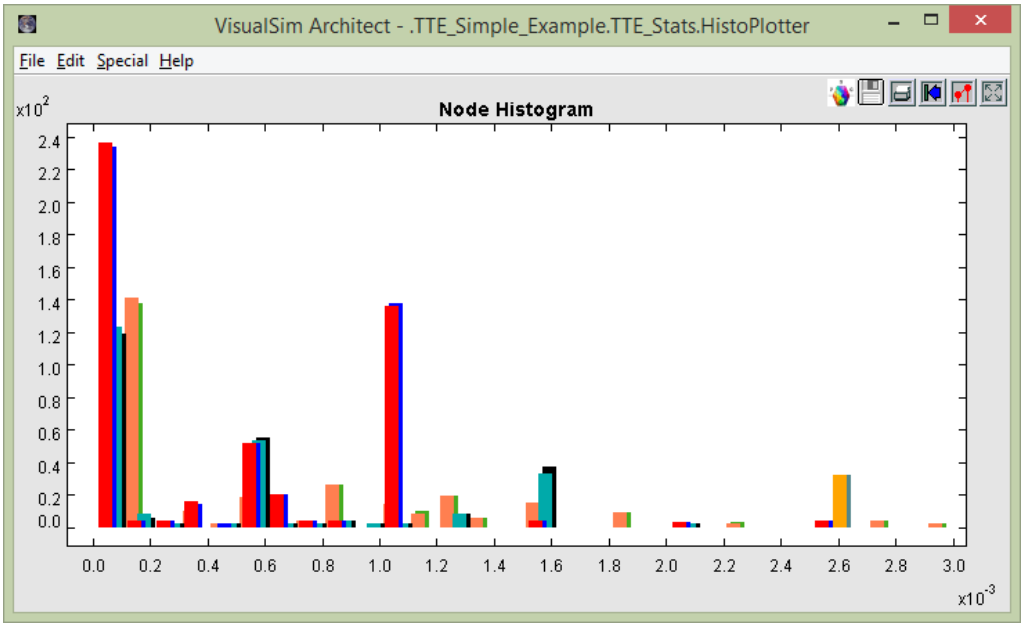


Figure 10 Histogram Latency plot

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.

13.12.2.6 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.

13.13 Advanced Tutorial

13.13.1 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.

13.13.1.1 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.

13.13.1.2 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

13.13.1.3 Step3: TTE_Setup

1. Update Traffic_Tables parameter with all additional Traffic_Table's names

13.13.1.4 Step4: Run Simulation

13.13.2 TTEthernet Model with 8 nodes and 3 Destination Nodes

In this tutorial we shall extend the model developed in advanced tutorial part 1.

13.13.2.1 Step1:

Open the model developed in Part-1 and instantiate additional destination nodes and TTE_Stats blocks. Provide unique names for TTE destination nodes.

13.13.2.2 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.

13.13.2.3 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

13.13.3 TTEthernet Model with 8 Nodes, 2 Bridges and 3 Destination Nodes

In this tutorial we shall extend the model developed in advanced tutorial Part2.

13.13.3.1 Step1:

Open the model developed in Part-1 and instantiate additional 1 TTE_Bridge block. Provide unique name.

13.13.3.2 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
;
```

13.13.3.3 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

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 **NODE**s in the network, or a **virtual connection** network of mapped **NODE**s. The **Routing_Table** Block provides a routing table, plus common routing parameters for a network of **NODE**s. 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.

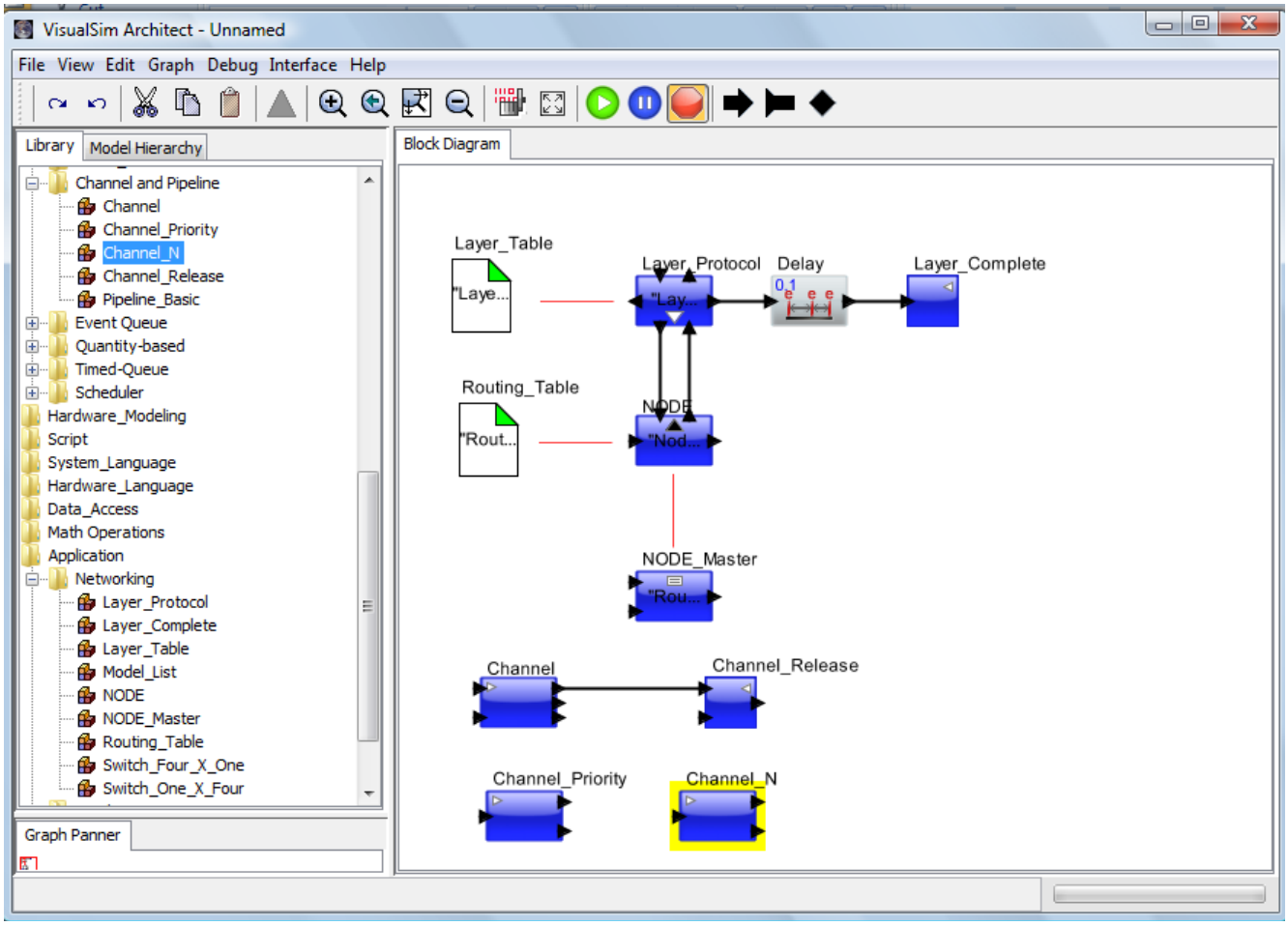


Figure 1-1 VisualSim Networking and Channel Blocks

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

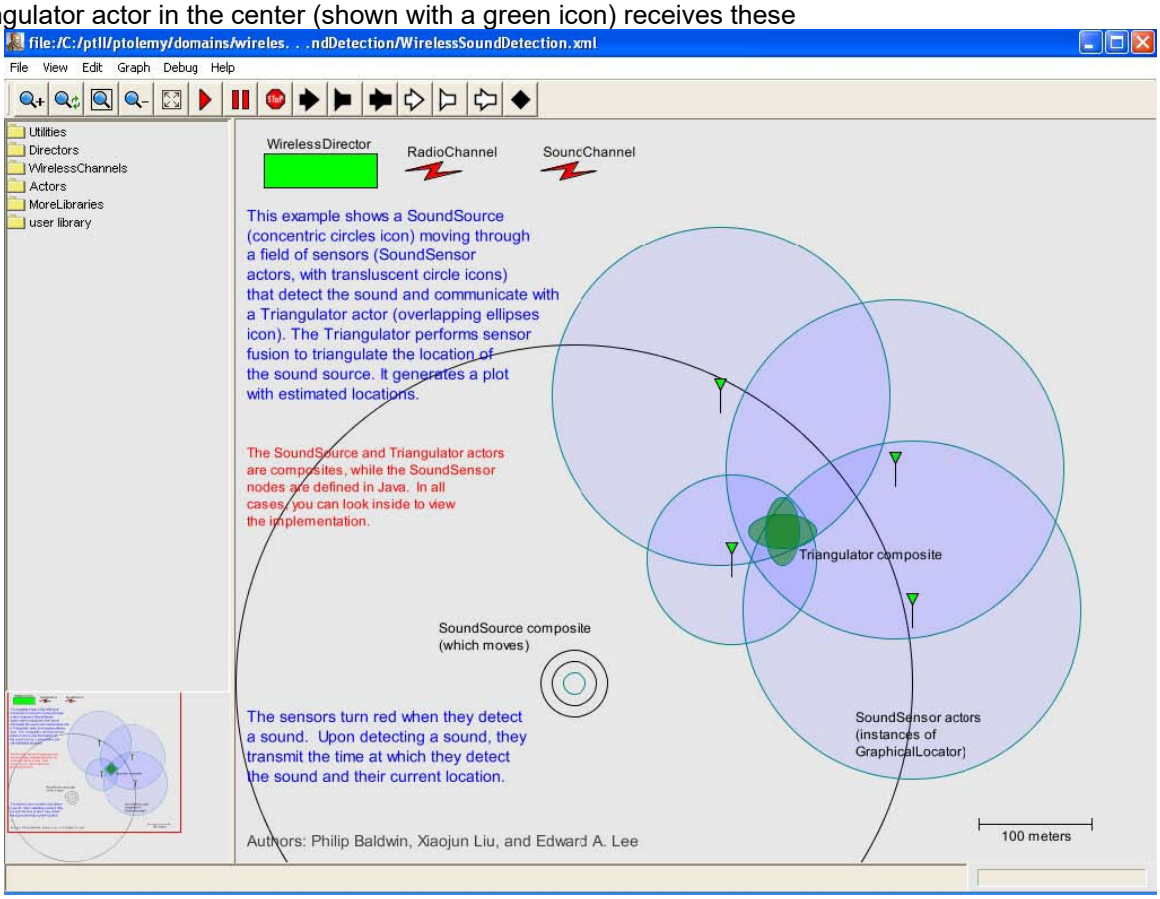


FIGURE 2. The VisualSim Wireless representation of a wireless sound detection model.

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.

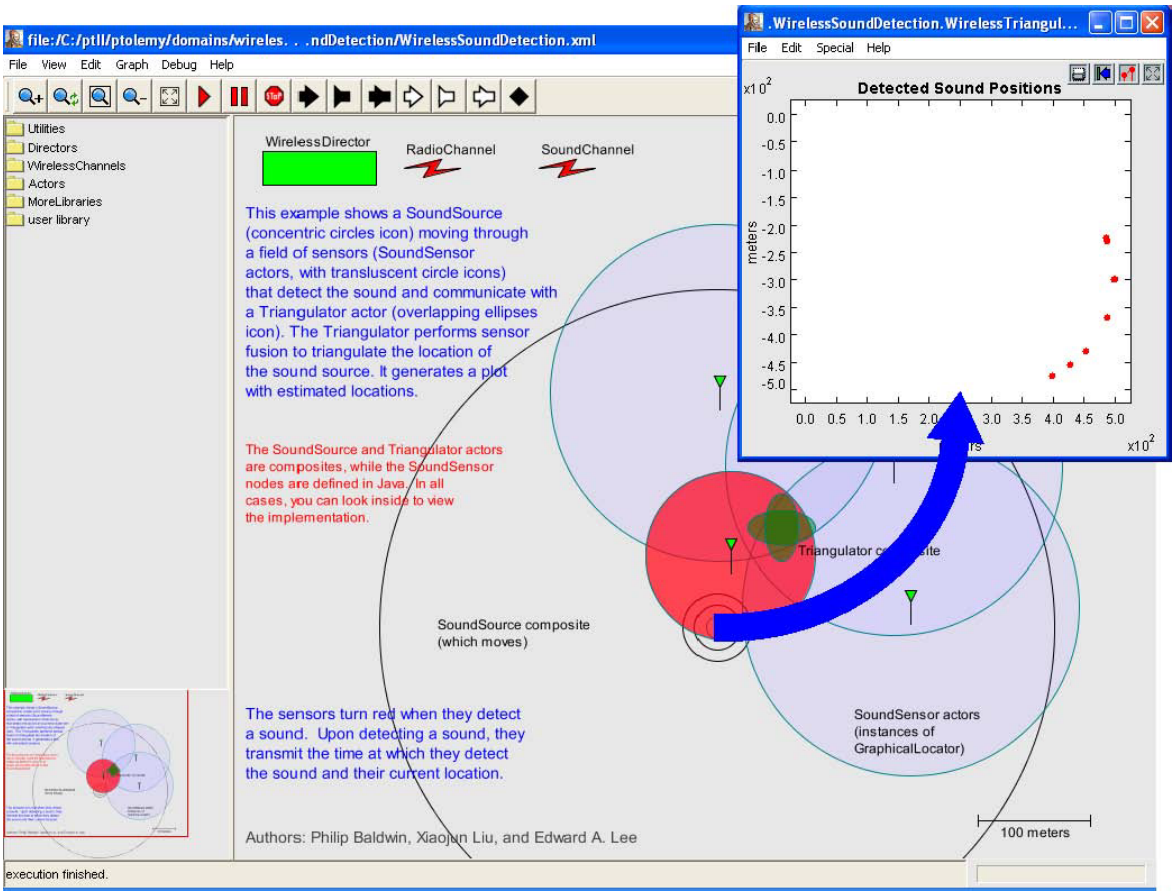


FIGURE 3. Animation as the model executes. The SoundSource actor moves in a circle through a field of Sound-Sensor actors. When these actors detect a sound, they transmit a radio signal to a Triangulator node, which estimates and plots (at the upper right) the position of the sound source.

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.

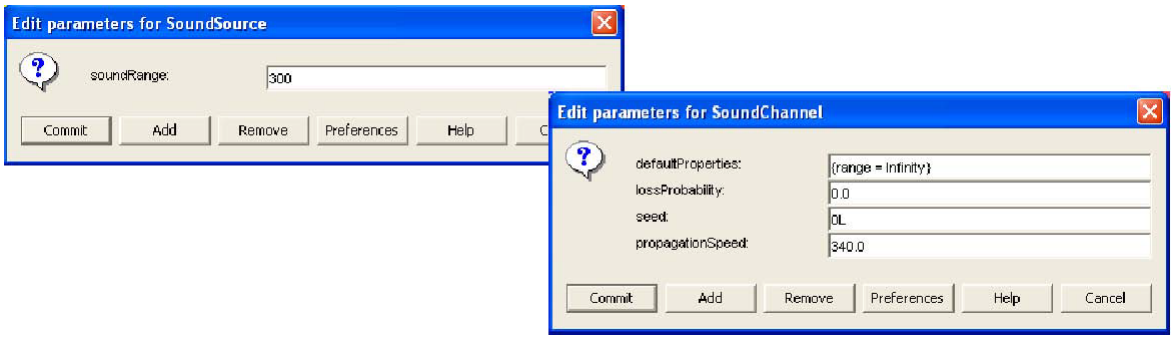


FIGURE 4. Parameters of the SoundSource actor (left) and SoundChannel channel model (right).

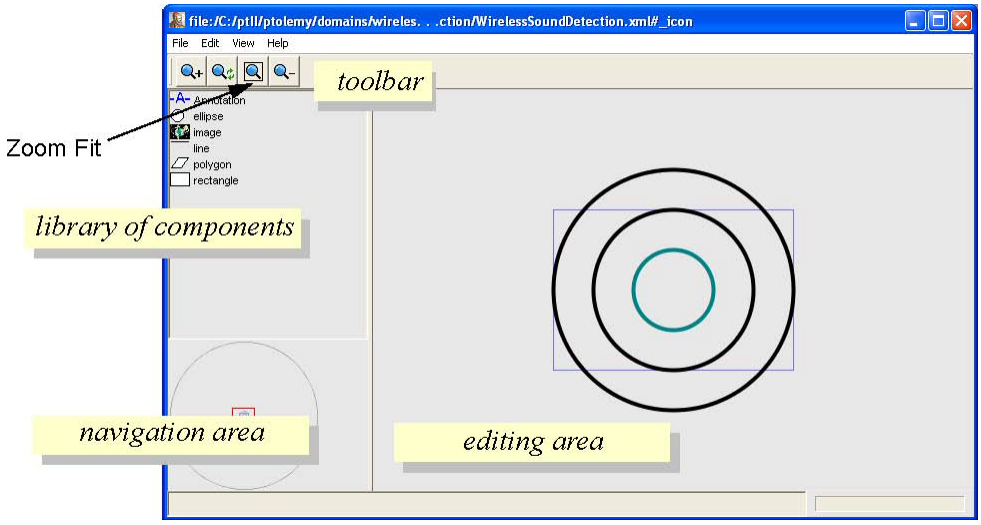


FIGURE 5. View resulting from selecting “Edit Custom Icon” after right clicking on the SoundSource in figure 2.

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.

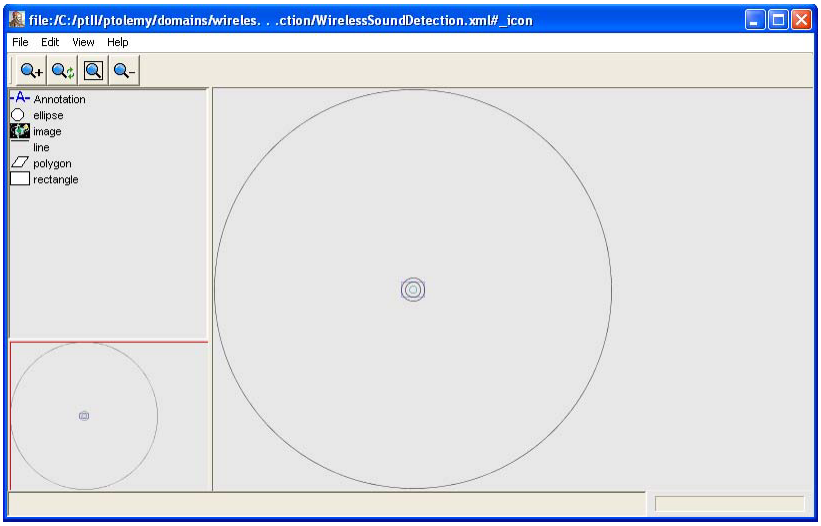


FIGURE 6. View resulting from clicking Zoom Fit in the toolbar of figure 5.

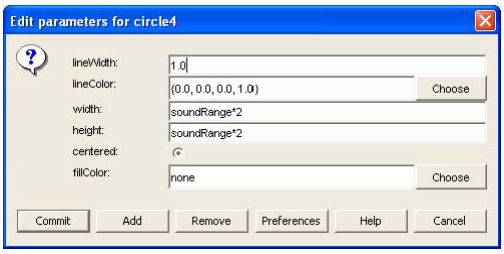


FIGURE 7. Parameters of the outer circle of the SoundSource actor icon in figure 5.

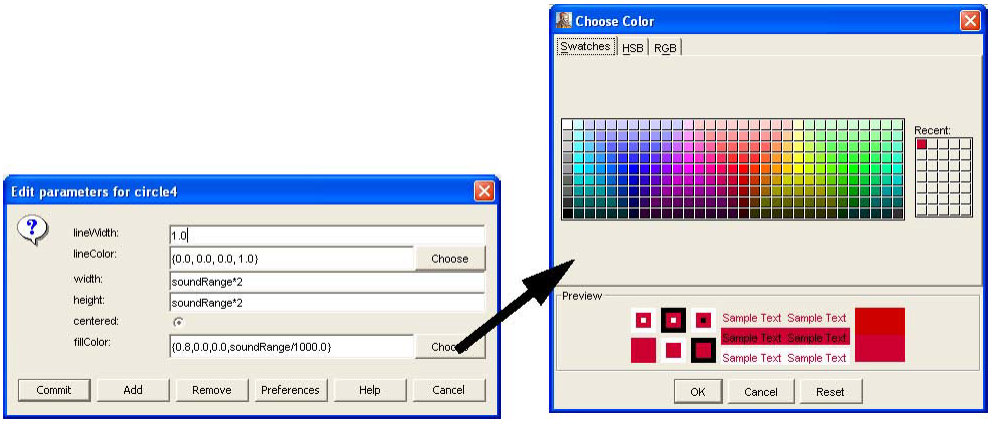


FIGURE 8. Setting the fill color of the outer circle of SoundRange to depend on its soundRange parameter.

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.

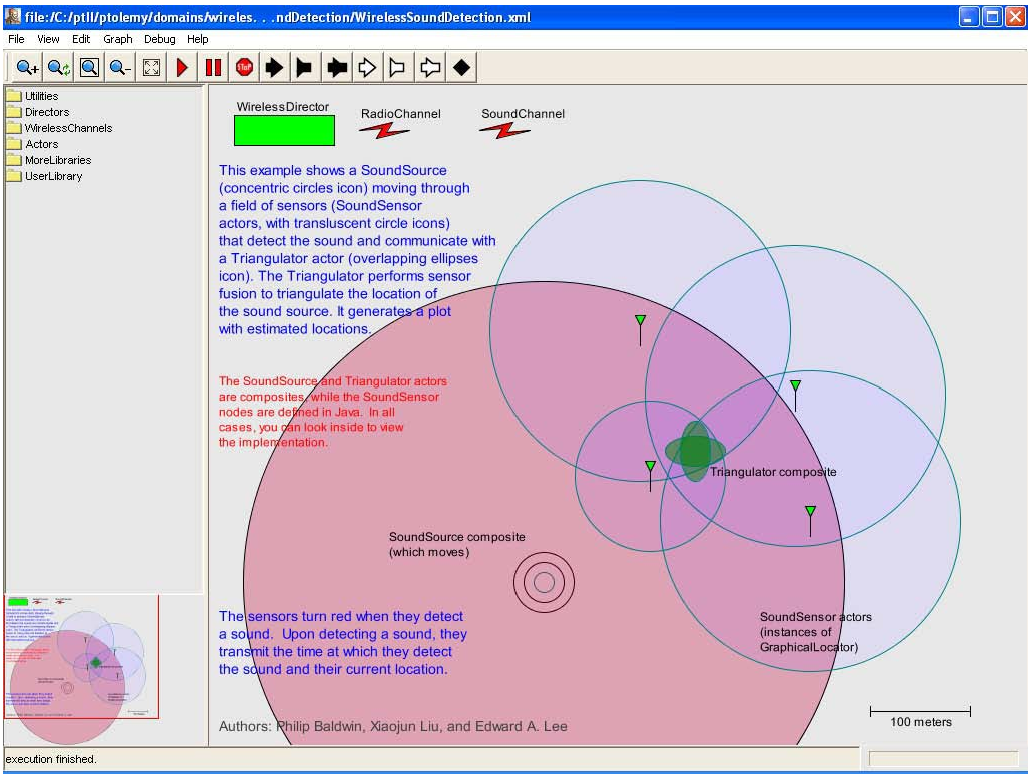


FIGURE 9. Result of changing the color of the outer circle of SoundRange as shown in figure 8.

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.

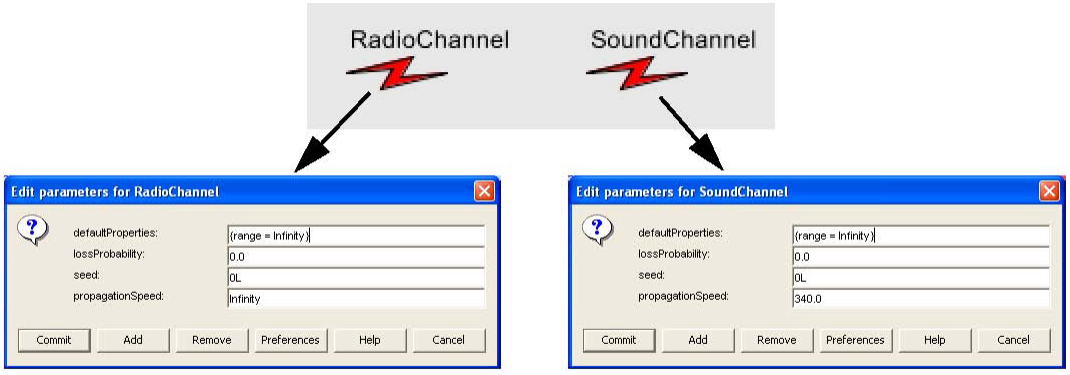
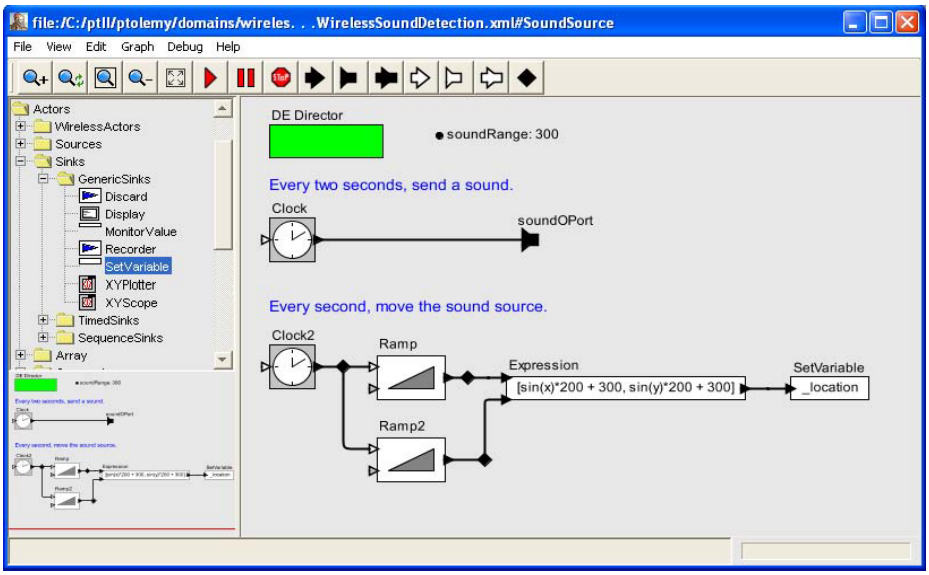


FIGURE 10. The channels of figure 2 and their parameters.



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.

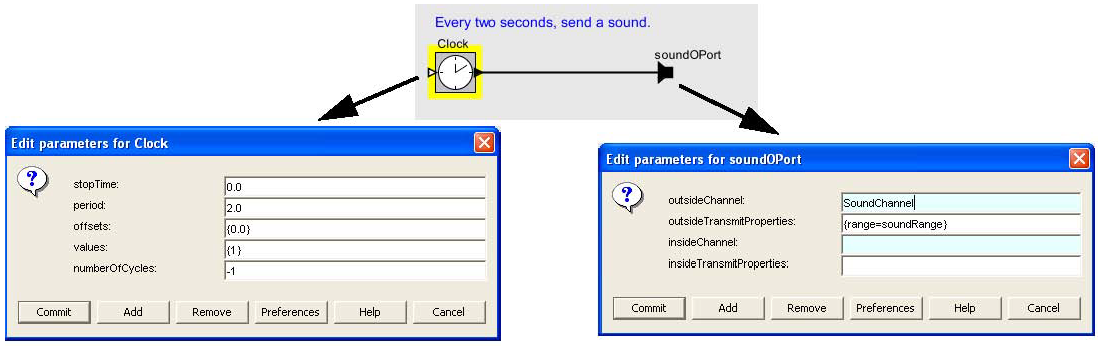
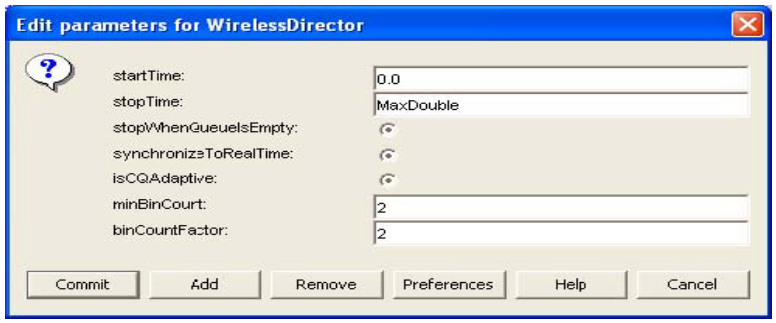


FIGURE 12. Portion of the composite in figure 11 that produces the sound event, with two parameter screens.



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 ErasureChannel, 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:

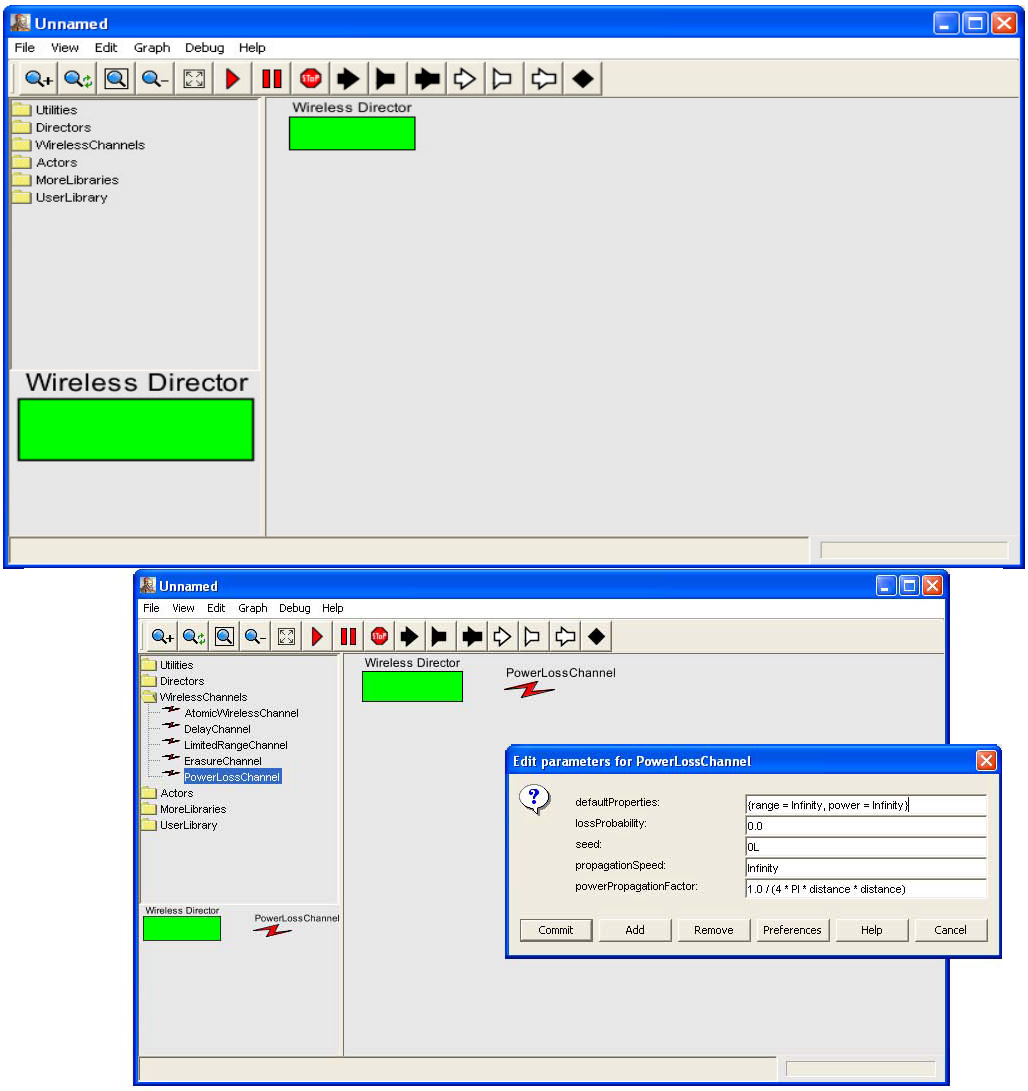
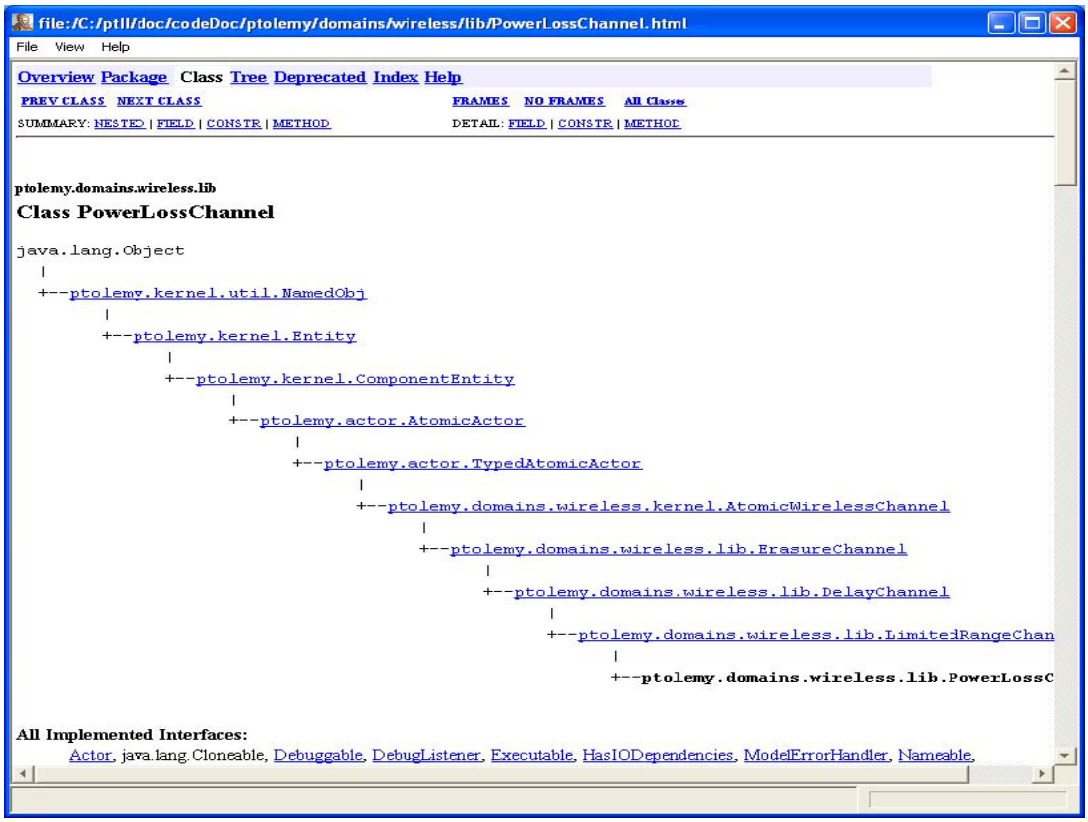


FIGURE 15. New model populated with a channel.

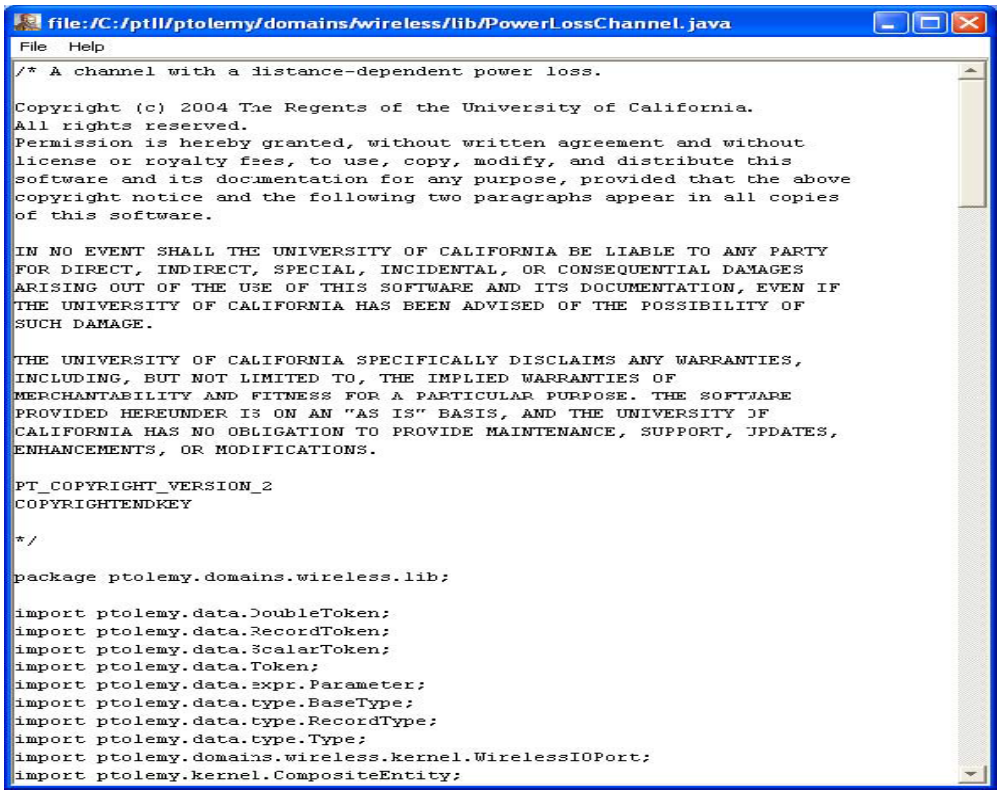


“The power propagation is given as an expression that is evaluated and then multiplied by the power field of the transmit properties before delivery to the receiver. For convenience, a variable named “distance” is available and equal to the distance between the transmitter and the receiver when the power 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 * PI * distance * 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.



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."

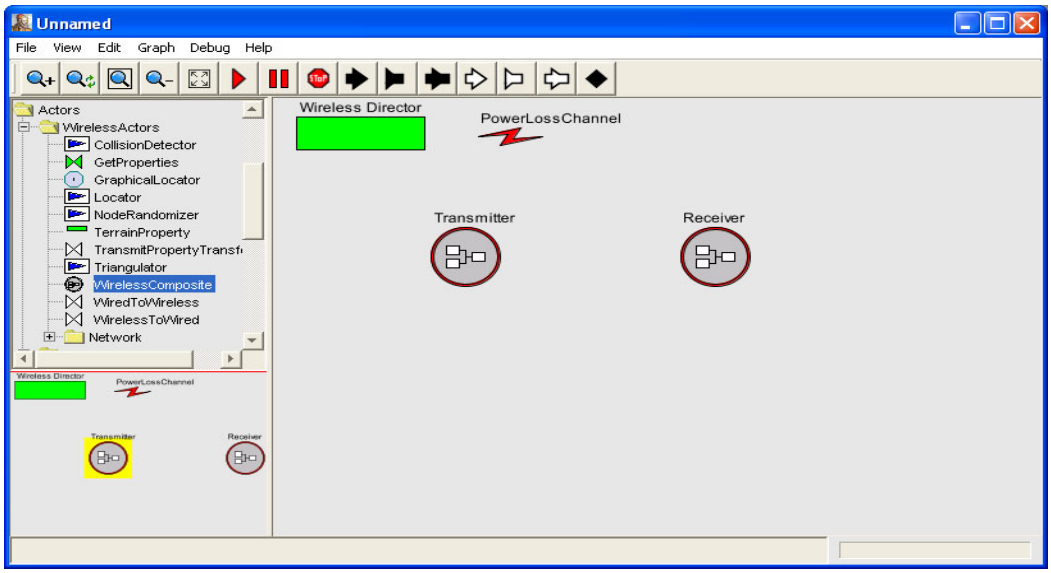
Hopefully, this makes it reasonably clear how to use these parameters. Let us build a model that uses them.

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.

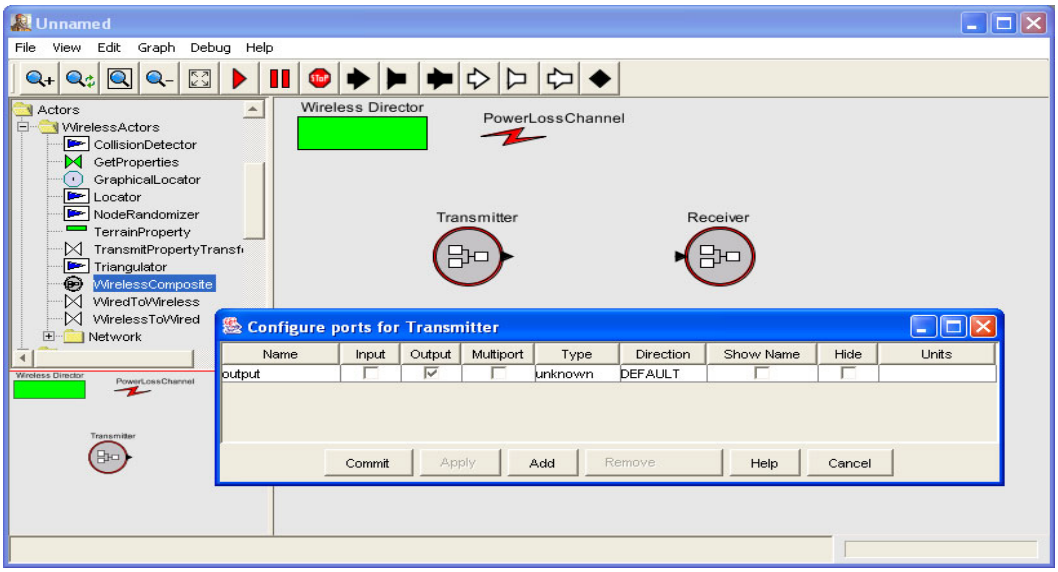
The PoissonClock actor will produce events at random times, where the time between events is obtained from an exponential random variable with mean given by the meanTime parameter of the PoissonClock. The default value is 1.0, which is fine for our purposes. If you return to the top-level window and 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.

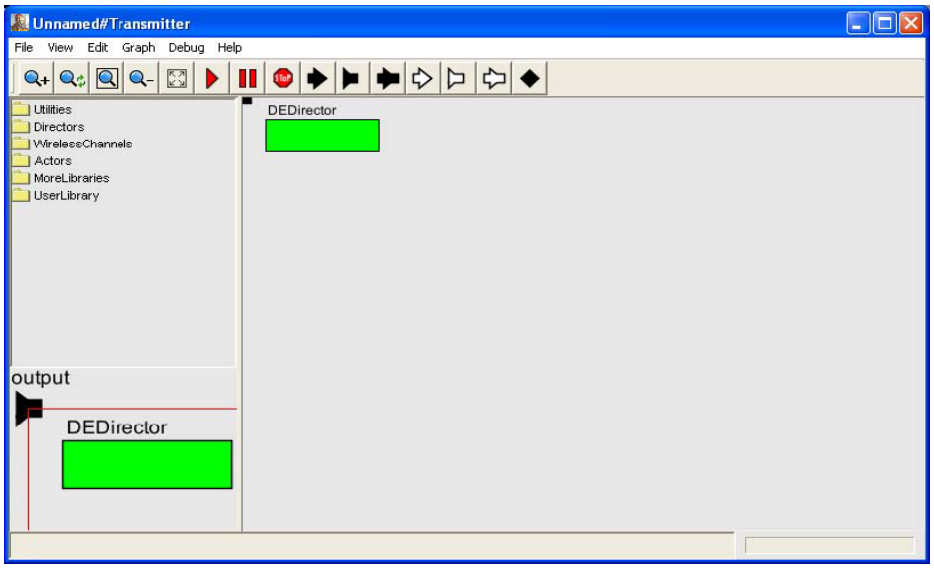


Look inside the Receiver actor and build the model shown in figure 22. The Ramp actor is found in the Actors library under Sources→Clocks, and the Display actor is found under Results→Plotter, as shown on the left in the figure. The model is now ready to execute. Clicking on the red triangle in the toolbar 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.

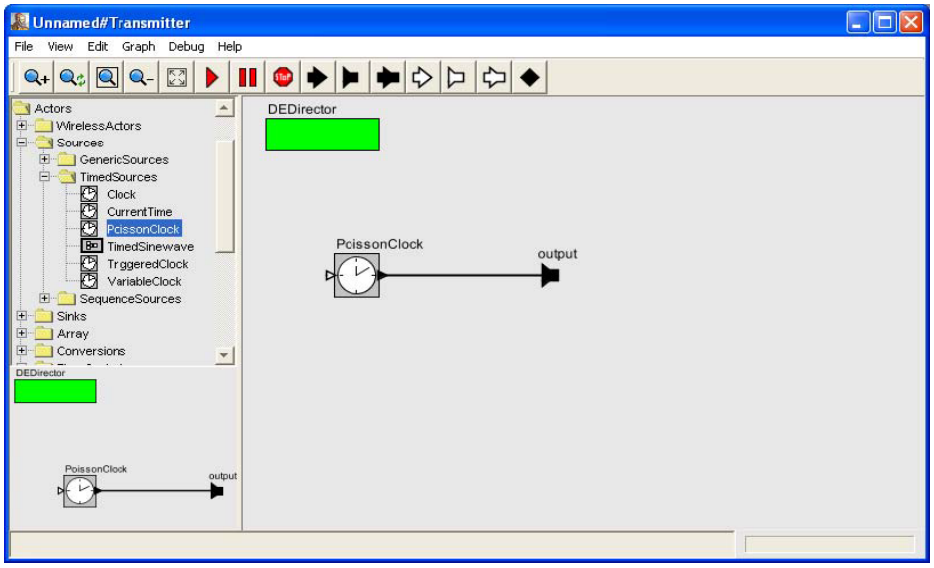
You may want to save your model using the File→Save menu command. Use the file extension .xml (or .moml) to ensure that VisualSim Wireless will recognize this as a model file. Notice that the title bar on the window now reflects the name of your model, which is the same as the name of the file.

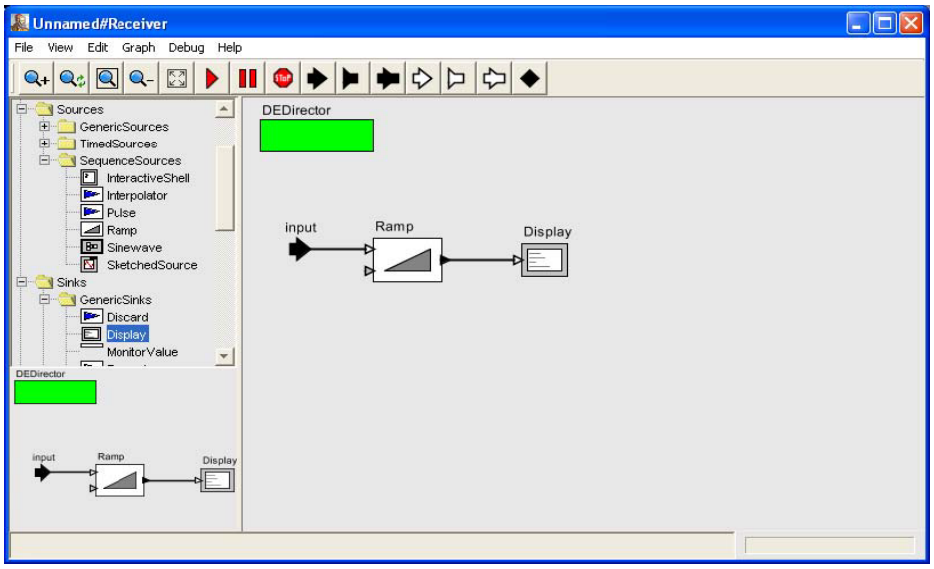
Let us modify this model so that the power loss of the channel as a function of distance is observed. To do this, find the GetProperties actor in the Full Library → Application→Wireless library, and replace





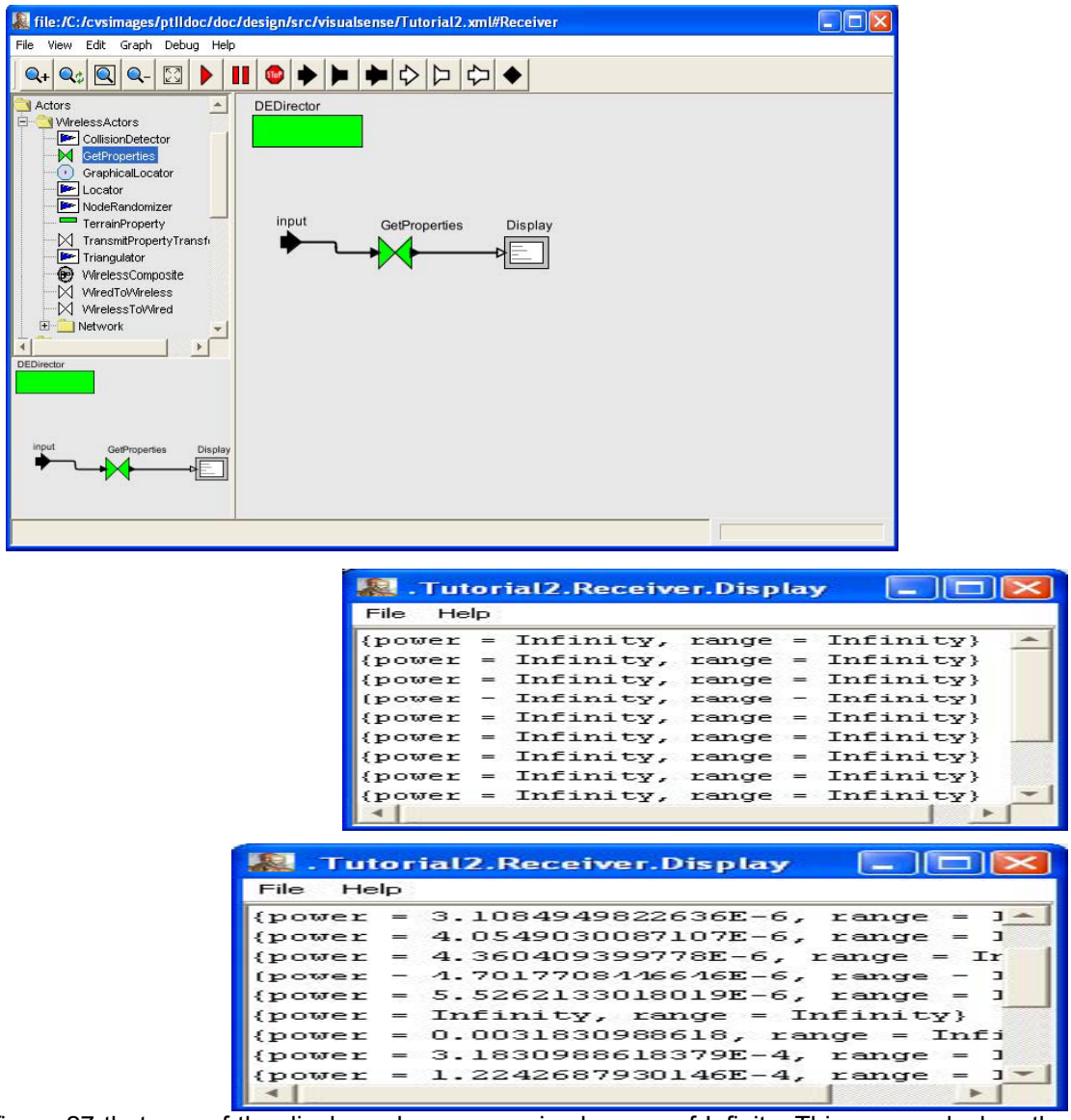
the Ramp block inside the Receiver as shown in figure 24. Running the model now results in the display shown in figure 25. Notice that the received power is always Infinity, which is not very useful.





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

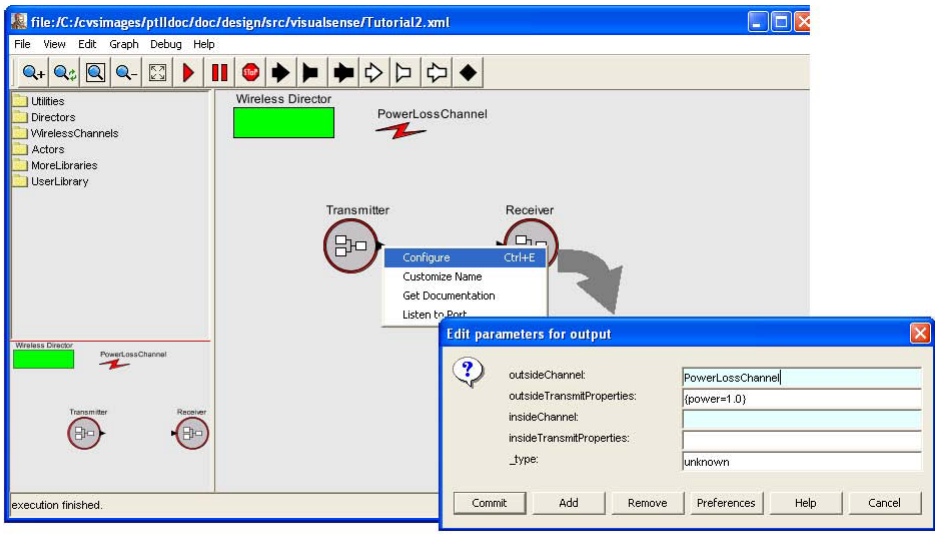
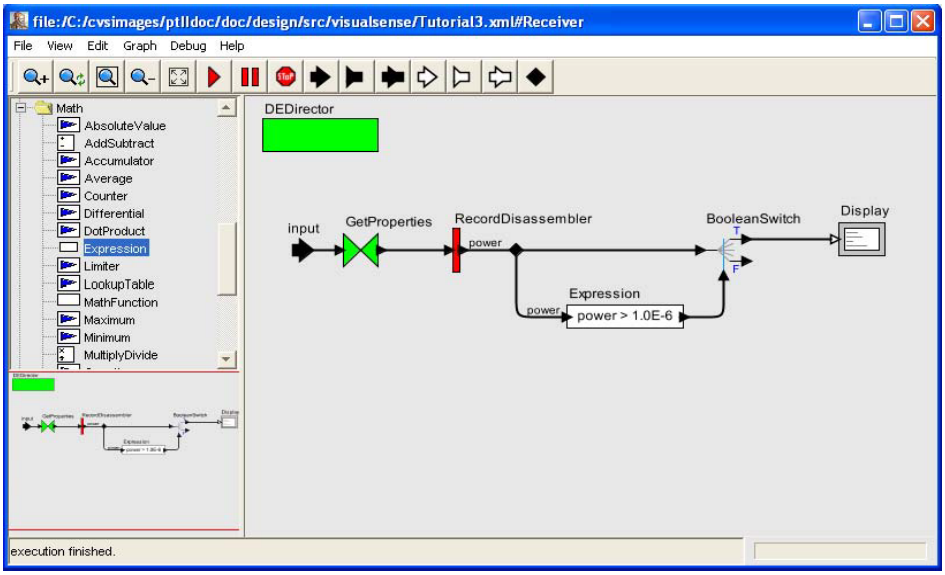


FIGURE 26. Setting the transmit power of the Transmitter.



Expression: Math
BooleanSwith: 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}.

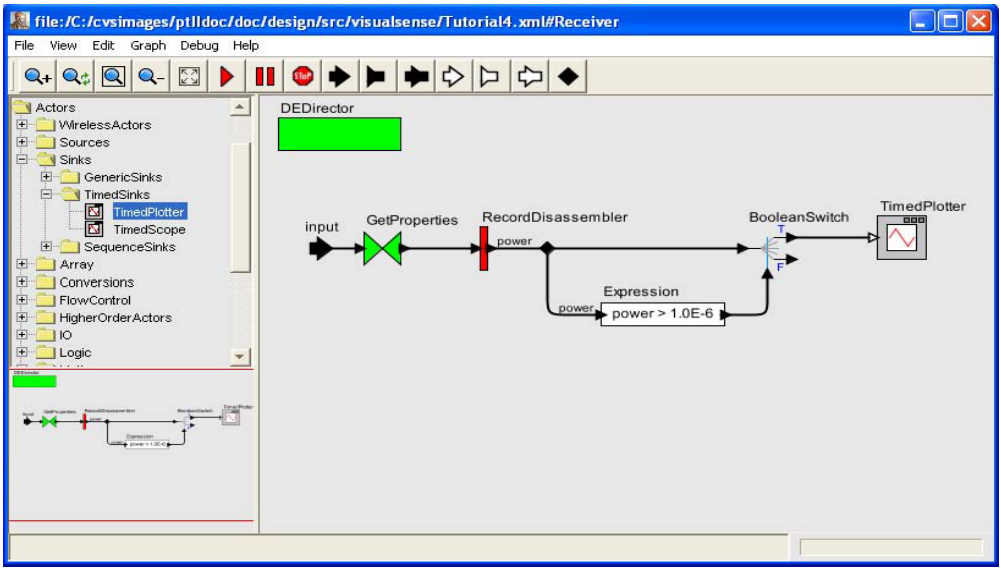
Notice that in figure 28, some connections involve a small black diamond. This is the visual mechanism for routing a signal to multiple places. To create the diamond (which is called a vertex), you can either control click on the background of the editor, or click on the black diamond in the toolbar. 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.

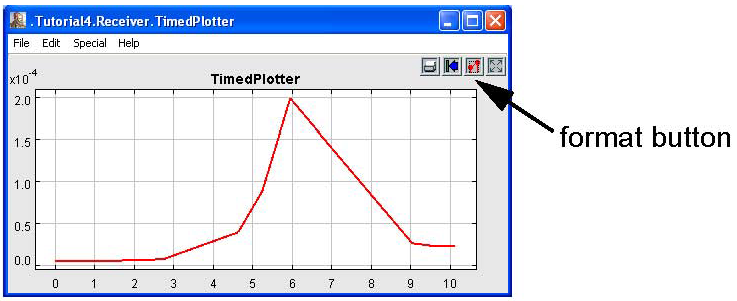
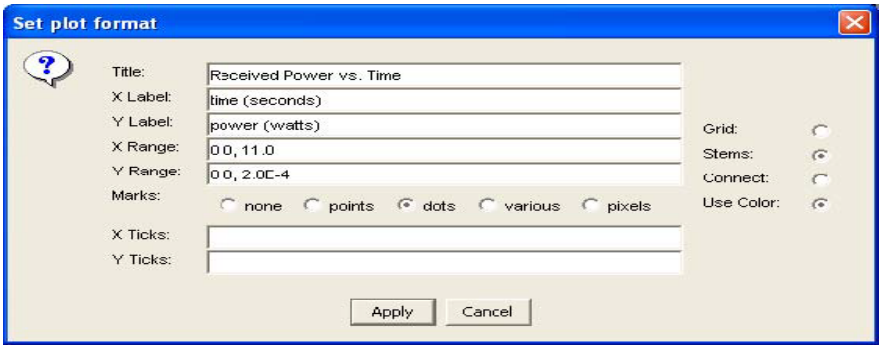
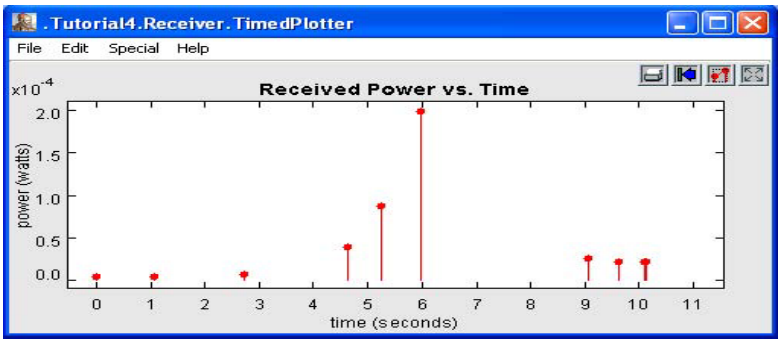


FIGURE 30. Plot showing the received power a function of time as the Receiver is moved close to the Transmitter.





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.

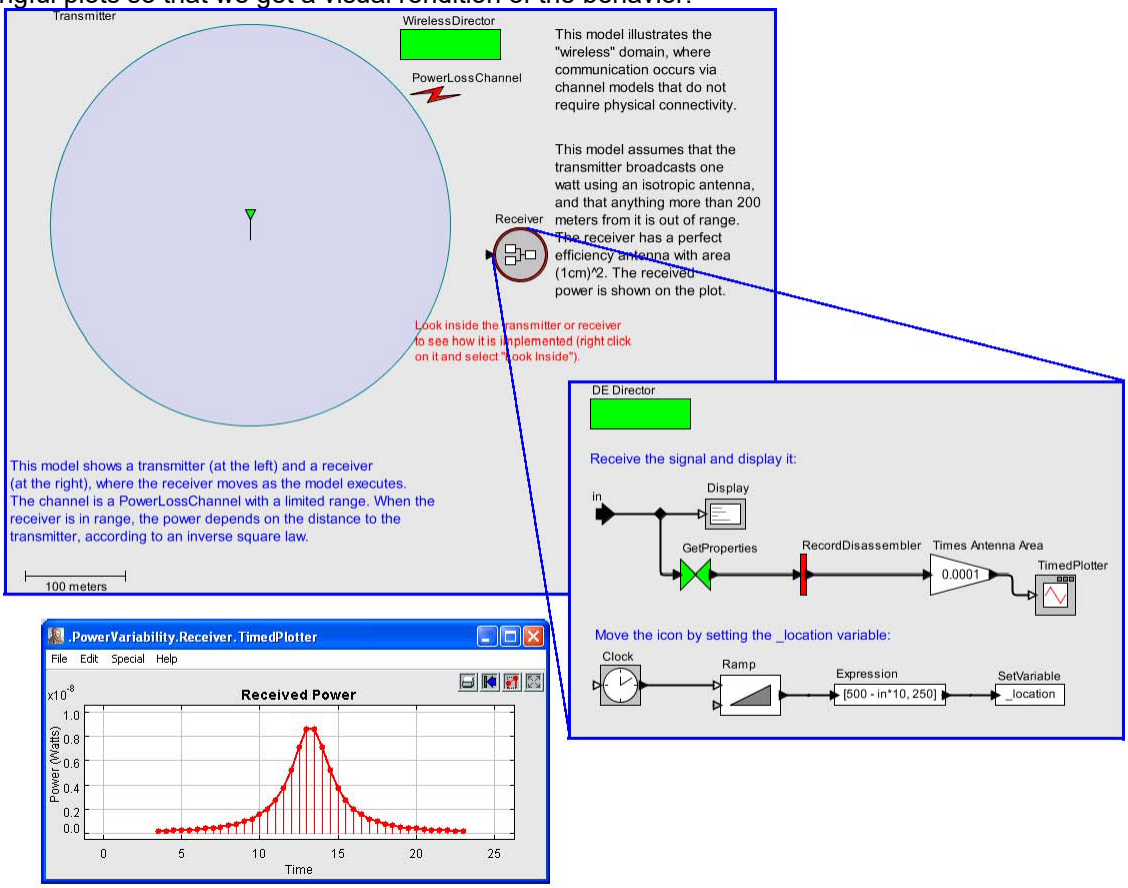


FIGURE 33. Model of power loss as a receiver moves into range and then close to a transmitter.

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.

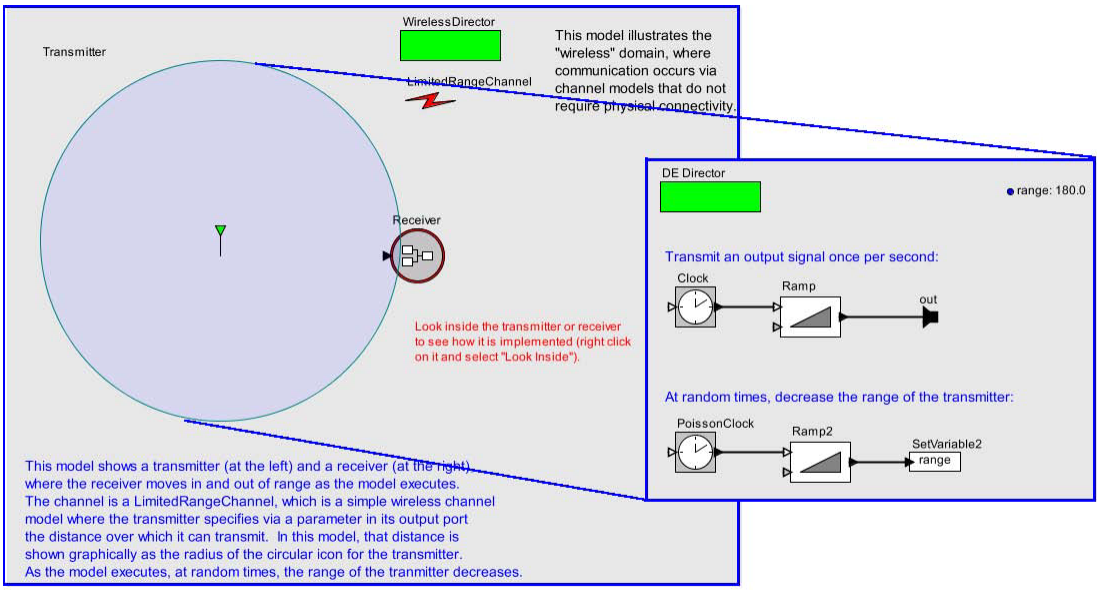


FIGURE 34. Model where transmission range degrades over time as a battery is depleted.

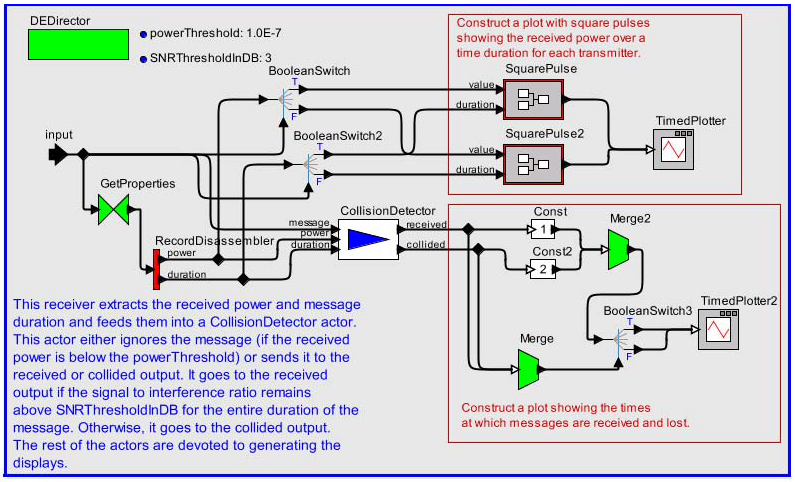


FIGURE 36. Implementation of the Receiver in figure 35, which models and tracks collisions.

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-

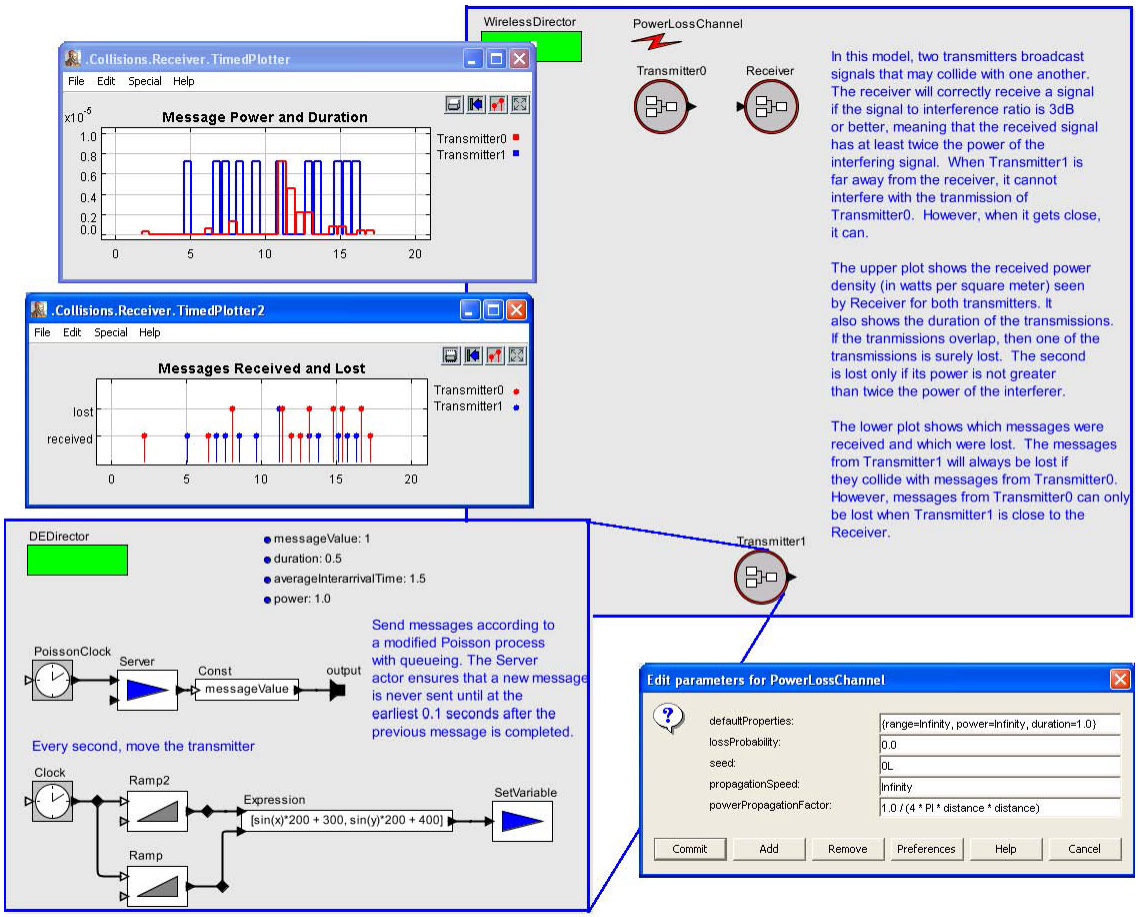


FIGURE 35. Model of collisions of messages that take time.

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

$$p_i()t \geq P \sum p_j()t \quad (1)$$ $j \neq i$ for all t where $p_i()t > 0$, where P = 10^(SNRThresholdInDB/10), 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 Transmit-PropertyTransformer 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.

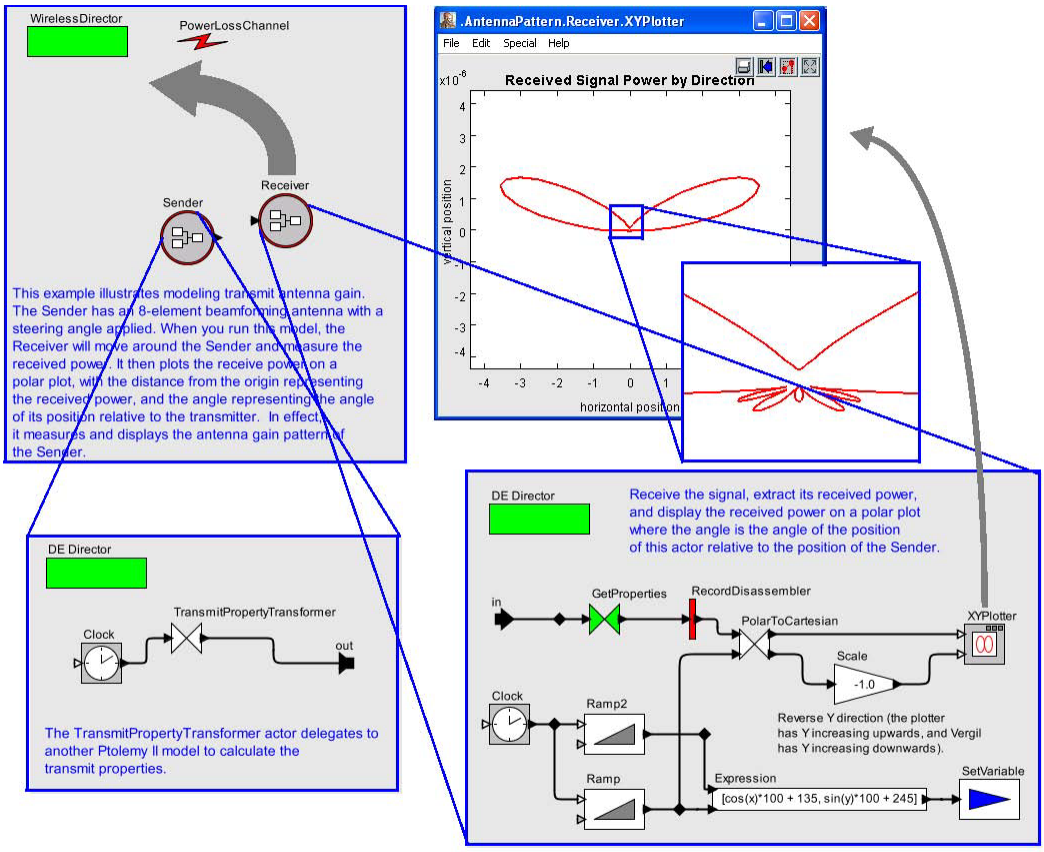


FIGURE 38. Model that includes a directional transmit antenna. As the model executes, the Receiver actor moves in a circular pattern around the transmitter and measures and plots the received power.

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.

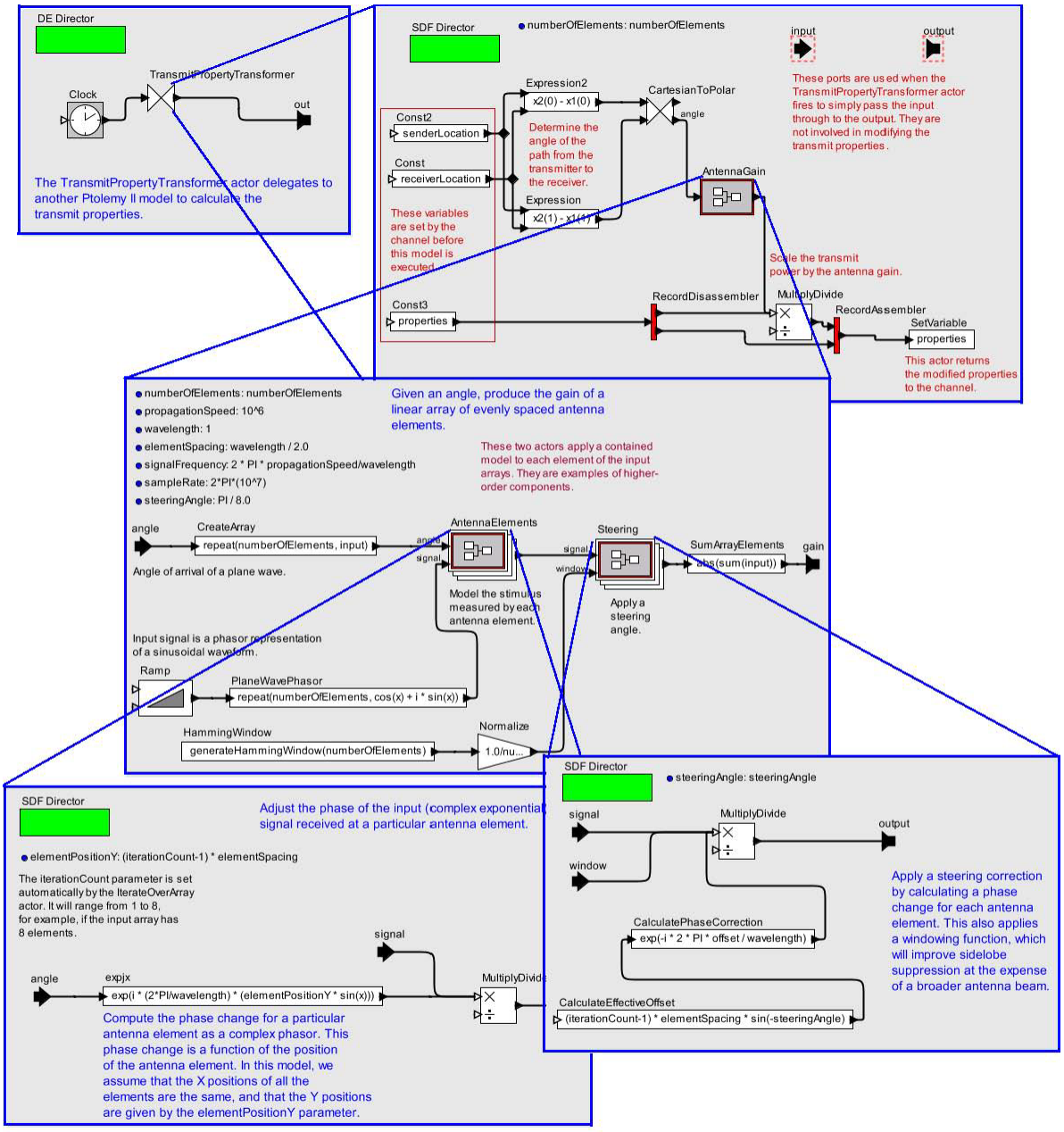


FIGURE 39. Transmitter design for the model in figure 38, showing how a Ptolemy II model (in this case a *synchronous dataflow* model) can be used to model transmission effects.

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} union {t_i} union {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 blockcalled 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.

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*log10(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 2^{order}.
3. **IFFT** (extends SDFTransformer): An inverse fast Fourier transform of size 2^{order}.
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 an 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 an 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 IllegalActionException 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, IllegalActionException {
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, IllegalActionException {
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 IllegalActionException If there is no simulator.
*/
public void fire() throws IllegalActionException {
    if (input.hasToken(0)) {
    Token in = input.get(0);
    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 objects[2].

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, IllegalActionException {
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 use 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 an 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, IllegalActionException {
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 (IllegalActionException ex) {
        // Should not be thrown because this is an output port.
        throw new InternalErrorException(ex.getMessage());
    }
}
public boolean prefire() throws IllegalActionException {
    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 an 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 IllegalActionException, 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, IllegalActionException {
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 IllegalActionException {
    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, IllegalActionException {
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 IllegalActionException {
    // 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 method returns 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 FooIcon.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 transluscent. 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

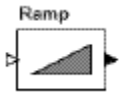


Figure 1-12 the Ramp icon.

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, IllegalActionException  {
        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");
    }

    ...

}
```

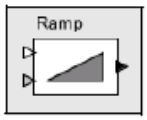


Figure 1-13 The Ramp actor defines a custom icon as shown.

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
rect	*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*
circle	*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*
ellipse	*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*
line	*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*
polyline	*points: List of x,y pairs of points, vertices of line segments, delimited by commas or spaces style: stroke, stroke-width*
polygon	*points: List of x,y pairs of points, vertices of the polygon, delimited by commas or spaces style: fill, stroke, stroke-width*
text	*x: horizontal position of the text y: vertical position of the text style: font-family, font-size, fill*
image	*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*

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 hen possible because they provide convenient arguments of type Nameable that identify the source f 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 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 IllegalActionException(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 Eclispe Debugger Setup

Create a new Java Project, to create a new java project click on

File → New → Java Project

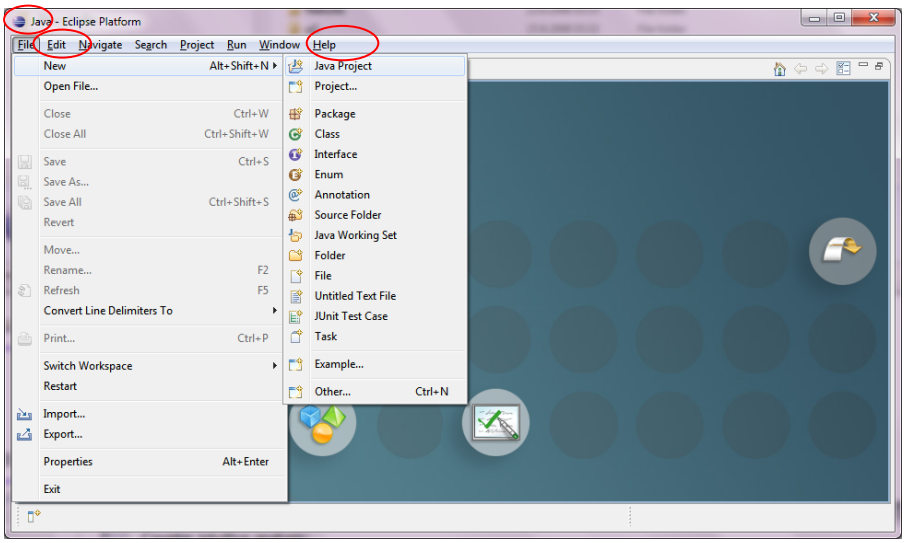


Figure 1.9.1

Set the project name and JDK as shown in the figure 1.9.2

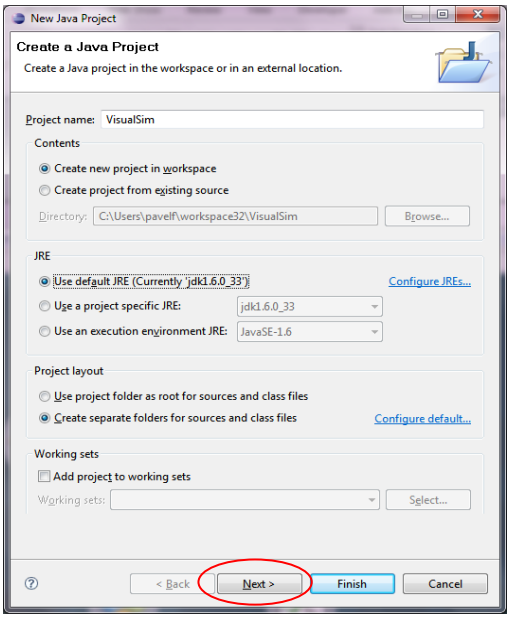


Figure 1.9.2

Add all the required external libraries, including any Apache and Java libraries on "Libraries" tab as shown in the figure 1.9.3

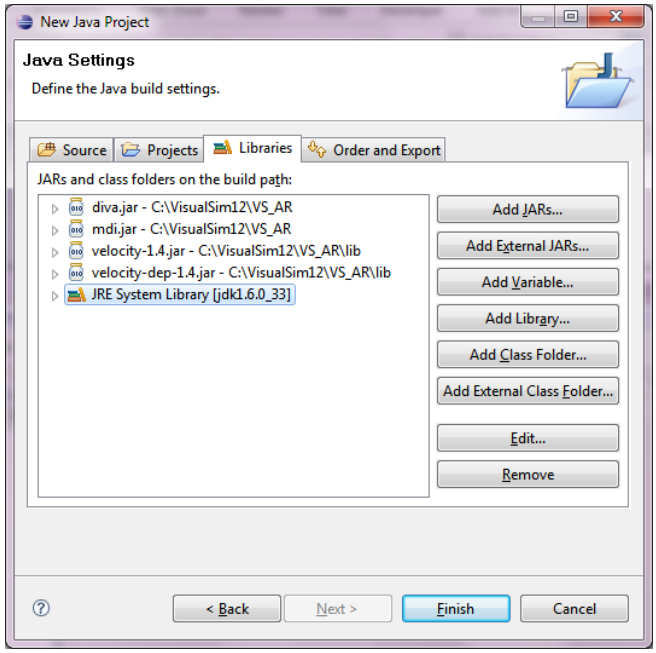


Figure 1.9.3

Using "Order and Export" tab on Eclipse debugger set the proper order of libraries. Source files should be placed on top.

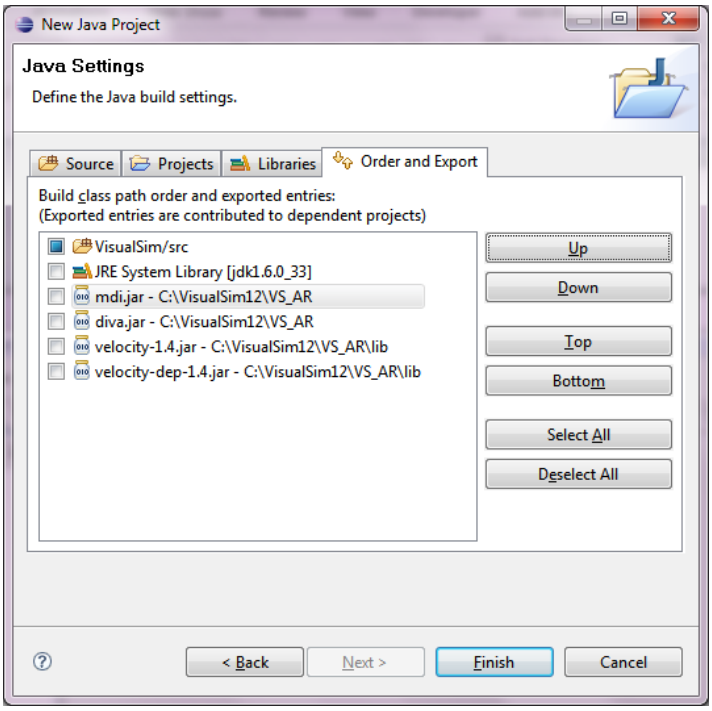


Figure 1.9.4

Update the VisualSim Start Script with below commands.

1. Add compiled classes to the class path
   set CLASSPATH = <path to complied classes>;%CLASSPATH%
2. Prepare Java debug settings
   set dbg = -Xdebug -Xnoagent -Djava.compiler=NONE
   -Xrunjdwp: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.

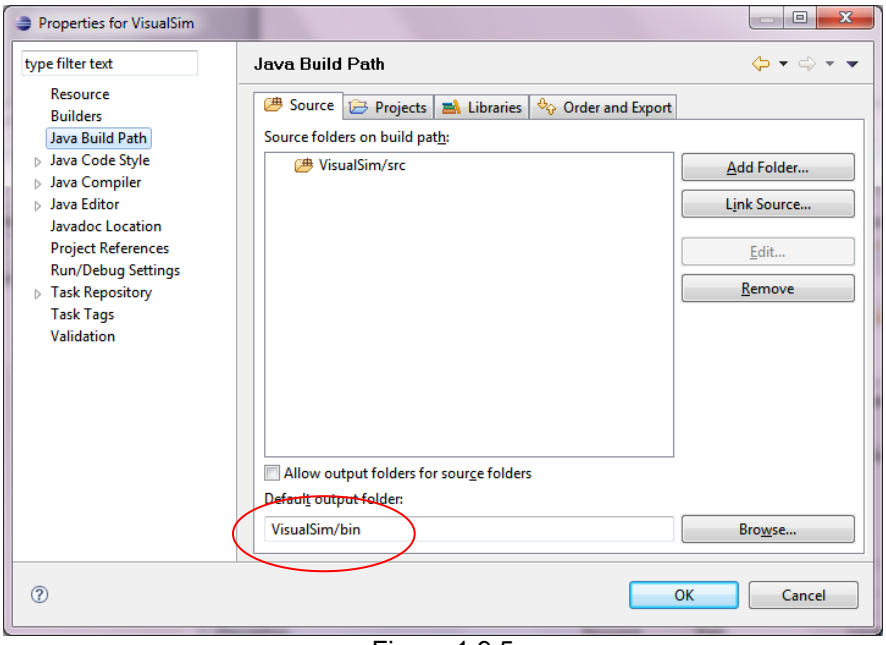


Figure 1.9.5

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)

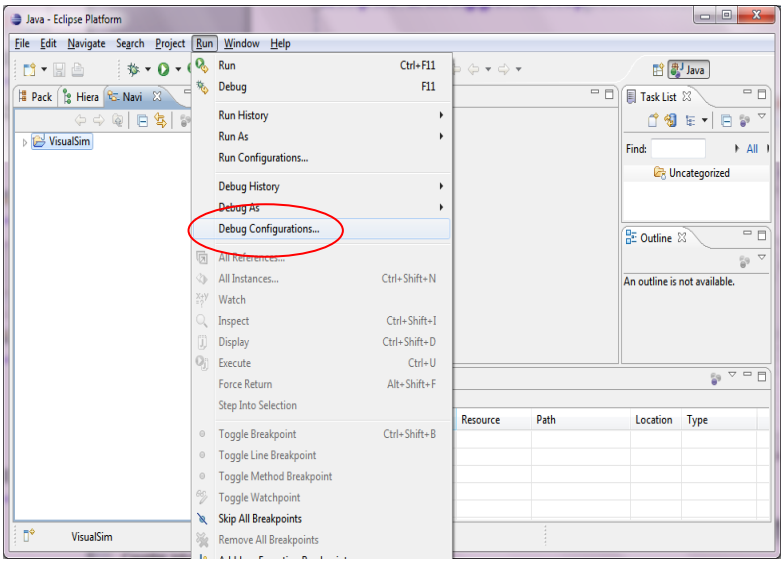


Figure 1.9.6

Choose “Remote Java Application”. You can use default settings. Port should correspond to <debug port> in Java debug options.

Click “Debug”

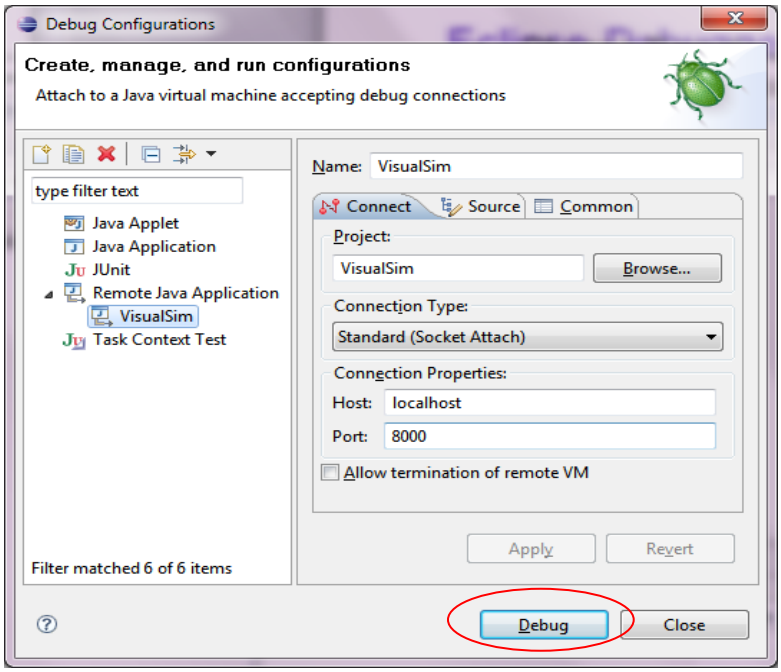


Figure 1.9.7

Now you are ready to debug your code!

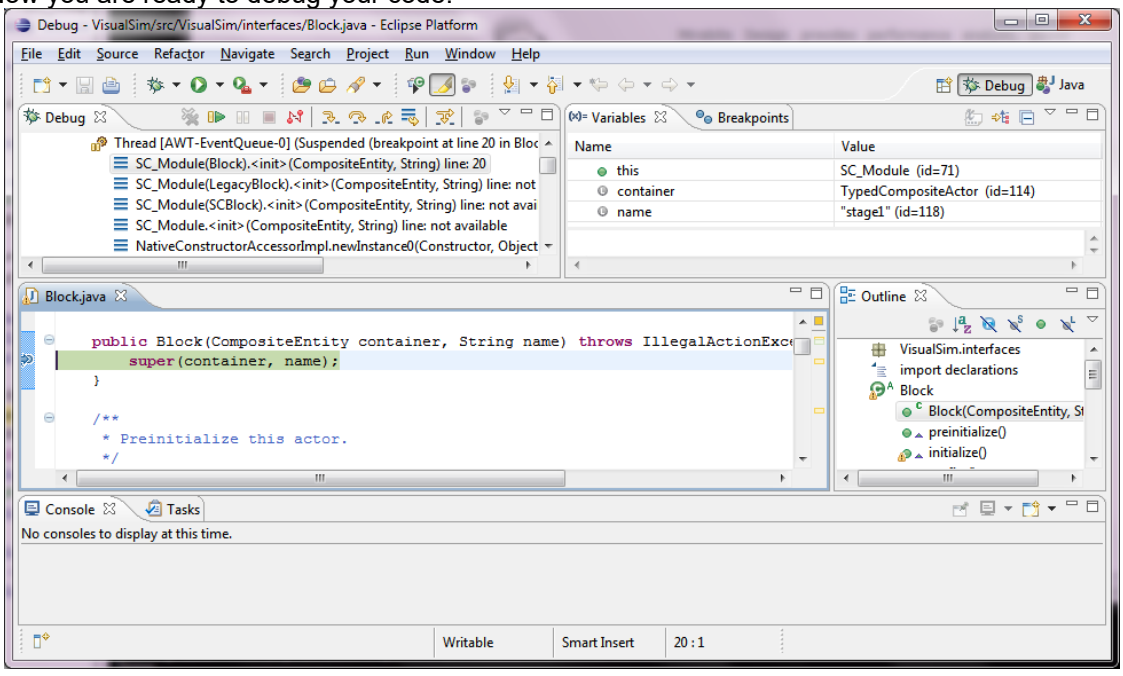


Figure 1.9.8

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, IllegalActionException  {
```

■ to

```
public Ramp2(CompositeEntity container, String name)
throws NameDuplicationException, IllegalActionException  {
```

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
- Clock 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.

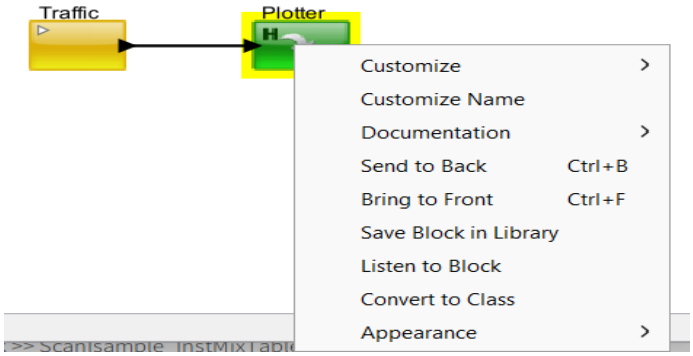
3 Encryption

VisuialSim provides an encryption support for the class files to make sure the library will not be corrupted or modified while sharing it with other teams or customers. This adds security to the model where the classified library or logic can be shared to others without compromising the details.
Follow the below steps to encrypt a class file:

1. Make sure the class file is created and saved within the VS_AR.
2. Use Command line and navigate the tool home directory (VS_AR).
3. Type VisualSim_Model_Encrypt.bat
4. Space
5. Type -xml_class
6. Space
7. Type class file location with xml file relatvie to the VS_AR folder. Eg:

   ./User_Library/Training/Classes/Plotter.xml
8. Space
9. Type -mode encrypt
10. Press enter

Example command:
D:\Tools\VisualSim\VS_2410\VS_AR>VisualSim_Model_Encrypt.bat -xml_class ./User_Library/Training/Classes/Plotter.xml -mode encrypt
Once the encryption is completed the the command line will show “File was successfully encrypted.”
Now if you open the model which contain the class file, no one will be able to open the block. The original file can be kept for modification.



4 SysML to VisualSim

Cameo Systems Modeler is an industry-leading cross-platform collaborative Model-Based Systems Engineering (MBSE) environment, which provides smart, robust, and intuitive tools to define, track, and visualize all aspects of systems in the most standard-compliant SysML models and diagrams.
One can create an activity diagram in the Cameo System Modeler, which contains a sequence of tasks that needs to be executed on a hardware platform. By importing this diagram to VisualSim environment we map it with the hardware platform & we can perform the Hardware-Software simulation.

Cameo System Modeler generates an XML file similar to VisualSim XML files but the format differs in both of them.

So, to import is successfully a SysML to VisualSim converter named "**Xml_Convertor**" is used. It analyses the generated XML file (Cameo System Modeler) & generates an XML file in the format supported by VisualSim.

Fig1: SysML to VisualSim

4.1 Pre-Requisite:

1) Python 3 or higher
2) Input file (i.e. Cameo generated XML file)

4.2 Features supported by XML Converter:

1) Conversion of Activity Diagram.
2) Conversion of Requiremnt Diagram.
3) Conversion of Package Diagram.
4) Conversion of Single block & Multiple block hierarchical diagram.

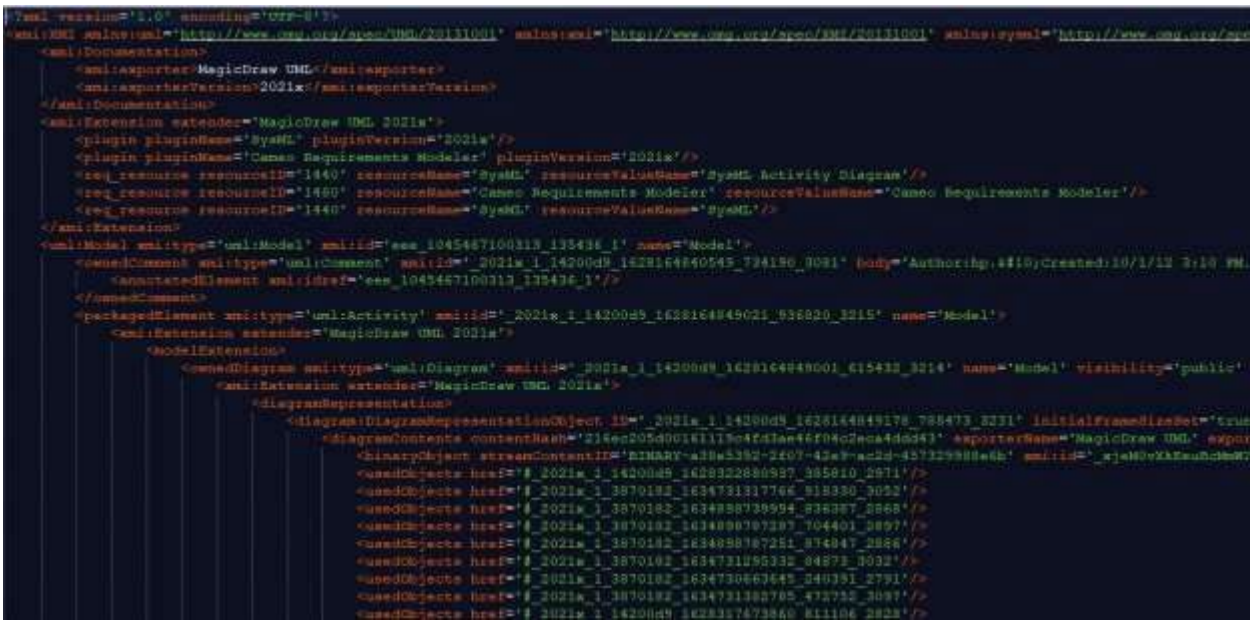


Fig2. Format of XML file in Cameo System Modeler.

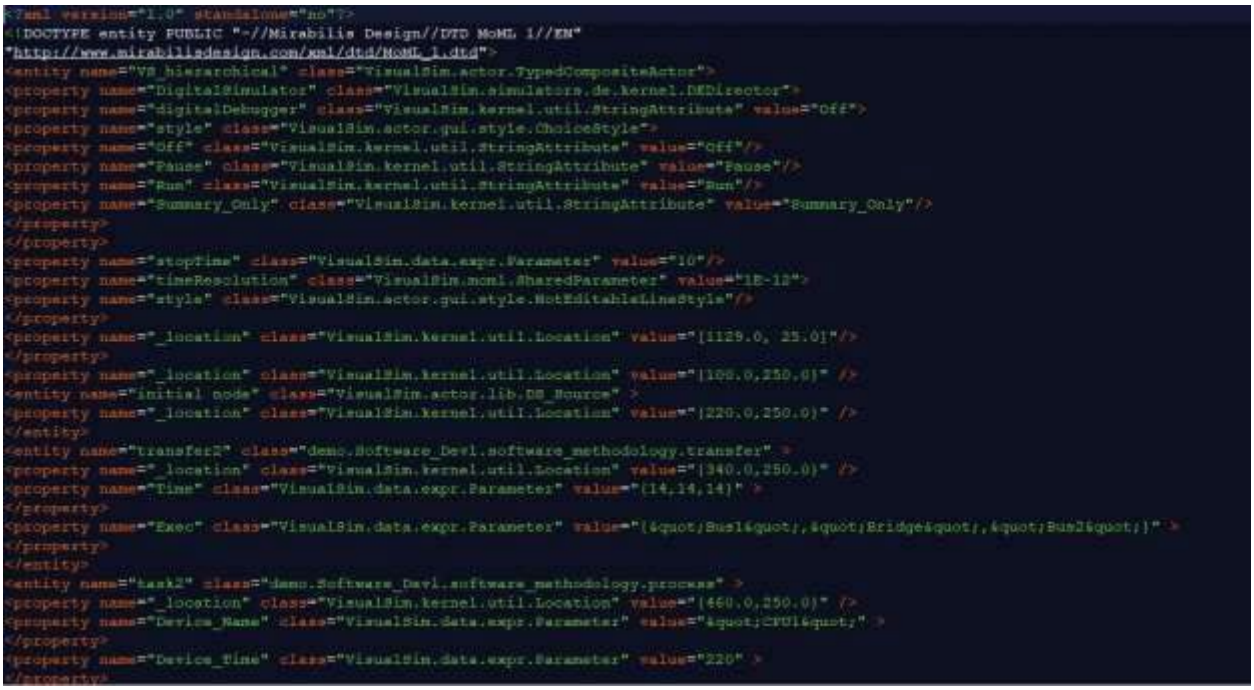


Fig3. Format of XML file in VisualSim Architect.

4.3 Activity and Requirement Diagram

4.3.1 Steps to import Activity Diagram in VisualSim

Step 1: Open Cameo System Modeler then create the requirement & Activity diagram in it. As shown in Fig 4.1 & Fig 4.2. Then click on save project to save. Now the xml file will be created.

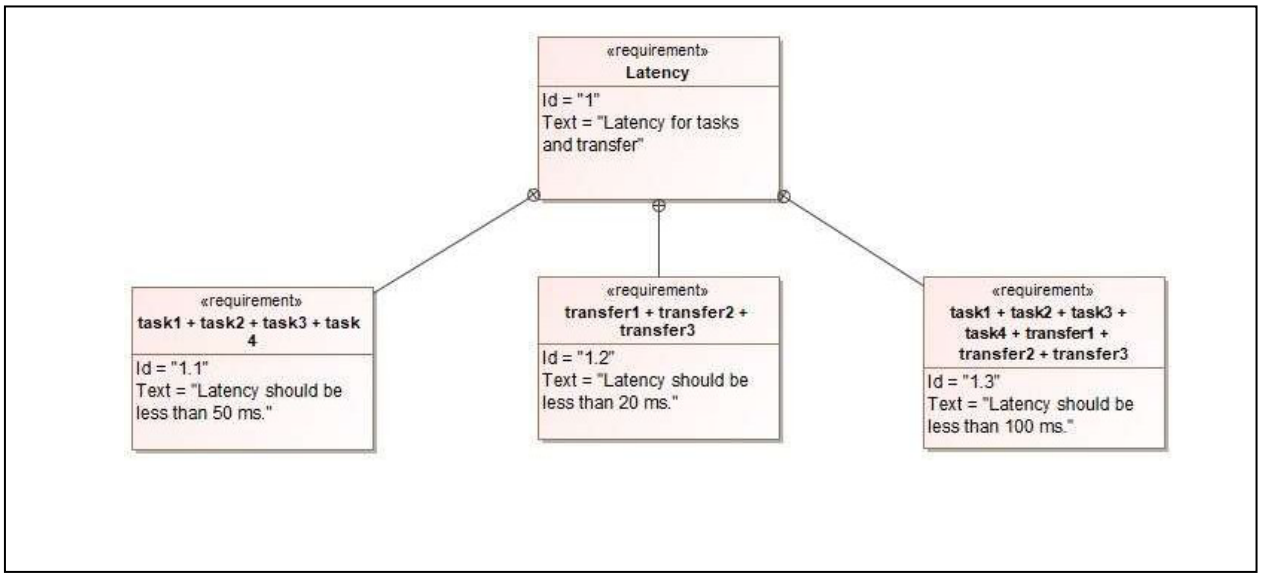
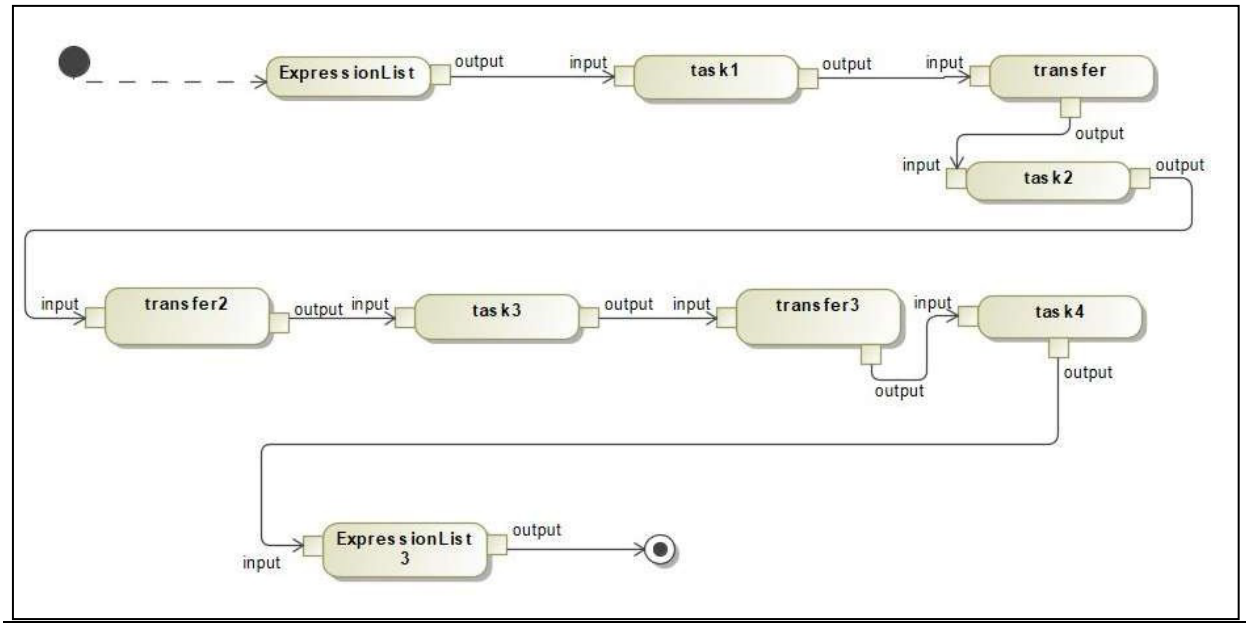


Fig 4.1: Requirement Diagram

Fig 4.2: Activity Diagram



When you double click on any block in the diagram then a window appears. In the Document/Comment section, you need to specify the parameters. As shown in the Fig 5.

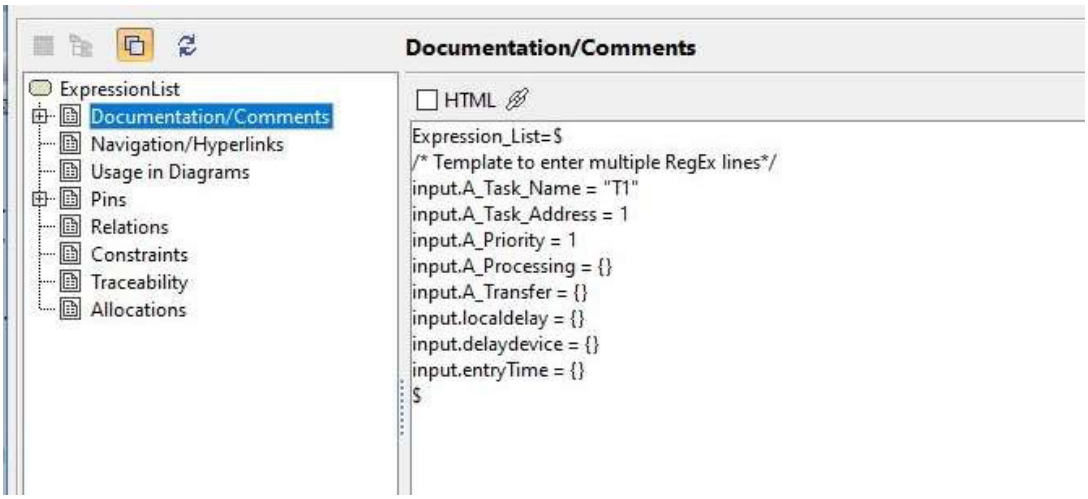


Fig 5: Parameters with multiline values

Expression_List is the name of the parameter & after "=" the value portion starts.
Note: If the value is multiline then you need to enclose the value part by symbol "$", as shown in Fig 5.
If there are multiple parameters then you can write each parameter in next line as shown in Fig 6.

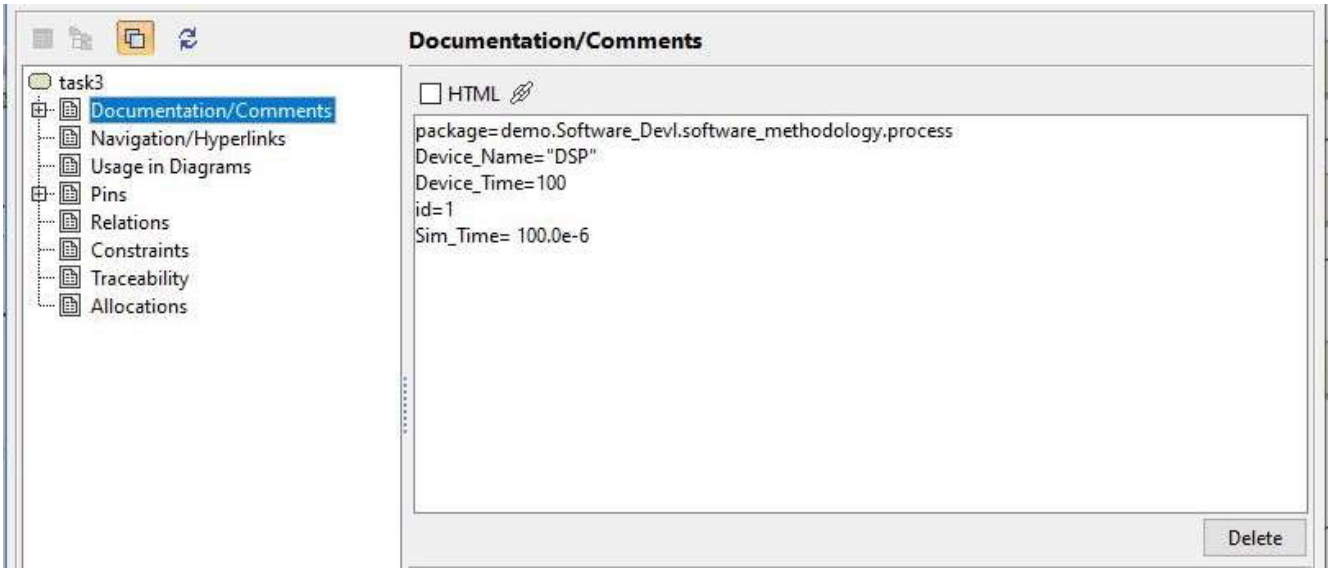


Fig 6: Multiple Parameters with single line values.

Step 2: Execute the python script “xml_convertor.py” & Enter the necessary details about the input and the output file. As shown in Fig7. If everything is correct then the XML file for visualSim will be generated.

```
E:\updated_sysml\final>py xml_convertor.py
Enter input file name:CM_Activity_Diagram_with_object_multiple.xml
Enter outupt file name:Activity_diagram_for_visualsim.xml
Activity_diagram_for_visualsim.xml generated successfully
```

Fig 7: Exection of xml_convertor.py

Here “CM_Activity_Diagram_with_object_multiple.xml” is the name of the input file, which is generated by Cameo System Modeler & “Activity_diagram_for_visualsim.xml” is the name of the output file, which will be generated by the tool.

NOTE : If there is requirement diagram, then one more file is generated programatically, i.e “data.csv” file. Contents of “data.csv” is shown in figure below.

	A	B	C	D	E	F	G
1	BLOCK	METRICS	CONSTRAINT	CONSTRAINT VALUE	STATISTIC TYPE	REFERENCE VARIABLE	
2		1.1 task1 + task2 + task3 + task 4	<	50	Max		
3		1.2 transfer1 + transfer2 + transfer3	<	20	Max		
4		1.3 task1 + task2 + task3 + task4 + transfer1 + transfer2 + transfer3	<	100	Max		
5							

Step 3: Open the VisualSim. As shown in Fig 8.

Fig 8: VisualSim

Step 4: Click on open button. Select the file which is generated programatically. As shown in Fig 9.

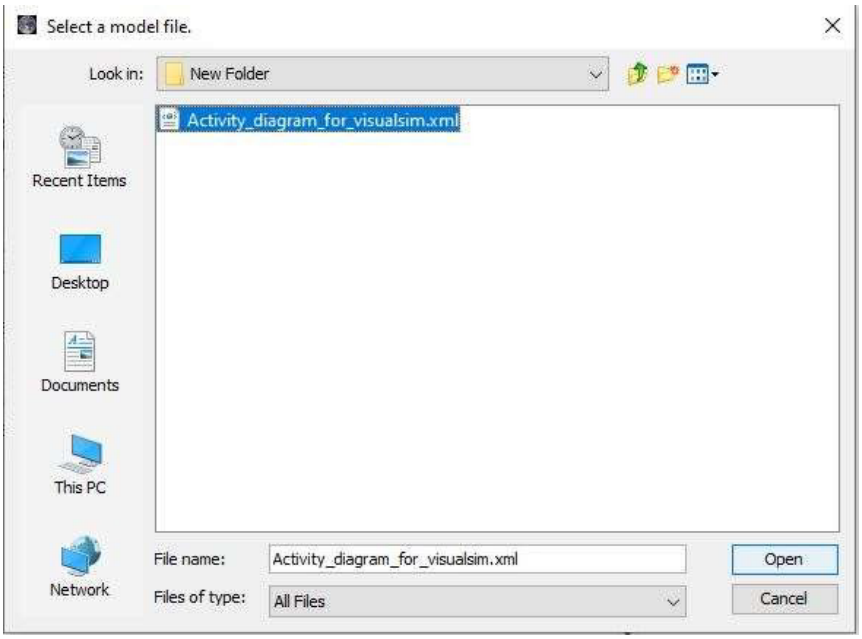
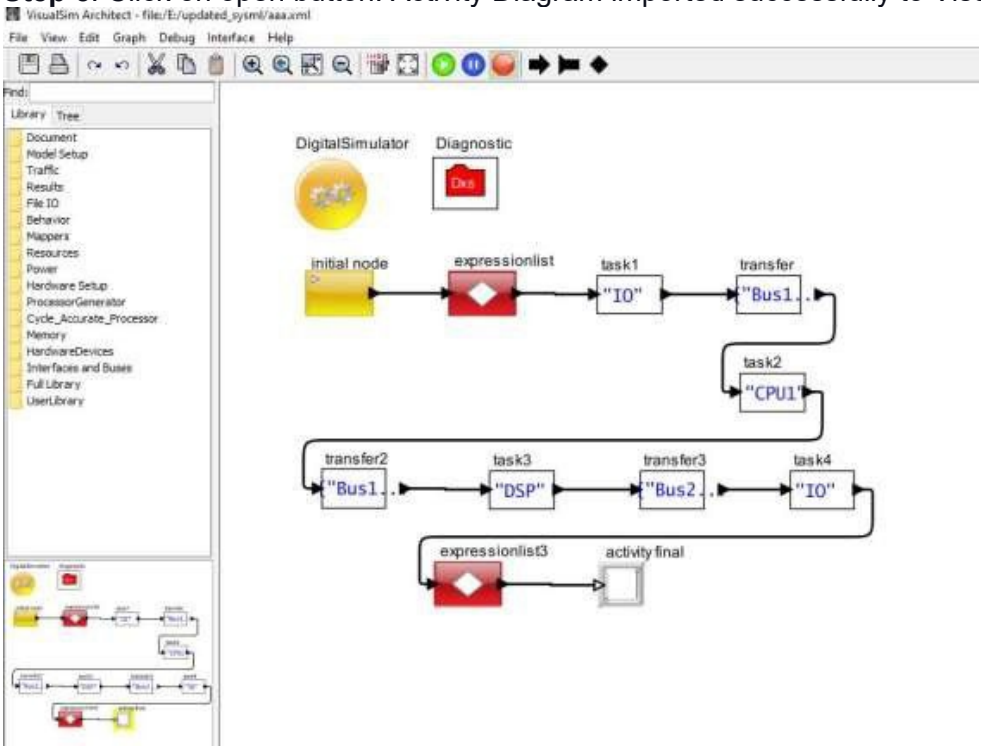


Fig 9: Browse the file

Step 5: Click on open button. Activity Diagram imported successfully to VisualSim.



When you click on any block in VisualSim then you can see the equivalent parameter we specified in Cameo System Modeler.

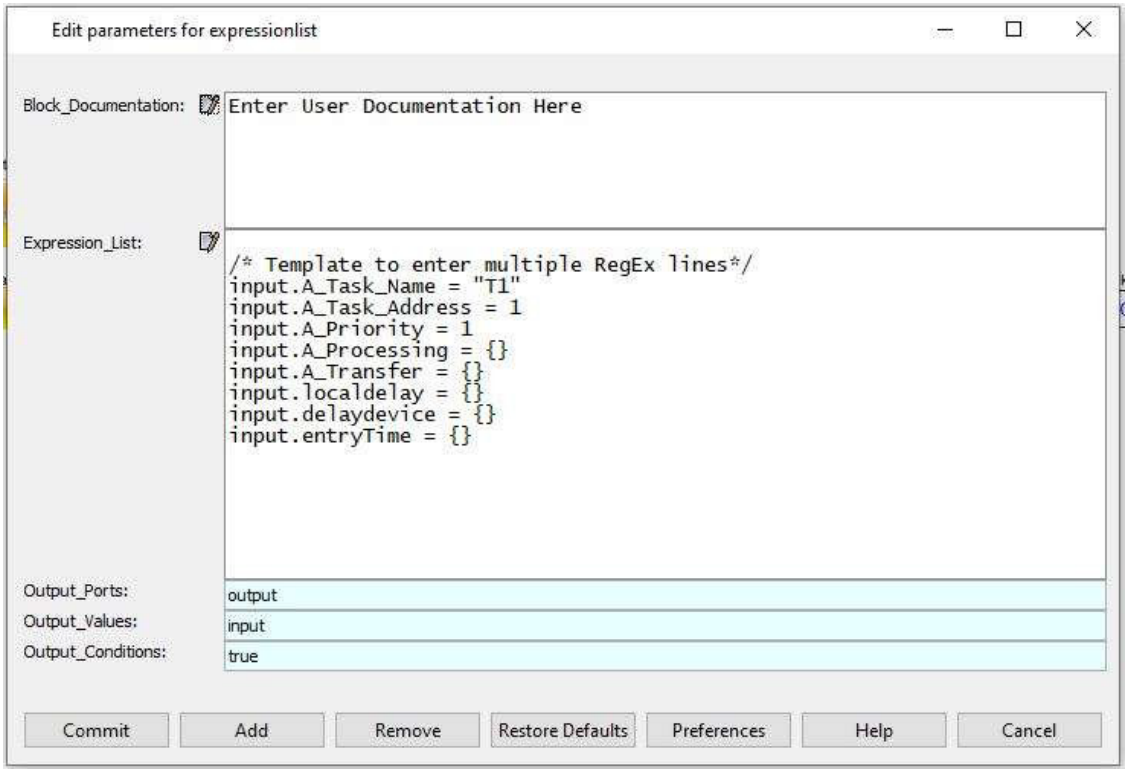
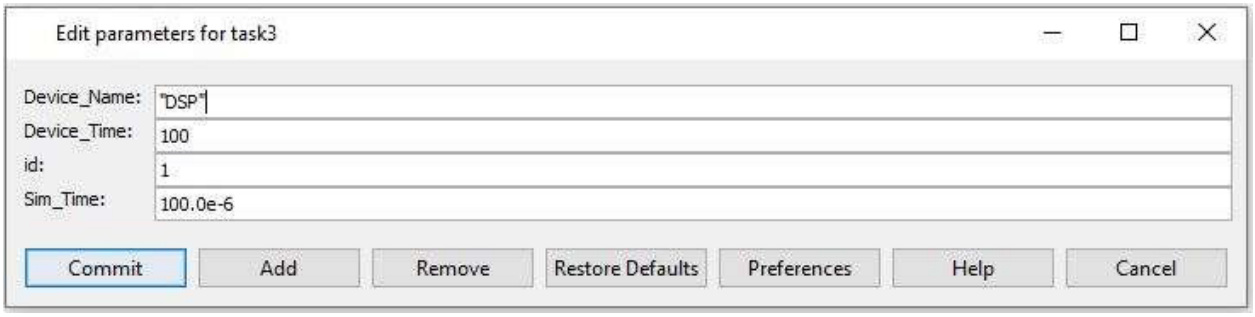


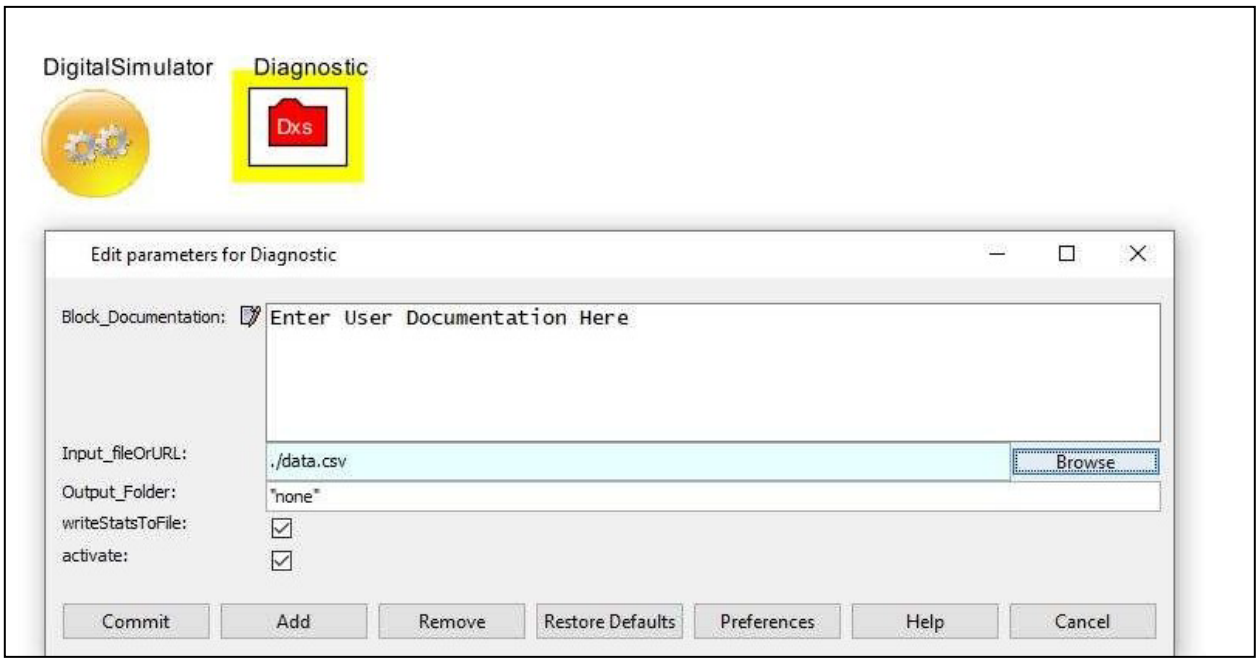
Fig 10(a): Parameters of Expression_list block

You can see in the Fig 10(a), the equivalent parameters for expression_list block listed here.



Similarly, for task3 block. As shown in Fig 10(b).

When you doube click on diagnostic block, then you can see the location of csv file mentioned there.



Using this methodology, you can evaluate the performance of any SysML activity in VisualSim

4.4 Package Diagram

4.4.1 Steps to import Package Diagram in VisualSim

Step 1: Open Cameo System Modeler then create the package diagram in it. As shown in Fig 11.

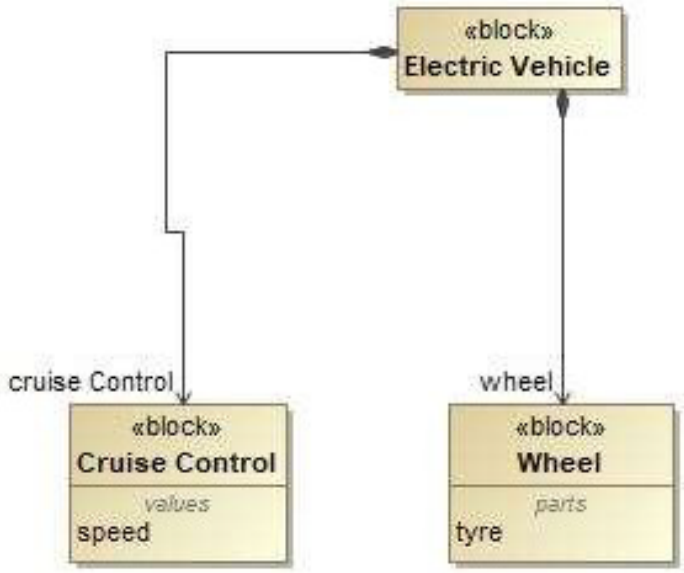


Fig 11: Package Diagram

Fig 11 shows the package diagram created in Cameo System Modeler. Speed is the attribute of Cruise Control block & type is the attribute of Wheel block.

Step 2: Execute the python script “xml_convertor.py” & Enter the necessary details about the input and the output file. If everything is correct then the XML file for visualSim will be generated. As shown in Fig 12.

```
E:\sysML_>py xml_convertor.py
Enter input file name:package_diagram_cameo.xml
Enter outupt file name:package_diagram_for_visualsim.xml
package_diagram_for_visualsim.xml generated successfully
```

Fig 12: Execution of xml_convertor.py for package diagram

Step 3: Open the VisualSim. As shown in Fig 13.

Fig 13: VisualSim

Step 4: Click on open button. Select the file which is generated programatically & open it.

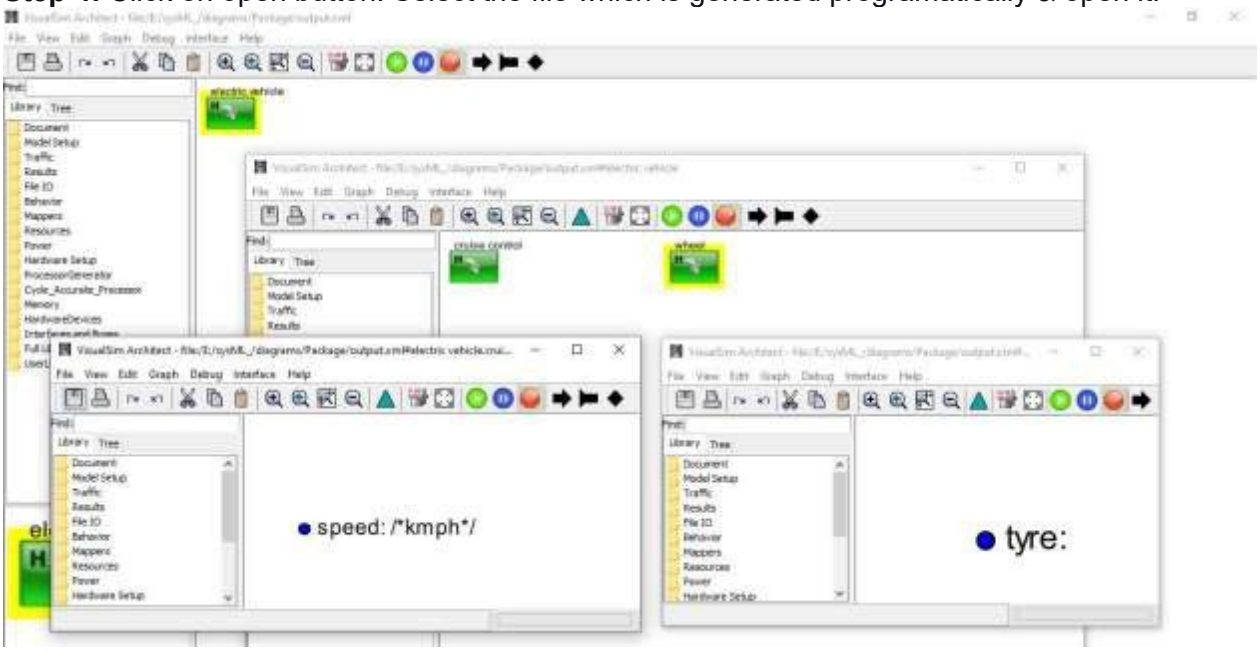


Fig 14: Package diagram in VisualSim

Note: If you see the diagram in the Cameo System Modeler, no value is specified for parameter speed. But in the VisualSim there is some default value present for parameter speed. Because the user has created the file named "parameter.txt" which contains the necessary information regarding the default value for the parameters.

The format of file “parameters.txt” should be like
parameter_first=default_value_for_parameter_first
parameter_second=default_value_for_parameter_seco
nd parameter_third=default_value_for_parameter_third
as shown in Fig 15.

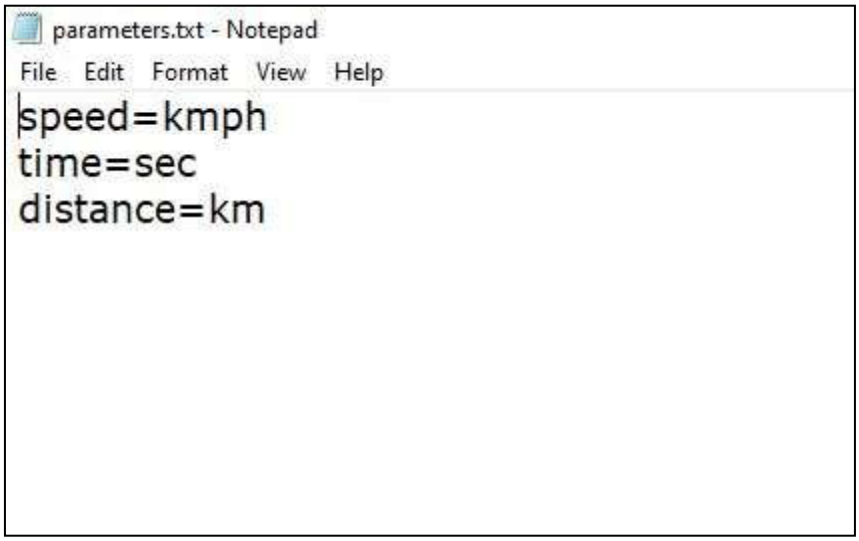


Fig 15: Format of the file parameters.txt

If the user does not create “parameters.txt” file, then no default values will be set against parameters.

4.5 Block Diagram

4.5.1 Steps to import Block Diagram in VisualSim

Step 1: Open Cameo System Modeler then create the block diagram in it. As shown in Fig 16.

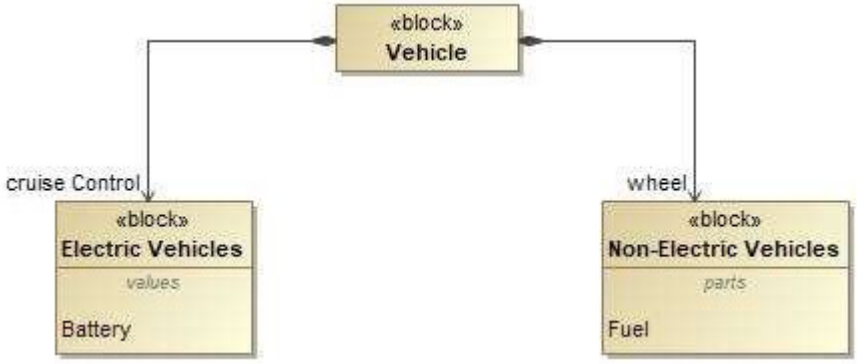


Fig 16: Package Diagram

Step 2: Execute the python script “xml_convertor.py” & Enter the necessary details about the input and the output file. If everything is correct then the XML file for visualSim will be generated. As shown in Fig 17.

```
E:\updated_sysml\block_diagram>py xml_convertor.py
Enter input file name:diagram.xml
Enter outupt file name:VS_diagram.xml
VS_diagram.xml generated successfully
```

Fig 17: Execution of xml_convertor.py for block diagram

Step 3: Open the VisualSim. As shown in Fig 18.

Fig 18: VisualSim

Step 4: Click on open button. Select the file which is generated programatically & open it.

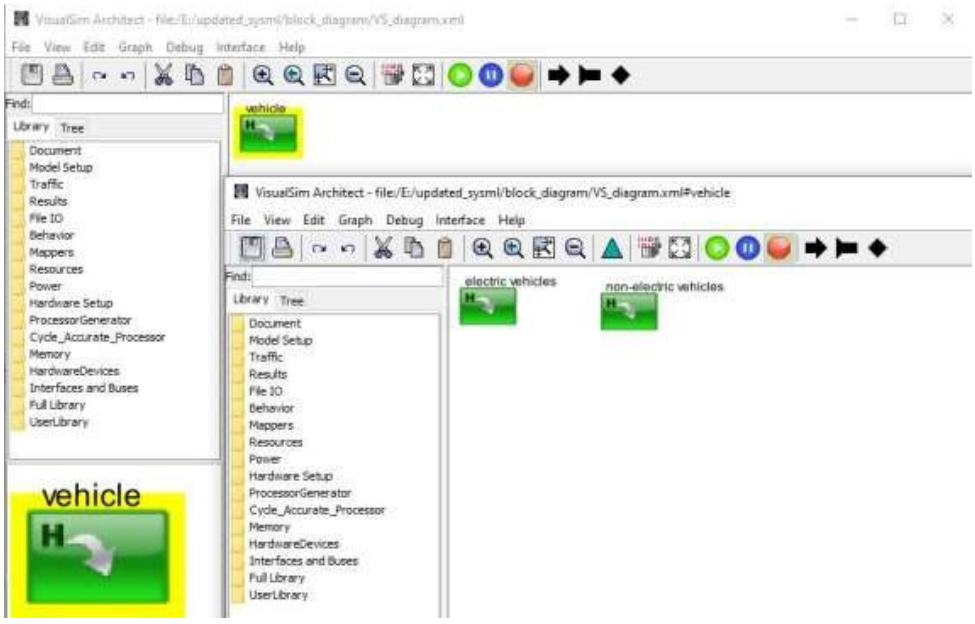


Fig 19: Package

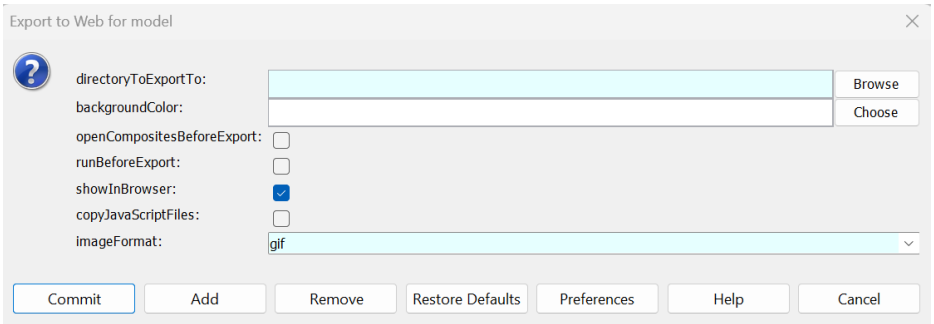
diagram in

----------**********----------

5 Export to Web (Disabled in the current version. It will be available in the future release)

User can share the model to someone else in the team or company, where the other member do not want to install the tool in the local system. The user has to export the model to web and share the html page. By this way others can access them using VisualSim cloud without full tool install in their system.

To Export a model to web, open the model in VisualSim and select File-> Export->Export to Web. Then user will get the following window and update them as mentioned below.



Model export to web configuration.

- Click Browse on directoryToExportTo option and select the model directory.
- Enable openCompositeBeforeExport and CopyJavaScriptFiles in the window.
- Click Commit.
- Now the Index file for this model is created in the specified directory along with the jnlp file.

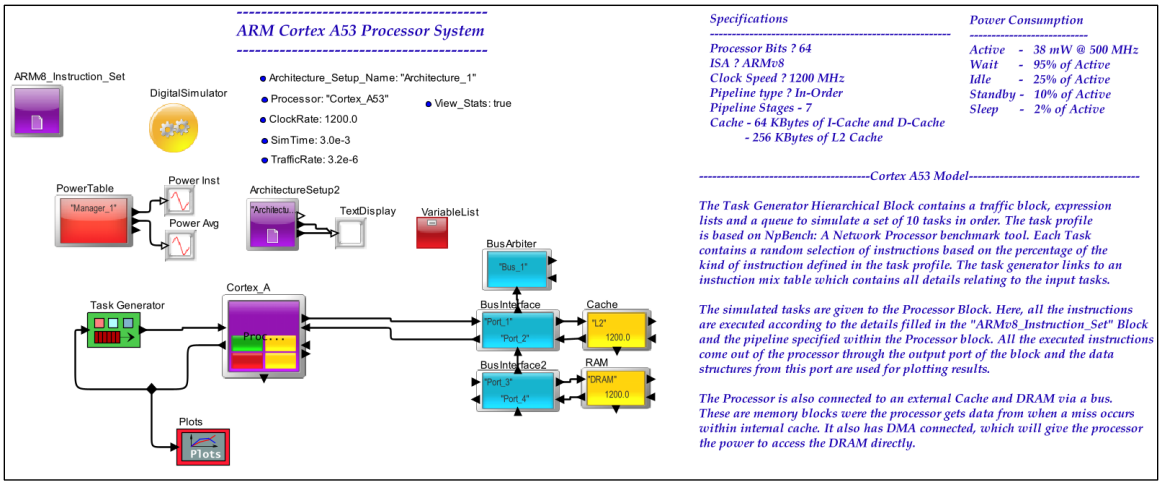
By sharing this index.html file the user can launch the model suing VisualSim cloud.
Once they click launch, the jnlp file can be downloaded in the local system, and then by double clicking them the VisualSim cloud will open the model.

6 Export to HTML

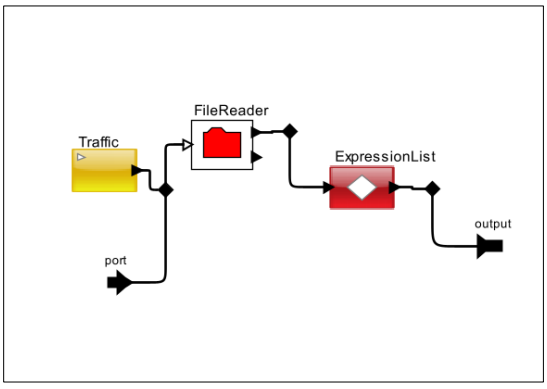
VisualSim provide a method to get an overview of a model by generating html files that describe the model with specific details. It allows user to open any model and generate an html document about it. If the model contain multiple hierarchical levels of implementation the tool generates separate page for each level.

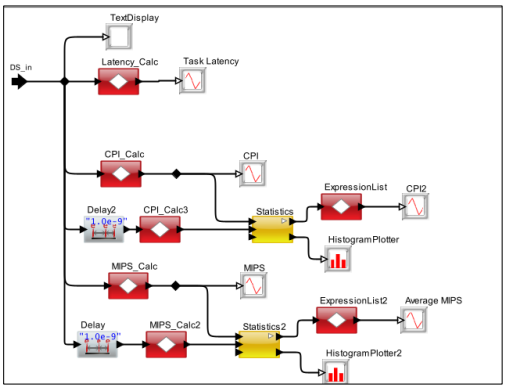
HTML file creation:

Lets take an example demo model from the tool. The following image shows the top level view of the model.



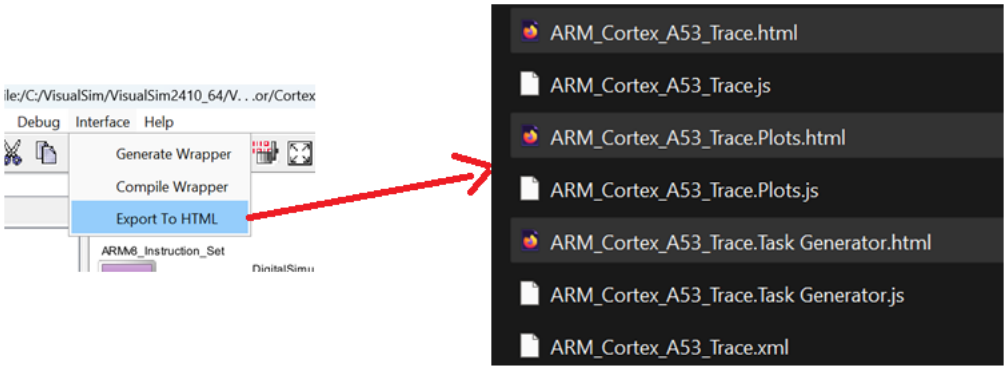
There are two blocks which contains another level of implementation, which are the Task Generator and the Plots block.
The following image the shows the internal implementation of each block.





To export this model into an HTML, select Interface in the top menu and then select Export To HTML.

On the Export to HTML window select the xml file of that model.
You will observe the html files generated in the model location.



The HTML file with the model name provide the documentation of the model with list of components, Introduction, list of parameter and list of VisualSim Blocks.
The Top page will look like this.

ARM_Cortex_A53_Trace

List of Components

Task Generator

Plots

Introduction

---------- ARM Cortex A53 Processor System ----------
----------Cortex A53 Model---------- The Task Generator Hierarchical Block contains a traffic block, expression lists and a queue to simulate a set of 10 tasks in order. The task profile is based on NpBench: A Network Processor benchmark tool. Each Task contains a random selection of instructions based on the percentage of the kind of instruction defined in the task profile. The task generator links to an instuction mix table which contains all details relating to the input tasks. The simulated tasks are given to the Processor Block. Here, all the instructions are executed according to the details filled in the "ARMv8_Instruction_Set" Block and the pipeline specified within the Processor block. All the executed instructions come out of the processor through the output port of the block and the data structures from this port are used for plotting results. The Processor is also connected to an external Cache and DRAM via a bus. These are memory blocks were the processor gets data from when a miss occurs within internal cache. It also has DMA connected, which will give the processor the power to access the DRAM directly.
Power Consumption ---------- Active - 38 mW @ 500 MHz Wait - 95% of Active Idle - 25% of Active Standby - 10% of Active Sleep - 2% of Active
Specifications ---------- Processor Bits ? 64 ISA ? ARMv8 Clock Speed ? 1200 MHz Pipeline type ? In-Order Pipeline Stages - 7 Cache - 64 KBytes of I-Cache and D-Cache - 256 KBytes of L2 Cache

List of Parameters

1. ClockRate : 1200.0
2. SimTime : 3.0e-3
3. Processor : "Cortex_A53"

The List of Components contains the blocks with hierarchical implementation. By selecting that component, you can observe the documentation about that block.

The Introduction contains the annotations added to the model in the top level and block documentation in each block parameters.

User can use this document to share about the model to other team member or review teams to get the overall top-level validation.

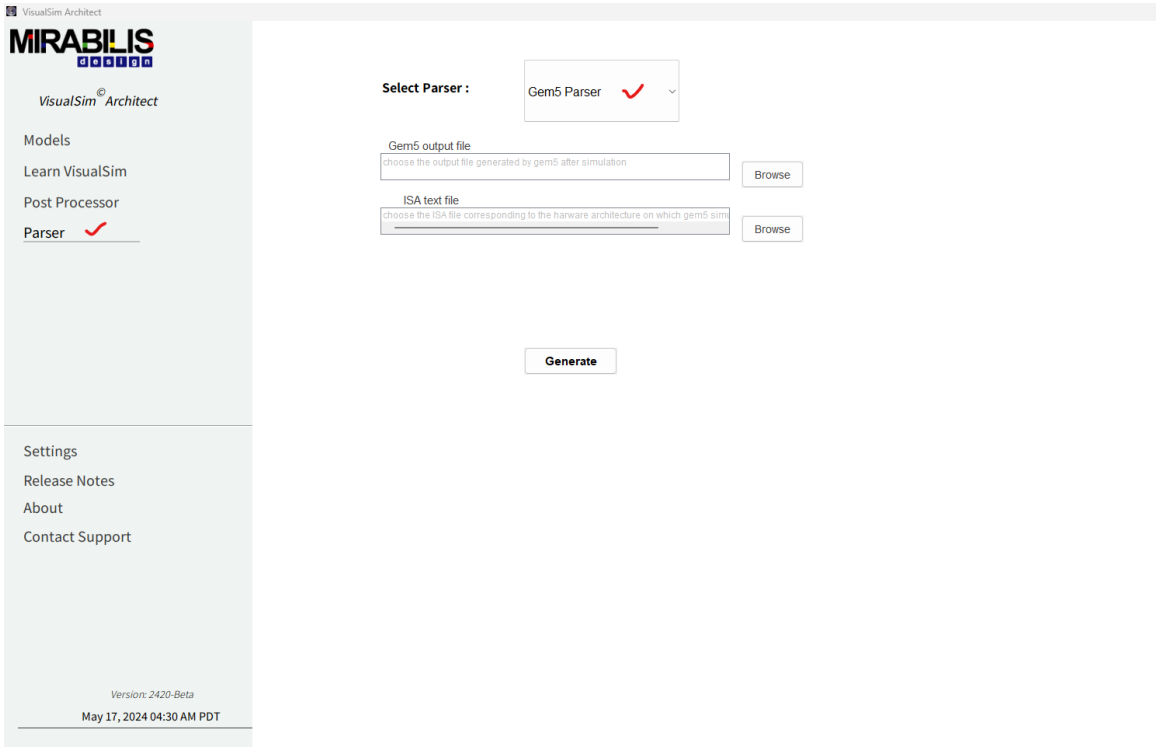
7 Methodology to generate software trace from C/C++

VisualSim processor module accept executed software trace from 3rd party tool where user can analyze and optimize the overall hardware architecture. Users can choose any 3rd party application that they have access to, and have to make sure that the generated traces contain the instructions executed, instruction addresses and data cache addresses for load store instructions.
In this section, GEM5 and ARM Fast model are used to explain the process.

Using GEM5 to capture traces:

1. GEM5 requires Linux to run. So if you have Linux machine, then good to go. Else, download a VMWare and install Linux on it.
   We were using an older version of VMWare Pro. But the website only has newer ones - https://www.vmware.com/in/products/workstation-player/workstation-player-evaluation.html
   Make sure to allocate at least 100 GB of size to the linux in VMWare allocation
1. GEM5 installation - https://www.gem5.org/documentation/general_docs/building
2. ARM compiler installation - https://developer.arm.com/downloads/-/gnu-a
   We need to use - aarch64-none-linux-gnu
3. Once all the above steps has been completed, try running hello world example on GEM5 (GEM5 installation already contains compiled hello word binaries). So we can run it using:
   build/ARM/gem5.opt configs/example/se.py --debug-flags=Exec --debug-file=arm_trace ./configs/example/arm/starter_se.py --cpu minor --cpu-freq 1.2GHz --mem-type DDR4_2400_8x8 ./tests/test-progs/hello/bin/arm/linux/hello
4. Once the above command has been run, a file will be created in ./m5out/ folder with the file name “arm_trace” . This file contains the exec traces generated while running Hello world on GEM5. It will give information on the instructions executed, instruction and data addresses etc,. You can use “gem5_all_parser_combined.py” to parse this file and generate a list of initialized data used by GEM5 while running the corresponding application.
   For more information on GEM5 - https://www.gem5.org/documentation/learning_gem5/introduction/
5. For compiling dhrystone, we used the following commands:
   aarch64-none-linux-gnu-gcc -O0 -mtune=cortex-a77 -mcpu=cortex-a77 --static -c -DHZ=60 -O2 -fno-inline dhry_1.c
   aarch64-none-linux-gnu-gcc -O0 -mtune=cortex-a77 -mcpu=cortex-a77 --static -c -DHZ=60 -O2 -fno-inline dhry_2.c
   aarch64-none-linux-gnu-gcc -O0 -mtune=cortex-a77 -mcpu=cortex-a77 --static -o dhrystone dhry_1.o dhry_2.o
6. The dhrystone binary can then be run on GEM5.
7. Similarly, users can compile the C/C++ and run it on GEM5. Make sure the debug flags are set so that the trace will be generated.
8. Once the Trace has been generated, we need to use the python parser provided by Mirabilis Design to convert the raw trace file into csv file format which is readable by VisualSim Architect.

Use the “Parser” section in tool home page and select Gem5 parser, then select the GEM5 output file and ISA text file. ISA Text file is the instruction set used by VisualSim Hybrid Processor in the demo model. Please check the below image for reference.



Using ARM Fast/Cycle Model to capture traces:

Once the source code has been compiled, user can run the binary on the fast/cycle model using the command below:

"%ISIM%" --stat -C bp.secure_memory=false --plugin=C:\Program Files\ARM\FastModelsPortfolio_11.16\plugins\Win64_VC2019\Release\TarmacTrace.dll -C TRACE.TarmacTrace.trace-file="D:\Projects\ARM_Trace_Prog\cca_trace.txt" -a D:\Projects\ARM_Trace_Prog\cca\Debug\microbench.axf

For generating the tarmac traces, we need to make sure the TarmacTrace.dll has been loaded and the trace file name has been defined, as shown above.

For more details on configuring fast models please refer the documentation - https://developer.arm.com/documentation/100964/1123/Plug-ins-for-Fast-Models/TarmacTrace/Instruction-trace?lang=en

Please check with mirabilis team for parsing the output file to a desired format that can be used in VisualSim environment.

8 Steps on How to run FPGA C++ Model

Custom C/C++ interface allows user to import existing C/C++ code into the VisualSim environment and build a model utilizing the application code. For example the a custom interconnect/bus block is designed in the C++ code and the user wants to build uses cases for the interconnect logic and validate them. Now the user can import the code using Custom C++ interface and build test models using VisualSim library components around the interface block.

Doing this the user can take advantage of VisualSim's analysis and debugging features to validate the logic and find bottlenecks with that. User can also use the this method to build modes for a standard C++ library code such as microprocessor. They can just use a low level design of the processor in c++ and connect the memory, buses and IO devices using VisualSim to build an SoC design.

The following content will walk you through the code interface and how to run a model with C++ code.

Modifications and verification before opening the tool.

1. Update "VSDIR" variable in VisualSim.bat/.sh compiler path.

```
17
18  set VSDIR = C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC
19  ::For Custom block development and generation
```

2. Update "VSDIR" variable in VSConfig.properties/VSConfig_linux.properties compiler path.

```
1  SimulatorName=ModelSim
2  VERILOGPATH=C:\\VisualSim\\Tools\\Modeltech_pe_edu_10.4a
3  VSDIR=C:\\Program Files (x86)\\Microsoft Visual Studio\\2019\\Community\\VC
4  VCVARS_CMD=%VSDIR%\\Auxiliary\\Build\\vcvars64.bat
```

3. Update "VS_C_DLL" variable in VisualSim.bat/.sh to specify the location for compiled object.

```
9   ::For Custom block development and generation
10
11  set VS_C_Library=%INSTALL_PATH%\demo\Interfaces\CustomC\FPGA
12  set VS_C_DLL=%VS_C_Library%\blocks
```

4. Update line 38 in vsmake.bat/.sh to specify the location for compiled object.

```
36  rem del Jni*.h
37
38  IF  "%DEMO_Compile%"=="Yes" (copy *.dll %INSTALL_PATH%\demo\Interfaces\UserNativeC\blocks) ELSE (copy *.dll %VS_C_Library%\blocks)
39
```

Model design:

1. Open an existing model or a new model in the tool.
2. Drag and Drop Application_Interface block (Full Library-> System_Language->C_and_CPP).
3. Set the module name and provide the path to the C/C++ file
4. The C++ code must have the following parts:

   include interface header file for the socket communication:

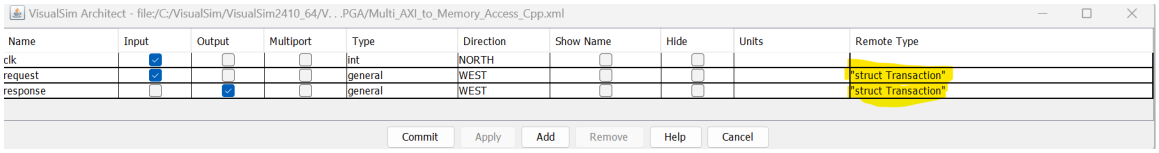
#include "../CustomCSlave/ProxyCustomCSlave.h"
where CustomCSlave is the directory of the generated files which is the module name and ProxyCustomCSlave is the concatenation of Proxy + module name.

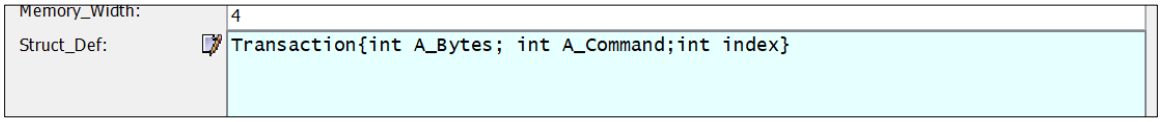
```
#include "../CustomCSlave/ProxyCustomCSlave.h"

namespace CustomC {

#define PKT_FETCH 0
#define PKT_RELEASE 1
#define BUFF_LEN 10
```

5. Add “ProxyCustomC* createProxy()” method to the source code. It is explained in the example below.

```
    };

    ProxyCustomC* createProxy() {
        return new CustomCSlave();
    }

} // namespace CustomC
```

6. Add parameters and link to the source code to easily run use cases without recompiling
7. Right Click -> Customize -> Ports : Add required ports and data types. For data structure, we need to define "struct Transaction“ under Remote Type column as well as define a top-level String Parameter called “Struct_Def” to configure the DS fields.

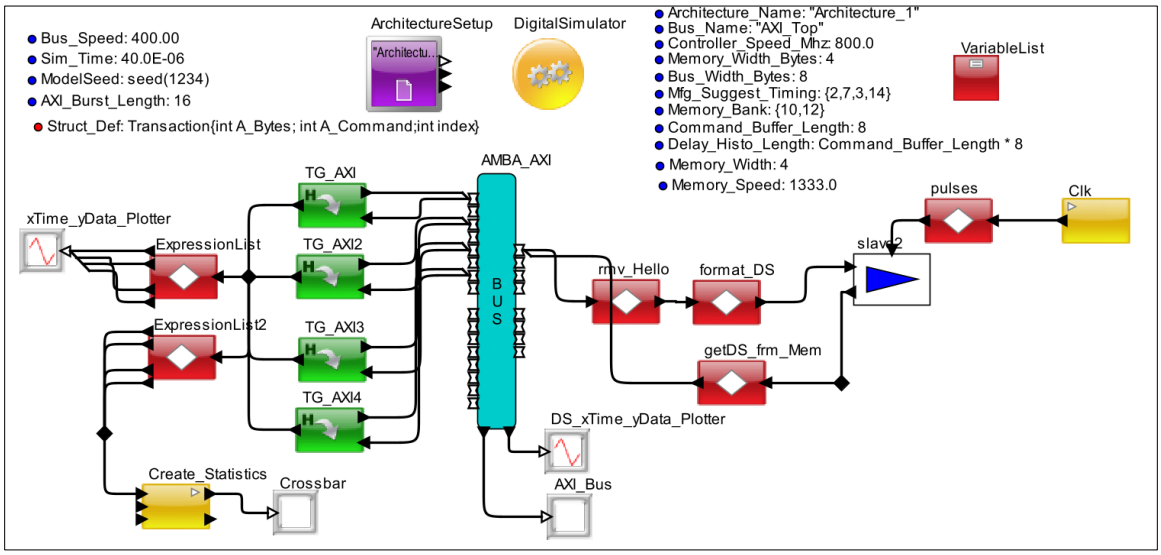




8. Once the code is set up, user need to – Generate Wrapper (Menu Bar -> Interface -> Compile Wrapper) and then Run simulation

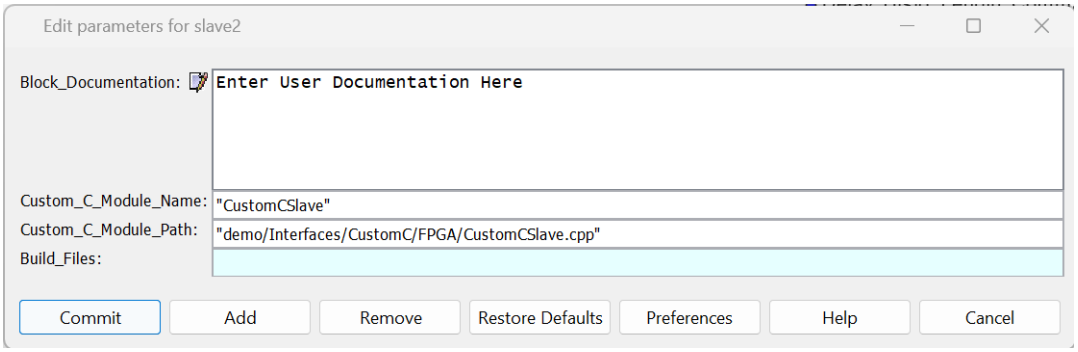
Example case:

1. Open the file in VS_AR\demo\Interfaces\CustomC\FPGA\ Multi_AXI_to_Memory_Access_Cpp.xml

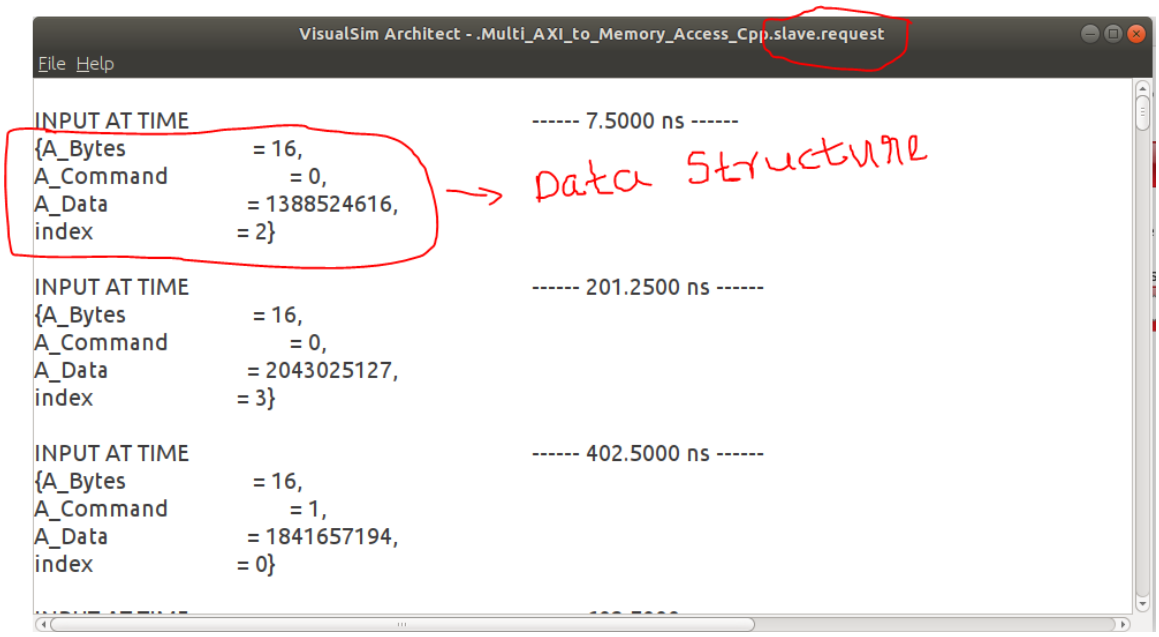


In this model the packets are being sent from the Traffic Generators (TG) through ARM AMBA AXI Bus which is then sent to the C++ module. Before the packets are sent through C++ module, we store the packet in memory and only send required field and values to the C++ module. It is done in format_DS block.

2. Double click on the Slave module (C++). A parameter configuration window for the C++ module comes up:

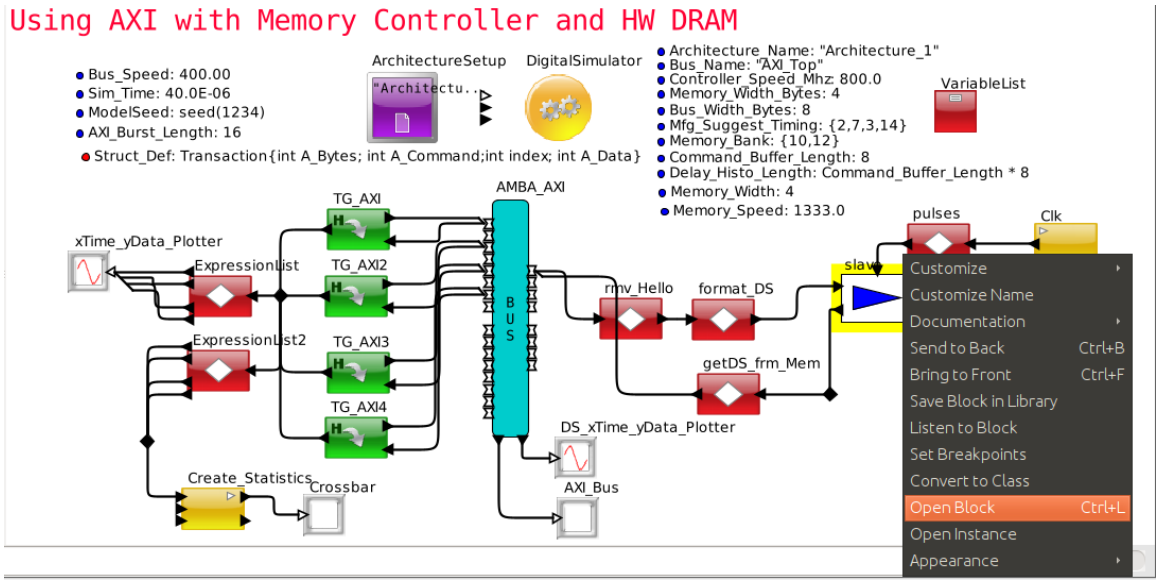


3. Confirm that the C++ file is placed in the correct path. Otherwise modify here.
4. The packet coming into the C++ module looks like the following:



We can also see the port name on top of the window.
A_Command = 0 denotes a Read operation and A_Command = 1 denotes a Write Operation,

5. When the packet comes into the C++ module, we place the packet into a buffer. According to the clock pulse value we process the packets.
6. If Clock pulse value received is 0, then we take the packet from the buffer for processing.
7. When the Clock pulse value becomes 1, the packet will be sent out back to the source via ARM AMBA AXI Bus. We Read the original packet from memory by using index value.
8. For viewing C++ code directly from VisualSim, Right click on Slave module -> Open Block



```
#include "../CustomCSlave/ProxyCustomCSlave.h"

namespace CustomC {

#define PKT_FETCH 0
#define PKT_RELEASE 1
#define BUFF_LEN 10

    class CustomCSlave : public ProxyCustomCSlave {

        int phase;
        int qLen;
        int qPos;
        int popPos;
        Transaction transactionValue;
        bool clkTriggered;
        Transaction bufferedDS[BUFF_LEN];
        public:

        void init() {
            phase=0;
            clkTriggered = false;
            qLen = 0;
            qPos = -1;
            popPos = -1;
        }
```

9. We have separate sections with certain format to receive the packets from input ports. They are:

```
    }

    void clk_received(int data) {
        phase = data;
    }

    void request_received(Transaction data) {
        if(qLen<BUFF_LEN)    {
            qLen = qLen + 1;
            qPos = qPos + 1;
            qPos = qPos%BUFF_LEN;
            bufferedDS[qPos] = data;
        }
    }

    void fire() {
```

The clk_received function handles the clock input and the request_received function handles the request input. The function argument type will be same as the type of data the port receives. clk port receives integer values, so the function type is int. request port receives Data Structure as input. So we defined the type as Transaction.
We can see that whatever is the clock pulse value, it will be stored in a variable called phase. Also, we can see the input packets are placed into the buffer.

10. The C++ file can have few other functions as well. The block supports initialize (init), pre-fire (prefire), fire and post-fire as methods to link to the simulation. At the start of the

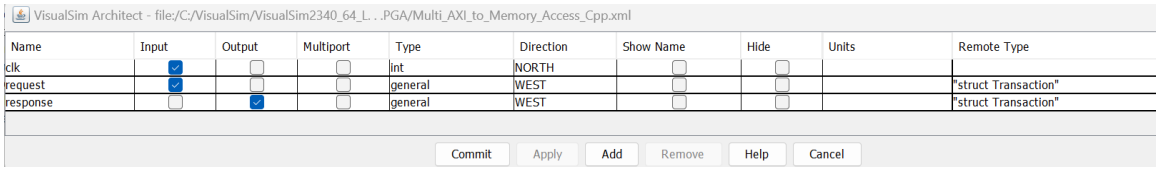
simulation, init method is invoked and executed. The block can be triggered by data arriving on any of the input ports. When this occurs the block sends the data to the code. If there code has any actions for the data, it can be performed. The fire method is where the execution occurs

```
void fire() {
    switch (phase) {
        case PKT_FETCH:
           if(qLen > 0) {
              clkTriggered = true;
              popPos = popPos + 1;
              popPos = popPos%BUFF_LEN;
              transactionValue = bufferedDS[popPos];
              qLen = qLen - 1;
            }
            break;
        case PKT_RELEASE:
            if(clkTriggered)    {
                int cmd = transactionValue.A_Command;
                if(cmd == 0)    {
                    transactionValue.A_Command = 1;
                }
                else    {
                    transactionValue.A_Command = 0;
                }
                send_response(transactionValue);
                clkTriggered = false;
            }
            break;
    }
```

According to the phase value, a type of action is selected.
So if phase value is 1, then we update A_Command field value and send it back to other VisualSim modules. For sending a data through a port, we have to use function send_{port name}(data value). It is shown here:

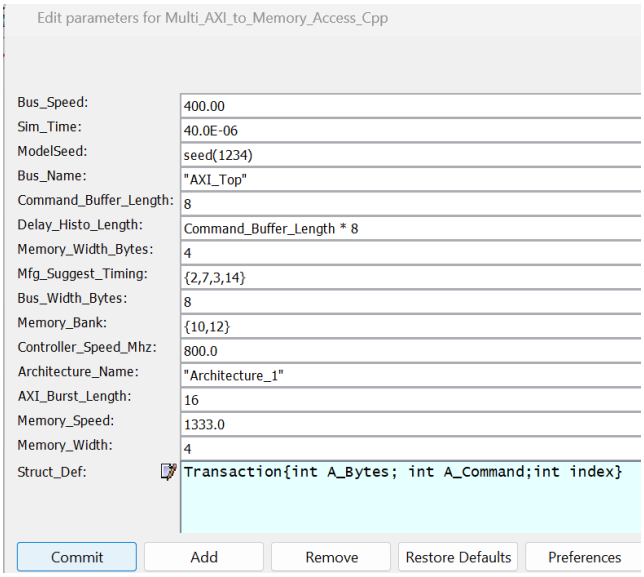
```
                transactionValue.A_Command = 0;
        }
        send_response(transactionValue);
        clkTriggered = false;
    }
    break;
```

11. One additional point that we have to keep in mind is that, if we want to receive and send data structures like we do in this example, then we have to configure some parameters.
    1. Right click on Slave (C++ module) -> Customize -> Ports
       We can see the port names, the type of data accepted on that port and the Remote Type. For ports configured to receive or send data structure, we define the Type as general and specify Remote Type as “struct Transaction”



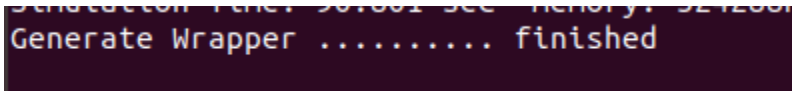
2. We have to add a StringParameter to the model with name Struct_Def. It is used to define the data structure fields that can be accepted.

To view all the model parameters, right click on the model window (anywhere in the window) -> Configure. We can see that, the Transaction is set to have 3 data structure fields and the data types are also mentioned here.

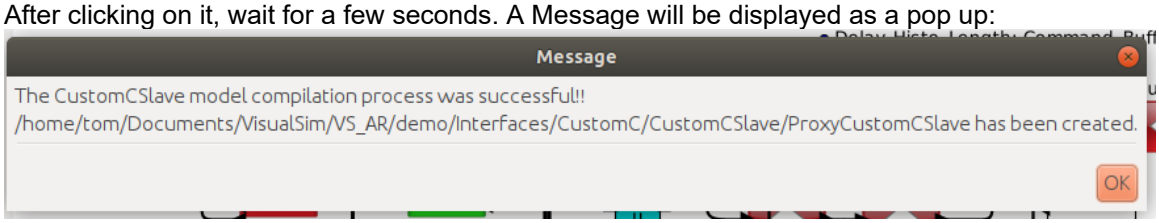


Runing the Model:
In order to run the model, we first have to generate a Wrapper. Select Interface Menu item and click on Generate Wrapper.

After clicking on it, check the VisualSim Architect Terminal running in the background and check for this message.



Now Go to Interfaces menu again and select Compile Wrapper:
After clicking on it, wait for a few seconds. A Message will be displayed as a pop up:



Now we can run the model. Click on the green play button on the control bar.

9 Functional Testing of applications

Prerequisites:
Install Visual Studio (Windows) with C/C++ packages
[https://visualstudio.microsoft.com/vs/community/] or gcc (Linux)

Open <VisualSim Installation folder>/VS_AR/VisualSim.bat or .sh(Linux) using a text editor:

```
set JAVA_HOME=C:\Tools\Java20
set INSTALL_PATH=D:\Tools\VisualSim\VS_2320_j20\VS_AR
set MATLAB_PATH=$MatLab_Location
set VS=%INSTALL_PATH%
set VERILOGPATH=C:\questasim_10.0b
set GCCPATH=C:\TDM-GCC-32\bin
cd %INSTALL_PATH%

::For Custom block development and generation

set VS_C_Library=%INSTALL_PATH%\demo\Interfaces\UserNativeC
set VS_C_DLL=%VS_C_Library%\blocks

:: Location of the User Library
set VS_Model_Library=%INSTALL_PATH%\User_Library;D:\Projects\Denso;D:\Projects\Interfaces_CustomC
:: Location of the User Library

set VSDIR = C:\Program Files (x86)\Microsoft Visual Studio\2019\Community
::For Custom block development and generation
```

Make sure the VSDIR variable is updated to the correct path.

Open <VisualSim Installation folder>/VS_AR/VSConfig.properties or VSConfig_linux.properties(Linux):

```
SimulatorName=ModelSim
VERILOGPATH=C:\\VisualSim\\Tools\\Modeltech_pe_edu_10.4a
VSDIR=C:\\Program Files (x86)\\Microsoft Visual Studio\\2019\\Community\\VC
VCVARS_CMD=%VSDIR%\\Auxiliary\\Build\\vcvars64.bat
SYSTEMC_HOME=%INSTALL_PATH%\\VisualSim\\simulators\\SystemC\\Win\\systemc-2.3.3
SYSTEMC_LIBPATH=%SYSTEMC_HOME%\\msvc10\\SystemC\\x64\\Release
SYSTEMC_LIB=SystemC-2.3.3.lib
Remote_Address=127.0.0.1
Port_Number=2400
Default_Browser=chrome
AUTOSAR=%INSTALL_PATH%\\demo\\automotive\\Autosar
RTE_PROXY=%AUTOSAR%\\rte_proxy
appletMode=POPUP
Auto_Save_Time=10
text.editor.font.size=19
undefined_resource=true
```

Make sure the VSDIR variable is updated to the correct path.

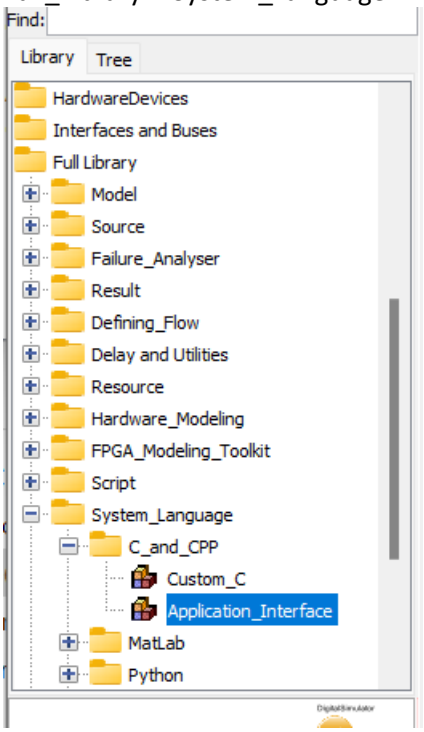
Example Application code:

The source code of a simple application:

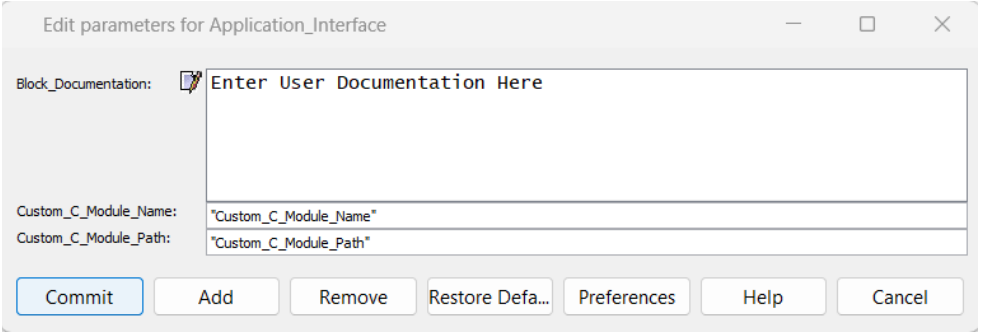
```c
#include <stdio.h>
#include "shared_variables.h"

int OP1_REG;
int OP2_REG;

// Function to perform sum operation
int sumOperation() {
    int a =OP1_REG;
    int b =OP2_REG;
    // Perform sum operation
    int sum = a + b;

    return sum;
}

// Function to perform subtraction operation
int subtractOperation() {
    int a =OP1_REG;
    int b =OP2_REG;
    // Perform subtraction operation
    int subtraction = a - b;

    return subtraction;
}
```

Steps to Build the funciotnal testing model:

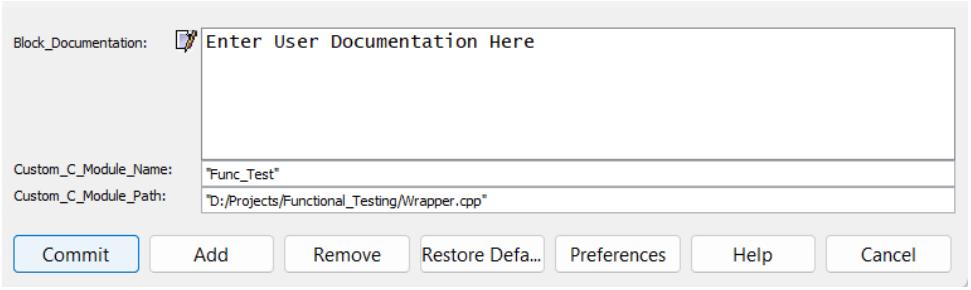
1. Drag and drop c code interface from library.
   Full_Library-> System_Language -> C_and_CPP -> Application_Interface



2. We need to double click on the Application_Interface



Set Custom_C_Module_Name and Custom_C_Module_Path
Custom_C_Module_Path is the path to the Wrapper code - which defines functions which read data from input ports, specifies include files, calls required function name which has to be tested etc.
Example:

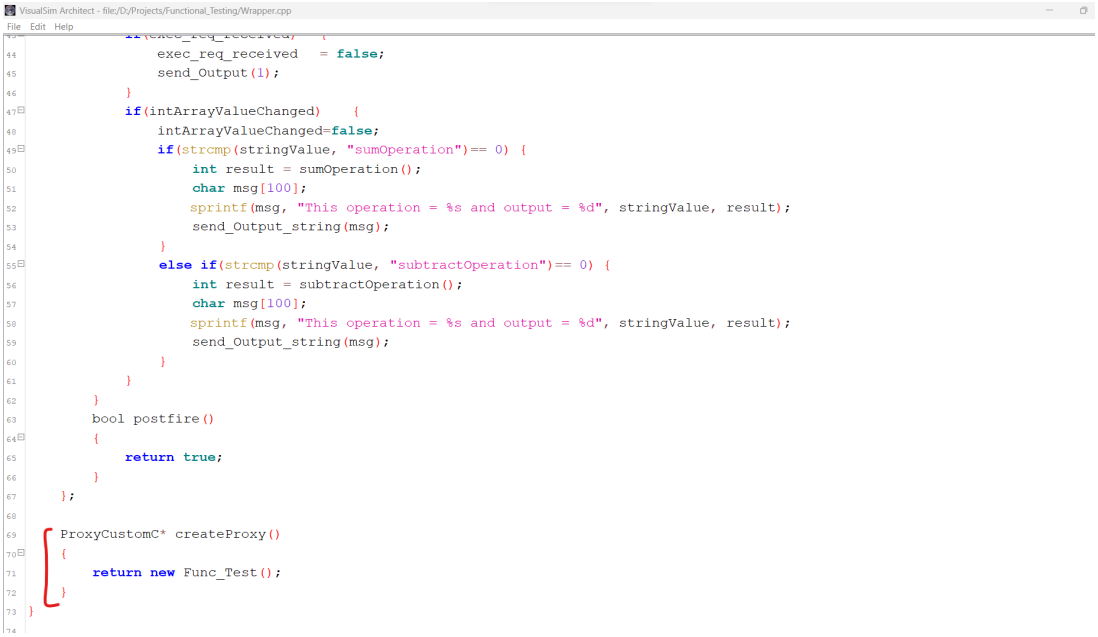


3. The Wrapper.cpp should be defined in a specific format.

```
#include "D:/Tools/VisualSim/VS_2320_j20/VS_AR/demo/Interfaces/UserNativeC/CustomC/Func_Test/ProxyFunc_Test.h"
#include <stdio.h>
#include "SW1.c"
#include "shared_variables.h"
extern int OP1_REG;
extern int OP2_REG;

namespace CustomC {

    class Func_Test : public ProxyFunc_Test{
    private:
        char* stringValue;
        int* intArrayValue;
        bool exec_req_received;
        bool intArrayValueChanged;

    public:
        void init()
        {
            exec_req_received=false;
            intArrayValueChanged=false;
            int intArray[] = {0, 0};
            intArrayValue = intArray;
            OP1_REG = 0;
            OP2_REG = 0;
        }

        void Input_received(char* data)
        {
            stringValue = data;
            //printf("Value = %s",stringValue);
```

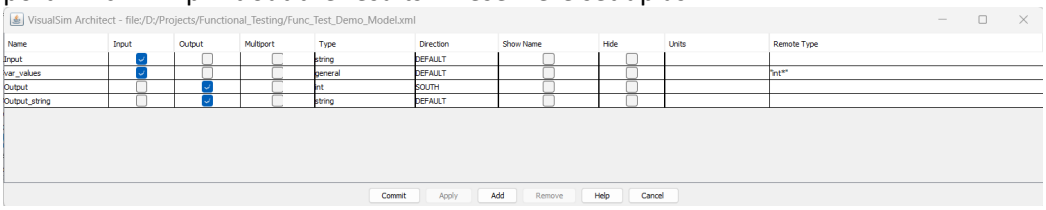
The making (1) refers to the location where the Machine generated include file is located. The base directory is VS_AR/demo/Interfaces/UserNativeC/CustomC/<Module Name>/ and the file name is Proxy<Module Name>.h
The Marking (2) refers to the namespace CustomC definition which encapsulate related classes and functions
The Marking (3) refers to the declaration of class “Func_Test” (Module Name) that inherits from “ProxyFunc_Test” (Proxy<Module Name>)

On top of the above updates, the following snippet needs to be included as well:



The Wrapper.cpp contains few other functions like void init() which will be executed at time 0.0 for initialization, void fire() which will be executed when an input comes in and void postfire() which is executed after the fire() method.

4. Now, we need to define the input and output ports to the Application_Interface.
Right click on the Application_Interface and do Customize-> Ports
Add ports as required. Here, in this example, I need one input port to read the function name that needs to be executed, one output port which will send a request to read the variable values, another input port which will read the variable values and an output port which will print out the results. These were set up as:



var_values is an input port which accepts array as input. So we need to specify the “Remote Type” column as well. We need to enter “int*” under Remote Column. For other data types, please refer to

VS_AR\demo\Interfaces\CustomC\ApplicationInterfaceTest4\
ApplicationInterfaceTest4.xml

5. Now, we need to update the Wrapper.cpp with the function definitions to read the data from the ports we have setup.

```
28          void Input_received(char* data)
29          {
30              stringValue = data;
31              exec_req_received=true;
32          }
33
34          void var_values_received(int* data, int length) {
35              intArrayValue = data;
36              OP1_REG     = intArrayValue[0];
37              OP2_REG     = intArrayValue[1];
38              intArrayValueChanged = true;
39          }
```

The above screenshot shows the format in which the ports needs to be defined. The input port called “Input” gets the function name “sumOperation” or “subtractOperation”. So the argument is char* data.
The input port called “var_values” gets the variable values {50,20} or {65,15} etc as input. 0th index is the value for OP1_REG and 1st index is the value for OP2_REG.

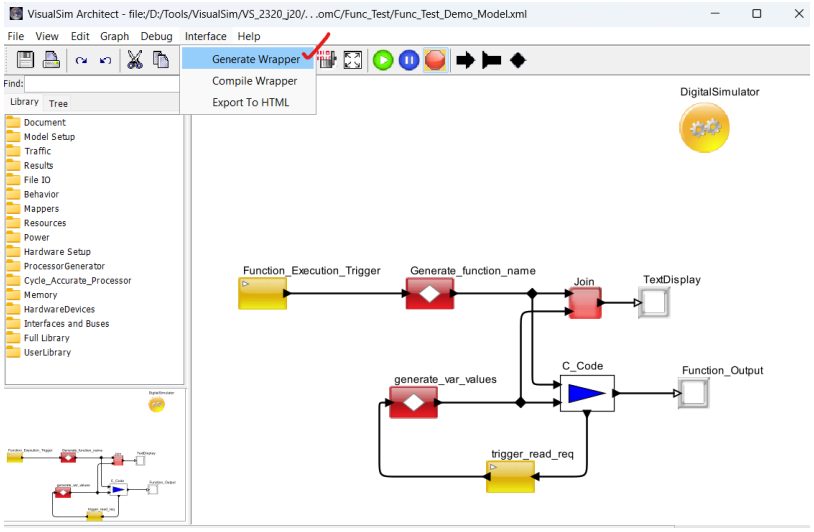
6. When an input comes in, for example on the var_values port, then the function void var_values_received(int* data, int length) will be executed and once completed, the void fire() function will be executed.

7. In this demo model, the “Input” port receives the function name which has to be executed and then we will send out a trigger to read the variable values.

   The variable values will be sent on var_values port which will be then used to execute the function we had received on “Input” port.

```
41      void fire()
42      {
43          if(exec_req_received)   {
44              exec_req_received   = false;
45              send_Output(1);                 // sending trigger to read variable values
46          }
47          if(intArrayValueChanged)    {       // variable values has been read, so now we can execute the function
48              intArrayValueChanged=false;
49              if(strcmp(stringValue, "sumOperation")== 0) {
50                  int result = sumOperation();
51                  char msg[100];
52                  sprintf(msg, "This operation = %s and output = %d", stringValue, result);
53                  send_Output_string(msg);
54              }
55              else if(strcmp(stringValue, "subtractOperation")== 0) {
56                  int result = subtractOperation();
57                  char msg[100];
58                  sprintf(msg, "This operation = %s and output = %d", stringValue, result);
59                  send_Output_string(msg);
60              }
61          }
62      }
```

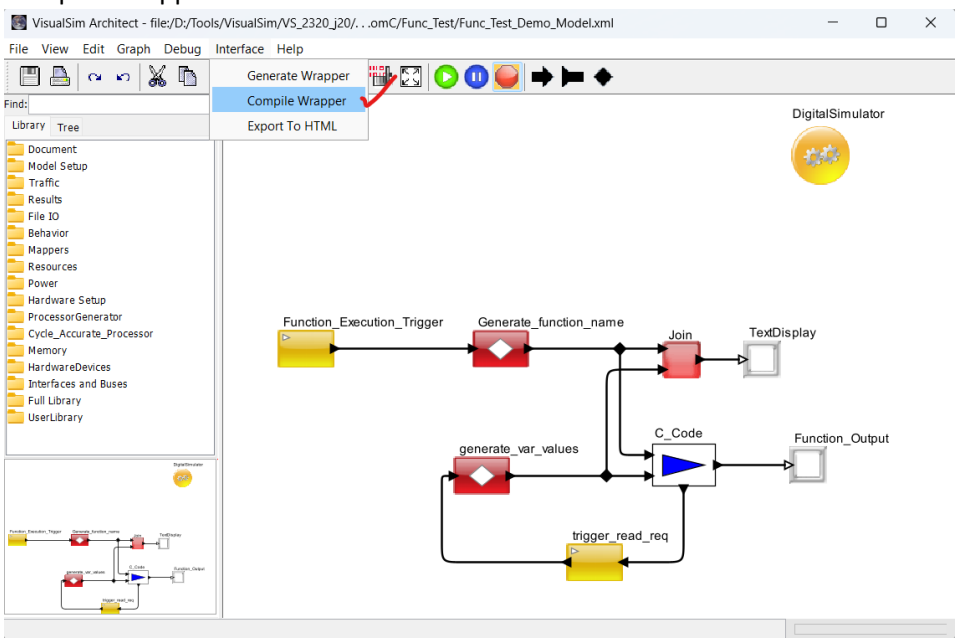
8. To run the demo model, we should first do – Generate Wrapper and Compile Wrapper.
9. Generate Wrapper:



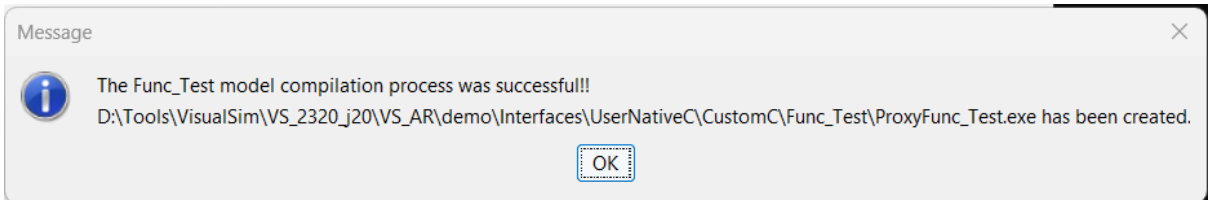
Check the commandline terminal to confirm the successful completion of wrapper generation.

```
Generate Wrapper .......... finished
```

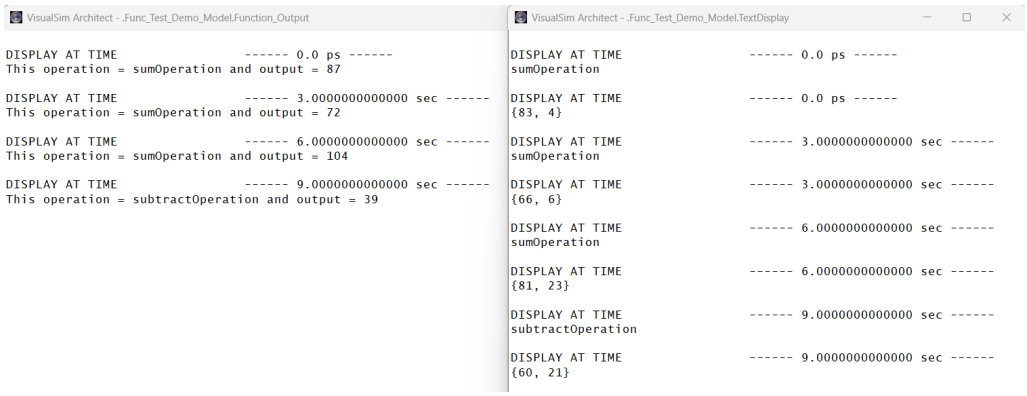
10. Compile Wrapper:



This will show compilation successful message:



11. Now we can run the demo model:



10 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.

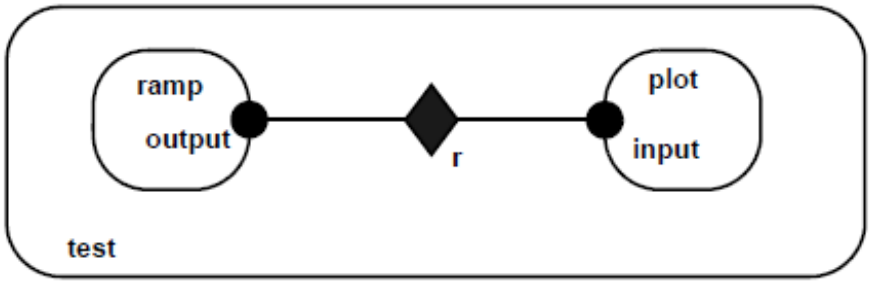
11 XML Details and File Parsing

11.1 Introduction

ModelBuilder stores models in ASCII files using an XML schema called Modeling Markup Language (MoML). MoML is the primary persistent file format for VisualSim models. It is also the primary mechanism for constructing models whose definition and execution is distributed over the network. This provides users the ability to directly edit and construct a MoML file. MoML is a modeling markup schema in the Extensible Markup Language (XML). It is intended for specifying interconnections of parameterized components. The filename extension can be ".xml" or ".moml" for MoML files. And the same XML file can be used in an applet2.
To get a quick start, try entering the following into a file called `test.xml`.

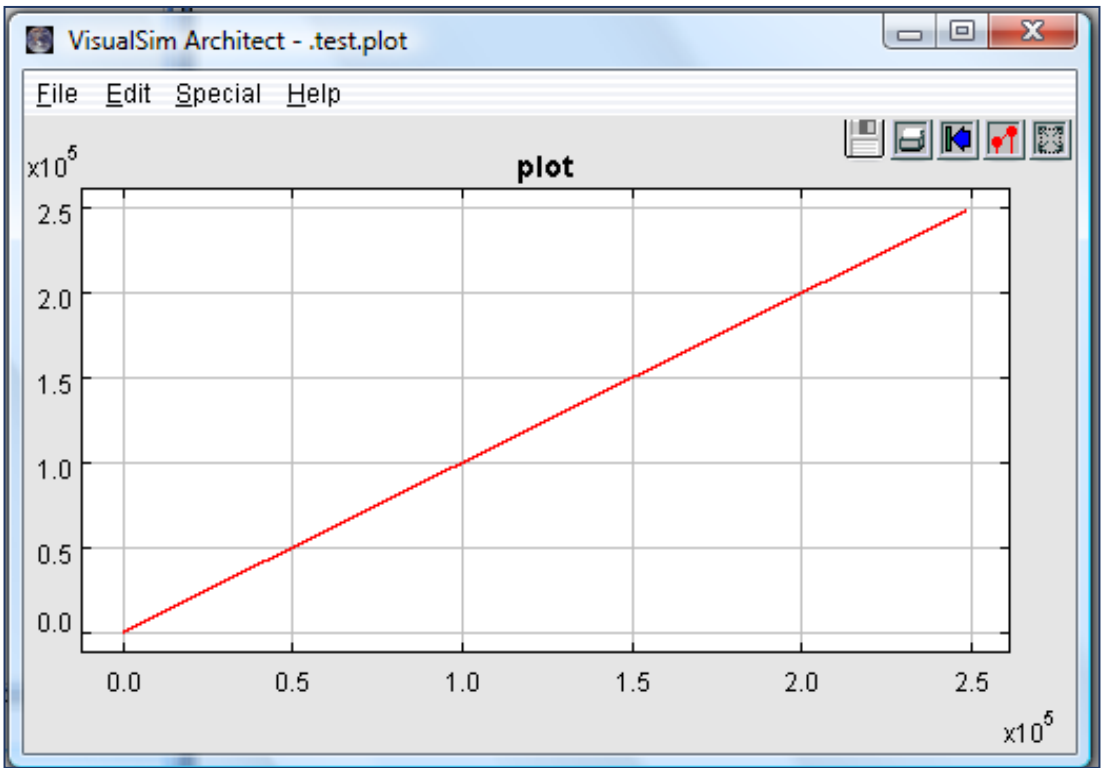
```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//Mirabilis Design//DTD MoML 1//EN"
"http://www.mirabilisdesign.com/xml/dtd/MoML_1.dtd">
<entity name="test" class="VisualSim.actor.TypedCompositeActor">
      <property name="Untimed_Digital"
            class="VisualSim.simulators.sdf.kernel.SDFDirector"/>
      <entity name="ramp" class="VisualSim.actor.lib.Ramp"/>
      <entity name="plot"
class="VisualSim.actor.lib.gui.SequencePlotter"/>
<relation name="r" class="VisualSim.actor.TypedIORelation"/>
<link port="ramp.output" relation="r"/>
<link port="plot.input" relation="r"/>
</entity>
```

This code defines a model in a top-level entity called "test". By convention, we use the same name for the top-level model and the file in which it resides. The top-level model is an instance of the VisualSim class `VisualSim.actor.TypedCompositeActor`. It contains a simulator, two entities, a relation, and two links. The model is depicted in Figure 50. When you execute the simulation, you get a window similar to the one in Figure 51. Enter "10" in the iterations box and click "Go" to execute the model for 10 iterations.



Simple example

The structure of the above MoML text is explained in detail in this chapter. A more interesting example is given in the $VS/doc/Training_Material/How_to_Tasks/XML. You may wish to refer to that example as you read about the details.

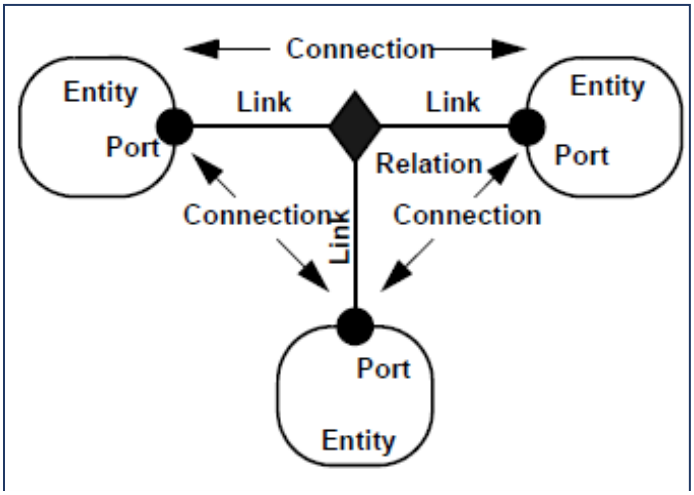


Simple example of a VisualSim model execution control window

MoML defines no semantics for an interconnection of components. It represents only the hierarchical containment relationships between entities with properties, their ports, and the connections between their ports. In VisualSim, the meaning of a connection (the semantics of the model) is defined by the simulator for the model, which is a property of the top-level entity. The simulator defines the semantics of the interconnection. MoML knows nothing about simulators except that they are instances of classes that can be loaded by the class loader and assigned as properties.

11.2 Toplogy

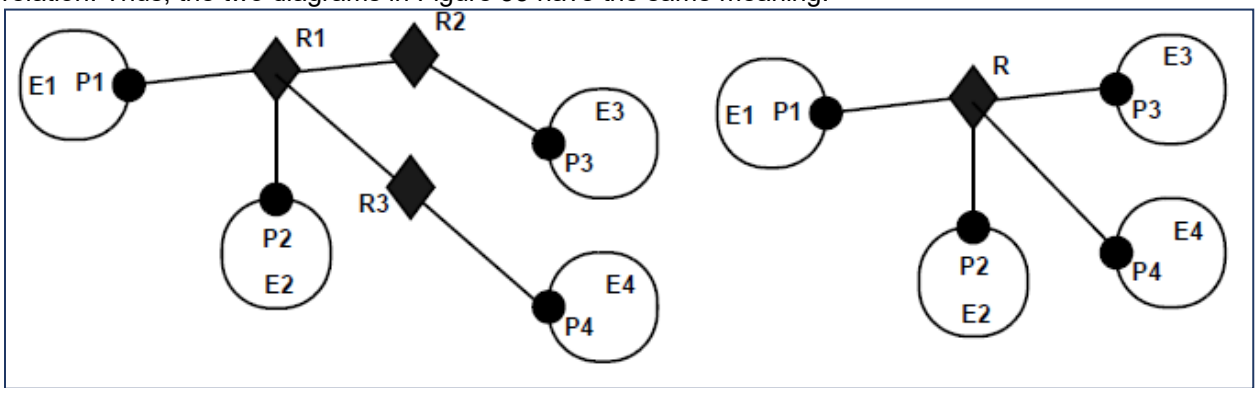
A model is given as a *topology*. A topology is a collection of *entities* and *relations*. Entities have *ports* and relations connect the ports. We use the term *connection* to denote the association between connected ports (or their entities), and the term *link* to denote the association between ports and relations or between relations and relations. Thus, a connection consists of one or more relations and two or more links. The visual notation is shown in Figure 52, where entities are depicted as rounded boxes, relations as diamonds, and entities as filled circles.



Visual notation and terminology

11.3 Relation Groups

Relations mediate connections between ports. For flexibility, particularly with visual syntaxes, the VisualSim abstract syntax permits any number of relations to be involved in any one connection. Figure 53 illustrates this. Relations may be linked to other relations. Any two relations that are linked are said to be members of the same *relation group*. Specifically, a relation group is a maximal set of linked relations. Semantically, a relation group has the same meaning as a single relation. Thus, the two diagrams in Figure 53 have the same meaning.



Relationship Groups

Relation GroupA relation group is a maximal set of linked relations. At the left, R1, R2, and R3 form a relation group. A relation group is a semantically identical to a single relation, so the two diagrams above have the same meaning.

11.4 Specification of a Model

In this section, we describe the XML elements that are used to define MoML models.

11.4.1 Data Organization

As with all XML files, MoML files have two parts - one defining the MoML language and one containing the model data. The first part is called the *document type definition*, or DTD. This dual specification of content and structure is a key XML innovation. The DTD for MoML is available in VisualSim at $VS/VisualSim/moml/MoML_DTD_1. If you are adept at reading these, it is a complete specification of the schema. However, as it is not particularly easy to read, we explain its key features here.

Every MoML file must either contain or refer to a DTD. The simplest way to do this is with the following file structure:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//Mirabilis Design//DTD MoML 1//EN"
"http://www.mirabilisdesign.com/xml/dtd/MoML_1.dtd">
<entity name="modelname" class="classname">
model definition ...
</entity>
```

Here, "*model definition*" is a set of XML elements that specify a clustered graph. The syntax for these elements is described in subsequent sections. The first line above is required in any XML file. It asserts the version of XML that this file is based on (1.0) and states that the file includes external references (in this case, to the DTD). The second and third lines declare the document type (model) and provide references to the DTD.

The references to the DTD above refer to a "public" DTD. The name of the DTD is

`-//Mirabilis Design//DTD MoML 1//EN`

which follows the standard naming convention of public DTDs. The leading dash "`-`" indicates that this is not a DTD approved by any standards body. The first field, surrounded by double slashes, is the name of the "owner" of the DTD, "`Mirabilis Design`".

The next field is the name of the DTD, "`DTD MoML 1`" where "`1`" indicates version 1 of the MoML DTD. The final field, "`EN`" indicates that the language assumed by the DTD is English. The VisualSim MoML parser requires that the public DTD be given exactly as shown, or it does not recognize the file as MoML. If a particular MoML tool does not have access to a local copy of the DTD, then it finds it at this web site.

The "entity" element may be replaced by a "class" element, as in:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//Mirabilis Design//DTD MoML 1//EN"
"http://www.mirabilisdesign.com/xml/dtd/MoML_1.dtd">
<class name="modelname" class="classname">
class definition ...
</class>
```

We say more about class definitions below.

11.5 Overview of XML

An XML document consists of the header tags "`<?xml ... ?>`" and "`<!DOCTYPE ... >`" followed by exactly one *element*. The element has the structure:

```
start tag
body
end tag
```

where the start tag has the form

```
<elementName attributes>
```

and the end tag has the form

```
</elementName>
```

The body, if present, can contain additional elements as well as arbitrary text. If the body is not present, then the element is said to be *empty* and it can optionally be written using the shorthand:

```
<elementName attributes/>
```

where the body and end tag are omitted.
The attributes are given as follows:

```
<elementName attributeName="attributeValue" .../>
```

Which attributes are legal in an element is defined by the DTD. The quotation marks delimit the value of the attributes, so if the attribute value needs to contain quotation marks, then they must be given using the special XML entity "`&quot;`" as in the following example:

```
<elementName attributeName="&quot;foo&quot;"/>
```

The value of the attribute is

```
"foo"
```

(with the quotation marks).
In XML "`&quot;`" is called an *entity*, creating possible confusion with our use of entity in VisualSim. In XML, an entity is a named storage unit of data. Thus, "`&quot;`" references an entity called "`quot`" that stores a double quote character.

11.6 Names and Classes

Most MoML elements have *name* and *class* attributes. The name is a handle for the object being defined or referenced by the element. In MoML, the same syntax is used to reference a pre-existing object as to create a new object. If a new object is being created, then the class attribute (usually) must be given. If a pre-existing object is being referenced, or if the MoML reader has a built-in default class for the element, then the class attribute is optional. If the class attribute is given, then the pre-existing object must be an instance of the specified class.

A name is either absolute or relative. Absolute names begin with a period "." and consist of a series of name fields separated by periods, as in ".x.y.z". Each name field can have alphanumeric characters, spaces, or the underscore "_" character. The first field is the name of the top-level model or class object. The second field is the name of an object immediately contained by that top-level. Any name that does not begin with a period is relative to the current context, the object defined or referenced by an enclosing element. The first field of such a name refers to or defines an object immediately contained by that object.

For example, inside of an object with absolute name ".x", the name "y.z" refers to an object with absolute name ".x.y.z". A name is required to be unique within its container. That is, in any given model, the absolute names of all the objects must be unique. There can be two objects named "z", but they must not be both contained by ".x.y". Not much more is said about classes. The class names that are used in the VisualSim implementation of MoML are always fully qualified Java class names. In addition, in VisualSim, a MoML file can be referenced as a class in the same way. In MoML, any named object can also have a distinct *display name*. The display name is used to present the object to the user in visualization and editing tools. It need not be unique and can consist of arbitrary text. It is specified by the *display* XML element, as in the following example.

```
<entity name="modelname" class="classname">
      <display name="arbitrary text here"/>
</entity>
```

11.7 Top-Level Entities

A very simple MoML file looks like this:

```
<?xml version="1.0" standalone="no"?> <!DOCTYPE entity
PUBLIC "-//Mirabilis Design//DTD MoML_1//EN"
```

```
"http://www.mirabilisdesign.com/xml/dtd/moml.dtd">

<entity name="modelname" class="classname"> </entity>
```

The *entity* element has name and class attributes, and defines a VisualSim model. This value of the class attribute must be a class that is instantiable by the MoML tool. For example, in VisualSim, we can define a model with:

```
<?xml version="1.0" standalone="no"?> <!DOCTYPE entity PUBLIC "-
//Mirabilis Design//DTD MoML_1//EN"
"http://www.mirabilisdesign.com/xml/dtd/moml.dtd">

<entity name="VSmodel"
class="VisualSim.actor.TypedCompositeActor"> </entity>
```

Here, `VisualSim.actor.TypedCompositeActor` is a class. Most useful models are instances of `VisualSim.kernel.CompositeEntity` or a derived class. `TypedCompositeActor`, as in the above example, is derived from `CompositeEntity`.

11.8 Entity Element

A model typically contains entities, as in the following VisualSim example:

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE entity PUBLIC "-//Mirabilis Design//DTD MoML_1//EN"
"http://www.mirabilisdesign.com/xml/dtd/moml.dtd">

<entity name="VSmodel"
class="VisualSim.actor.TypedCompositeActor">

   <entity name="source" class="VisualSim.actor.lib.Ramp"/>

   <entity name="sink"
  class="VisualSim.actor.lib.SequencePlotter"/>

 </entity>
```

Notice the common XML shorthand here of writing "`<entity ... />`" rather than "`<entity ...></entity>`." Of course, the shorthand only works if there is nothing in the body of the entity element.
An entity can contain other entities, as shown in this example:

```
<entity name="VSmodel" class="VisualSim.actor.TypedCompositeActor">

  <entity name="container"
class="VisualSim.actor.TypedCompositeActor">

   <entity name="source" class="VisualSim.actor.lib.Ramp"/>

  </entity>

</entity>
```

An entity must specify a class unless the entity already exists in the containing entity or model. The name of the entity reflects the container hierarchy. Thus, in the above example, the *source* entity has the full name " `.VSmodel.container.source`". The definition of an entity can be distributed in the MoML file. After it is created, it can be referred to again by name as follows:

```
<entity name="top"
class="classname"> <entity name="x"
class="classname"/> ... <entity
name="x"><property name="y">
</entity></entity>
```

MoML files support multiple containment, as in the following example:

```
<entity name="top" class="classname">
      <entity name="x" class="classname"/>
            ...
            <entity name="y" class="classname">
                  <entity name=".top.x"/>
            </entity>
</entity>
```

Here, the element named "x" appears both in "top" and in ".top.y", i.e. the same instance appears in two different places. Thus, it would have two full names, ".top.x" and ".top.y.x". However, VisualSim does not support this, as it implements a strict container relationship, where an object can have only one container. Thus, attempting to parse the above MoML results in an exception being thrown.

11.9 Properties

Entities (and some other elements) can be parameterized. There are two mechanisms. The simplest one is to use the *property* element:

```
<entity name="source" class="VisualSim.actor.lib.Ramp">
      <property name="init" value="5"
class="VisualSim.data.expr.Parameter"/>
</entity>
```

The property element has a name, at minimum (the value and class are optional). It is common for the enclosing class to already contain properties, in which case the property element is used only to set the value. For example:

```
<entity name="source"
class="VisualSim.actor.lib.Ramp">

      <property name="init" value="5"/>

</entity>
```

In the above, the enclosing object (*source*, an instance of `VisualSim.actor.lib.Ramp`) must already contain a property with the name *init*. This is typically how library components are parameterized. In VisualSim, the value of a property may be an expression, as in "`PI/50`". The expression may refer to other properties of the containing entity or of its container. Note that the expression language is not part of MoML, but is rather part of VisualSim. In MoML, a property value is simply an uninterpreted string. It is up to a MoML tool, such as VisualSim, to interpret that string.

A property can be declared without a class and without a pre-existing property if it is a *pure property*, one with only a name and no value. For example:

```
<entity name="source"
class="VisualSim.actor.lib.Ramp"> <property
name="abc"/></entity>
```

A property can also contain a property, as in

```
<property name="x" value="5">
<property name="y"
value="10"/></property>
```

A second, much more flexible mechanism is provided for parameterizing entities. The *configure* element can be used to specify a relative or absolute URL pointing to a file that configures the entity, or it can be used to include the configuration information in line. That information need not be MoML information. It need not even be XML, and can even be binary encoded data (although binary data cannot be in line; it must be in an external file). For example,

```
<entity name="sink"
class="VisualSim.actor.lib.SequencePlotter"> <configure
source="url"/></entity>
```

Here, *url* can give the name of a file containing data, or a URL for a remote file. (For the SequencePlotter actor, that external data has PlotML syntax. (PlotML is another XML schema for configuring plotters.). Configure information can also be given in the body of the MoML file as follows:

```
<entity name="sink"
class="VisualSim.actor.lib.SequencePlotter">
<configure>configure information </configure></entity>
```

With the above syntax, the configure information must be textual data. It can contain XML markup with only one restriction: if the tag "`</configure>`" appears in the textual data, then it must be preceded by a matching "`<configure>`". That is, any configure elements in the markup must have balanced start and end tags[7].

You can give both source attribute and in-line configuration information, as in the following:

```
<entity name="sink"
class="VisualSim.actor.lib.SequencePlotter"> <configure
source="url">configure information </configure></entity>
```

In this case, the file data is passed to the application first, followed by the in-line configuration data.

In VisualSim, the configure element is supported by any class that implements the Configurable interface. That interface defines a configure() method that accepts an input stream. Both external file data and in-line data are provided to the class as a character stream by calling this method. There is a subtle limitation with using markup within the configure element. If any of the elements within the configure element match MoML elements, then the MoML DTD is applied to assign default values, if any, to their attributes. Thus, this mechanism works best if the markup within the configure element is not using an XML schema that happens to have element names that match those in MoML. Alternatively, if it does use MoML element names, then those elements are used with their MoML meaning. This limitation can be fixed using XML namespaces, something we will eventually implement.

5.9.1 Doc Element

Some elements can be documented using the *doc* element. For example,

```
<entity name="source" class="VisualSim.actor.lib.Ramp">
```

[7] XML allow markup to be included in arbitrary data as long as it appears within either a processing instruction or a CDATA body. However, for reasons that would only baffle anyone familiar with modern programming languages, processing instructions and CDATA bodies cannot be nested within one another. The MoML configure element can be nested, so it offers a much more flexible mechanism than the standard ones in XML.

```
        <property name="init" value="5">

              <doc>Initialize the ramp above the default
              because... </doc>

        </property>

              <doc> This actor produces an increasing sequence
        beginning with 5. </doc>

    </entity>
```

formatting within the doc element: With the above syntax, the documentation information must be textual data. It can include markup, as in the following example, which uses XHTML[8].

```
    <entity name="source"
    class="VisualSim.actor.lib.Ramp"> <doc><H1>Using
    HTML</H1>Text with <I>markup</I>.</doc></entity>
```

An alternative method is to use an XML processing instruction as follows:

```
    <entity name="source" class="VisualSim.actor.lib.Ramp">
         <doc><?xhtml <H1>Using HTML</H1>Text with <I>markup</I>.?>
         </doc>
    </entity>
```

This requires that any utility that uses the documentation information be able to handle the xhtml processing instruction, but it makes it very clear that the contents are XHTML. However, for reasons we do not understand, XML does not allow processing instructions to be nested and so this technique has its limitations. More than one doc element can be included in an element. To do this, give each doc element a name, as follows:

```
    <entity name="entityname"
    class="classname"> <doc name="docname">doc
    contents </doc></entity>
```

The name must not conflict with any preexisting property. If a doc element or a property with the specified name exists, then it is removed and replaced with the property. If no name is given, then the doc element is assigned the name "_doc".

A common convention, used in VisualSim, is to add doc elements with the name "tooltip" to define a tooltip for GUI views of the component. A tooltip is a small window with short documentation that pops up when the mouse lingers on the graphical component.

Note that the same limitation of using markup within configure elements also applies to doc elements.

5.9.2 Ports

An entity can declare a port:

```
    <entity name="A"
    class="classname"> <port
    name="out"/></entity>
```

In the above example, no class is given for the port. If a port with the specified name already exists in the class for entity `A`, then that port is the one referenced. Otherwise, a new port is created in VisualSim by calling the newPort() method of the container. Alternatively, we can specify a class name, as in

```
    <entity name="A" class="classname">
    <port name="out"
    class="classname"/></entity>
```

[8] XHTML is HTML with additional constraints so that it conforms with XML syntax rules. In particular, every start tag must be matched by an end tag, something that ordinary HTML does not require (but fortunately, does allow).

In this case, a port is created if one does not already exist. If it does already exist, then its class is checked for consistency with the declared class (the pre-existing port must be an instance of the declared class). In VisualSim, the typical classname for a port is

```
VisualSim.actor.TypedIOPort
```

In VisualSim, the container of a port is required to be an instance of VisualSim.kernel.Entity or a derived class.
It is often useful to declare a port to be an input, an output, or both. To do this, enclose in the port a property named “input” or “output” or both, as in the following example:

```
<port name="out"
class="VisualSim.actor.IOPort"> <property
name="output"/></port>
```

This is an example of a pure property. Optionally, the property can be given a boolean value, as in

```
<port name="out"
class="VisualSim.actor.IOPort"> <property
name="output" value="true"/></port>
```

The value can be either “true” or “false”, where the latter defines the port to not be an output. A port can be defined to be both an input and an output, as follows

```
<port name="out"
class="VisualSim.actor.IOPort"> <property
name="output" value="true"/> <property
name="input" value="true"/></port>
```

It is also sometimes necessary to declare that a port is a multiport. To do this, enclose in the port a property named “multiport” as in the following example:

```
<port name="out"
class="VisualSim.actor.IOPort"> <property
name="multiport"/></port>
```

The enclosing port must be an instance of IOPort (or a derived class such as TypedIOPort), or else the property is treated as an ordinary property. As with the input and output attribute, the multiport property can be given a boolean value, as in

```
<port name="out"
class="VisualSim.actor.IOPort"> <property
name="multiport" value="true"/></port>
```

If a port is an instance of TypedIOPort (for library actors, most are), then you can set the type of the port in MoML as follows:

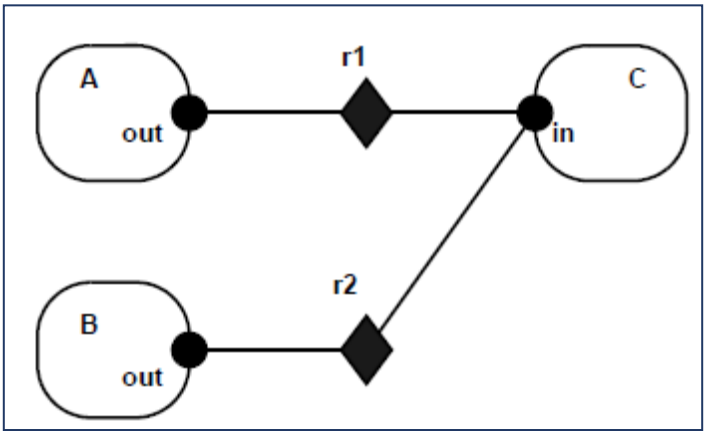
```
<port name="out"
class="VisualSim.actor.IOPort"> <property
name="type" value="double"
            class="VisualSim.actor.TypeAttribute
"/> </port>
```

This is occasionally useful when you need to constrain the types beyond what the built-in type system takes care of. The names of the built-in types are (currently) boolean, booleanMatrix, complex, complexMatrix, double, doubleMatrix, fix, fixMatrix, int, intMatrix, long, longMatrix, unsignedByte, unsignedByteMatrix, object, string, and general. These are defined in the class VisualSim.data.type.BaseType.

5.9.3 Relations and Links

To connect entities, you create relations and links. The following example describes the topology shown in Figure 54:

```
<entity name="top" class="classname">
<entity name="A" class="classname">
      <port name="out"/>
</entity>
<entity name="B" class="classname">
      <port name="out"/>
</entity>
<entity name="C" class="classname">
      <port name="in">
            <property name="multiport"/>
      </port> </entity>
<relation name="r1" class="classname"/>
<relation name="r2" class="classname"/>

<link port="A.out" relation="r1"/>
<link port="B.out" relation="r2"/>
<link port="C.in" relation="r1"/>
<link port="C.in" relation="r2"/>
</entity>
```



Example topology.

In VisualSim, the typical classname for a relation would be `VisualSim.actor.TypedIORelation`. The classname may be omitted, in which case the newRelation() method of the container is used to create a new relation. The container is required to be an instance of VisualSim.kernel.CompositeEntity, or a derived class. As usual, the class attribute may be omitted if the relation already exists in the containing entity.

The `link` elements may appear anywhere in the body of an `entity` or `class` element. They will be processed after all the contained entities, properties, and relations are created. However, the order of the link elements relative to each other does matter. Notice in this example that there are two distinct links to `C.in` from two different relations. The order of these links may be important to a MoML tool, so any MoML tool must preserve the order in which they are specified, as VisualSim does. We say that C has two links, indexed 0 and 1.

The `link` element can explicitly give the index number at which to insert the link. For example, we could have achieved the same effect above by saying

```
      <link port="C.in" relation="r1"
```

```
insertAt="0"/> <link port="C.in"
relation="r2" insertAt="1"/>
```
Whenever the insertAt option is not specified, the link is always appended to the end of the list of links.
When the insertAt option is specified, the link is inserted at that position, so any pre-existing links with larger indices will have their index numbers incremented. For example, if we do

```
<link port="C.in" relation="r1"
insertAt="0"/> <link port="C.in"
relation="r2" insertAt="2"/> <link
port="C.in" relation="r3" insertAt="1"/>
```
then there will be a link to r1 with index 0, a link to r2 with index 2 (note! not 1), and a link to r3 with index 1.
If the specified index is beyond the existing number of links, then null links (that is links to nothing) are created to fill in. So for example, if the first link we create is given by

```
<link port="C.in" relation="r2" insertAt="1"/>
```
then the port has *two* links, not one, but the first one is an empty link. If we then say

```
<link port="C.in" relation="r2"/>
```
then the port has *three* links, with the first one being empty. If we then say

```
<link port="C.in" relation="r2" insertAt="0"/>
```
then there are *four* links, with the *second* one being empty. Normally, it is not necessary in MoML to specify whether a link occurs on the inside of a port or on the outside. This can be determined automatically by identifying the relation. For example, in figure 54, port P4 is linked on the inside to relation R2 and on the outside to relations R3 and R4.
However, close examination of the DTD reveals that the relation attribute is optional. If the relation attribute is not present, then a null link is inserted. However, if you do not specify a relation, then there is no way to determine whether an inside null link or an outside null link was intended. MoML defines the default to be an outside null link. To specify an inside null link, use the insertInsideAt attribute. For example, to insert a null link on the inside of P4 in Figure 54 prior to the link to R2, use:

```
<entity name="E0.E1">
      <link port="P4.in" insertInsideAt="0"/>
</entity>
```
Note that the index number is not the same thing as the channel number in VisualSim. In VisualSim, a relation may have a width greater than one, so a single link may represent more than one channel (actually, it could even represent zero channels if that relation is not linked to another ports).
There is no significance to the order in which relations are linked, unlike the order in which ports are linked to relations. Unlike links between relations and ports, there is no significance to multiple links between the same relations. Thus, in the following MoML, the second line is redundant and can be omitted with no change in meaning:

```
<link relation1="R1"
relation2="R2"/> <link
relation1="R1" relation2="R2"/>
```
The same is not true of links between relations and ports. A relation group is a maximal set of linked relations. If a port links to multiple relations within the same relation group, then the meaning is exactly the same as if the port were linked to just one relation in the group. The order in which a link between ports and relations in a group is made matters, but the order in which links between relations are made does not. Thus, a relation group is semantically equivalent to a

single relation where the links between ports and the single relation are made in the same order as the links between a port and relations in a relation group.

11.10 Classes

So far, entities have been instances of externally defined classes accessed via a class loader. They can also be instances of classes defined in MoML. To define a class in MoML, use the *class* element, as in the following example:

```
<?xml version="1.0" standalone="no"?> <!DOCTYPE class PUBLIC
"-//Mirabilis Design//DTD MoML
1//EN""http://Www.mirabilisdesign.com/xml/dtd/moml.dtd">
<class name="Gen"
extends="VisualSim.actor.TypedCompositeActor">
<entity name="ramp" class="VisualSim.actor.lib.Ramp"> <port
name="output"/> <property name="step"
value="2*PI/50"/></entity> <entity name="sine"
class="VisualSim.actor.lib.TrigFunction"><port
name="input"/> <port name="output"/></entity> <port
name="output" class="VisualSim.actor.TypedIOPort"/>
<relation name="r1"
class="VisualSim.actor.TypedIORelation"/> <relation
name="r2" class="VisualSim.actor.TypedIORelation"/> <link
port="ramp.output" relation="r1"/> <link port="sine.input"
relation="r1"/> <link port="sine.output" relation="r2"/>
<link port="output" relation="r2"/></class>
```

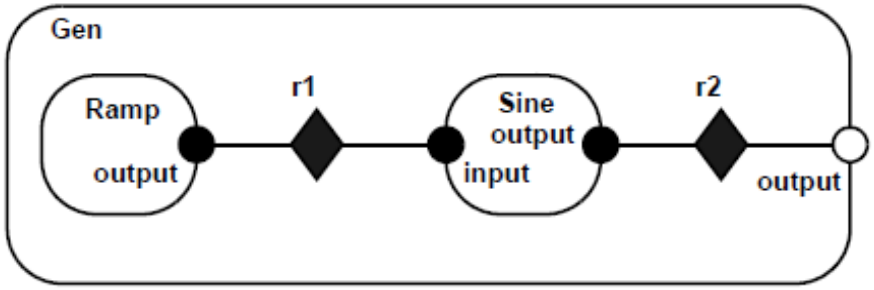
The class element may be the top-level element in a file, in which case the DOCTYPE should be declared as “class” as done above. It can also be nested within a model. The above example specifies the topology shown in figure 55. After it is defined, it can be instantiated as if it were a class loaded by the class loader:

```
<entity name="instancename" class="classname"/>
```

or

```
<entity name="instancename" class="classname" source="url"/>
```

The first form can be used if the class definition can be found from the classname. There are two ways that this could happen. First, the classname might match a class definition that is in scope; a class definition is in scope if the class is defined within the same container where the entity is being created, or within the container of that container, or the container of that container, and so on.



Sine wave generator topology.

That is once a class is defined, it can be instantiated anywhere (deeply) within the container in which is defined. Second, the *classname* might be sufficient to find the class definition in a file, much the way Java classes are found. For example, if the classname is VisualSim.actor.lib.Sinewave and the class is defined in the file $PTII/VisualSim/actor/lib/Sinewave.xml, then there is no need to use the second form to specify the URL where the class is defined. Specifically, the CLASSPATH is searched for a file matching the classname. By convention, the file defining the class has the same name as the class, with the extension " .xml" or ".moml".An example of the first of these techniques is given below:

```
<?xml version="1.0" standalone="no"?> <!DOCTYPE entity PUBLIC
"-//Mirabilis Design//DTD MoML
1//EN""http://Www.mirabilisdesign.com/xml/dtd/moml.dtd">
<entity name="top"
class="VisualSim.actor.TypedCompositeActor"><class name="Gen"
extends="VisualSim.actor.TypedCompositeActor"> class definition
...
</class> <entity name="inside"
class="VisualSim.actor.TypedCompositeActor"><entity
name="instance" class="Gen"/> </entity></entity>
```

The ability to give a URL as the source of a class definition is very powerful. It means that a model may be build from component libraries that are defined worldwide. There is no need to localize these. Of course, referencing a URL means the usual risks that the link becomes invalid. It is our hope that reliable and trusted sources of components emerge and not allow this to happen.

The Gen class given at the beginning of this subsection generates a sine wave with a period of 50 samples. It is not all that useful without being parameterized. Let us extend it and add properties:

```
<class name="Sinegen" extends="Gen">
  <property name="samplingFrequency"value="8000.0"
  class="VisualSim.data.expr.Parameter">
        <doc>The sampling frequency in Hertz.</doc>
  </property>
  <property name="frequency"value="440.0"
  class="VisualSim.data.expr.Parameter">
        <doc>The frequency in Hertz.</doc>
  </property>
<property name="ramp.step"value="frequency*2*PI/samplingFrequency">
  <doc>Formula for the step size.</doc>
</property>
<property name="ramp.init" value="phase"> </property>
</class>
```

This class extends Gen by adding two properties, and then sets the properties of the component entities to have values that are expressions.

11.11 Inheritance

MoML supports inheritance by permitting the user to extend existing classes. For example, consider the following MoML file:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//Mirabilis Design//DTD MoML
1//EN""http://Www.mirabilisdesign.com/xml/dtd/moml.dtd">
<entity name="top" class="VisualSim.kernel.CompositeEntity">
 <class name="base" extends="VisualSim.kernel.CompositeEntity">
       <entity name="e1" class="VisualSim.kernel.ComponentEntity">
       </entity>
 </class>
<class name="derived" extends="base">
 <entity name="e2" class="VisualSim.kernel.ComponentEntity"/>
</class>
<entity name="instance" class = ".top.derived"/>
</entity>
```

Here, the “derived” class extends the “base” class by adding another entity to it, and “instance” is an instance of derived class. The class “derived” can also give a source attribute, which gives a URL for the source definition.

A derived class (or subclass) can contain additional entities, relations, ports, and links. However, it cannot remove entities, relations, ports or links defined in the base class. Moreover, it cannot add links that are exclusively between ports and relations defined in the base class. New links must involve either a port or a relation that is new in the derived class.

11.12 Simulators

VisualSim requires the specification of a Simulator associated with a model, an entity, or a class. The Simulator is a property of the model. The following example gives Digital semantics to a VisualSim model:

```
<entity name="top" class="VisualSim.actor.TypedCompositeActor">
<property name="Simulator"
class="VisualSim.simulators.de.kernel.DEDirector>
       <property name="stopTime" value="100.0"/>
 </property>
 ...
 </entity>
```

This example also sets a property of the Simulator. The name of the Simulator is not important, except that it cannot collide with the name of any other property in the model.

11.13 Input Element

It is possible to insert MoML from another file or URL into a particular point in your model. For example:

```
<entity name="top" class="...">

       <entity name="a"
      class="...">

            <input
          source="url"/>
```

```
        </entity>

    </entity>
```

This takes the contents of the URL specified in the source attribute of the input element and places them inside the entity named “a”. The base of the current document (the one containing the import statement) is used to interpret a relative URL, or if the current document has no base, then the current working Simulator is used, or if that fails, the current CLASSPATH.

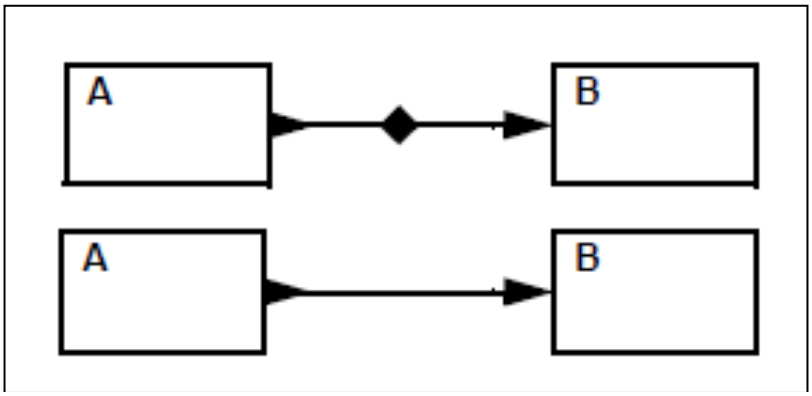
11.14Annotations for Visual Rendering

The abstract syntax of MoML, clustered graphs, is amenable to visual renditions as bubble and arc diagrams or as block diagrams. To support tools that display and/or edit MoML files visually, MoML allows a relation to optionally have a *vertex* which gives it a specific physical location in a visual syntax. For example:

```
<relation name="r" class="VisualSim.actor.TypedIORelation">
      <vertex name="v1" class="classname" value="location"/>
</relation>
```

A visual rendition (such as that created by ModelBuilder) would render the relation with a located icon, rather than simply as a wire between ports. Figure 56 illustrates two links, the top one of which has a vertex, and the bottom one of which does not. The two are semantically identical, but obviously the rendition is different. A vertex is given as follows:

```
<vertex name="v1" class="VisualSim.moml.Vertex" value="{184.0,
93.0}"/>
```



Vertex Example

The above is an example showing how a vertex contained by a relation results in a visual representation of the relation at a specific location (top) vs. a relation with no vertex that is rendered as a wire (bottom). The two connections have the same meaning.

This indicates that the vertex should be rendered at the location 184.0, 93.0.
VisualSim uses ordinary MoML properties to specify other visual aspects of a model. First, an entity can contain a location property, which is a hint to a visual renderer, as follows:

```
<entity name="ramp" class="VisualSim.actor.lib.Ramp">
      <property name="location" class="VisualSim.moml.Location"
      value="50.0, 50.0"/>
</entity>
```

This suggests to the visual renderer that the Ramp actor should be drawn at location 50.0, 50.0.