The HWSim Simulation Language Reference Manual

Sean Halle

May 19, 2012

Contents

1 Introduction and Overview **5**
1.1 The Philosophy of the Language 5
1.2 Overview of Usage of the Language 5

2 Commands and Usage **7**
2.1 Configuration File 7
2.2 Netlist Creation 7
2.3 Activities, Behavior, Timing, and Communications 10
2.3.1 ActivityType and ActivityInst 10
2.3.2 Behavior Functions and Ending Activities 11
2.3.3 Activity Timing Functions 11
2.3.4 Calculating the time-in-flight of a communication path 12

1 Introduction and Overview

1.1 The Philosophy of the Language

A simulator consists of two parts: behavior logic, and a separate underlying implementation of simulated time. Many simulators tightly integrate the behavior logic with the code of the simulator itself.

In contrast, HWSim takes the approach of providing a separate language that implements simulated time. One creates elements, each of which has an independent timeline, and sends communications between them. The communications trigger behaviors in the elements, which then send more communications.

What HWSim provides is assurance that the memory image of the Host machine executing the behavior is always consistent with the memory image that should exist due to the simulated-time ordering. This is valuable on multi-core machines, as it allows full usage of the parallel Host hardware.

Providing such consistency is tricky because, in general, a given core takes turns executing behavior of different elements. If not carefully controlled, behavior will be executed in a different order in physical time than in simulated time. Such out of order execution can cause different results to be calculated on the Host than should be calculated by the simulated hardware.

HWSim tracks the start and end of simulated-time activities and implements an algorithm to ensure the state of each element stays consistent with the simulated-time order.

1.2 Overview of Usage of the Language

Elements, and the connections between them, are specified in a netlist. Behavior and timing of the elements are provided as functions, which are attached to the elements. A behavior can call an HWSim construct to send a communication on an out-port. The out-port connects to the in-port of another element. The destination in-port has an activity-type attached, which contains pointers to the behavior and timing functions. Reception of the communication then triggers the functions in the destination element.

The language is supplied as a library, divided into calls for a main program to use, and calls for use within the behavior and timing functions.

The main program is currently (as of May 2012) charged with constructing the netlist and passing it to HWSim. It is also charged with handling results, although services are supplied to transform HWSim results into formats for various visualizers like Paraver.

The hardware behavior and timing functions are collected into a separate directory

and pointed to by the netlist. The behavior and timing functions for a particular kind of element are grouped in informal ways, such as defining them in the same file, or using naming schemes. The functions are attached to elements inside the netlist creator.

The hardware behavior is defined separately, then referred to by a netlist. The netlist is created within a main C program, then passed to an HWSim call that initiates the simulation. The main thread suspends until simulation completes, then calls HWSim services to transform the results to the form for the visualizers.

2 Commands and Usage

This chapter gives precise information about the various calls available in HWSim, and how to use them. As of May 2012, we take the shortcut of defining these through an example of their usage. The example is a circuit with two elements that are cross-coupled. The output from one goes to the input of the other. One of them is connected to the reset signal, triggers it's behavior, which is simply to send a communication on its outport. That triggers the same behavior in the other, and so on, back and forth.

2.1 Configuration File

The configuration file states the amount of simulated time and the name of the file to save the generated traces in.

```
int, SimulationEndTime, 100

string, resultTraceFileName,
   data_files/HWSim_results/Ping_Pong_results/test_config_results.txt
```

2.2 Netlist Creation

Here is an example of netlist creation:

The circuit has two elements, each with one input port, one output port, and a single activity-type. The elements are cross-coupled, so output port of one connects to input port of the other. The input port has the activity-type attached as its trigger. The activity is empty, and just sends a NULL message on the output port. The activity's duration in simulated time and the resulting communication's flight duration in simulated time are both constants.

Note that HWSimElem data type is generic. An elem is specialized by declaring inports and outports, and by connecting activity types to in-ports. Behavior is attached to an element by attaching activity types to in-ports of the element.

First, here is the top-level function that creates and returns the netlist structure:

```
HWSimNetlist *
createPingPongNetlist()
 { HWSimNetlist       *netlist;
   HWSimElem         **elems;
   HWSimActivityType **activityTypes;
   HWSimCommPath     **commPaths;
```

```
int32               numElems, numActivityTypes, numCommPaths;
```

The first thing to do is create the netlist structure, which holds three things: element structs, activity type structs, and communication path structs. It also has two collections of pointers to the traces collected during the run, but these are handled internally by HWSim.

```
netlist = malloc( sizeof(HWSimNetlist) );

numElems         = 2;
elems            = malloc( numElems * sizeof(HWSimElem *) );

numCommPaths     = 2;
commPaths        = malloc( numCommPaths * sizeof(HWSimCommPath *) );

numActivityTypes = 1;
activityTypes    = malloc( numActivityTypes * sizeof(HWSimActivityType *) );

netlist->numElems         = numElems;
netlist->elems            = elems;
netlist->numCommPaths     = numCommPaths;
netlist->commPaths        = commPaths;
netlist->numActivityTypes = numActivityTypes;
netlist->activityTypes    = activityTypes;
```

Now, create the activity types. During the run, an activity instance is created each time a communication arrives on an in-port. The activity instance is a data structure that points to the activity type. The activity type holds the pointers to the behavior and timing functions.

```
   //have to create activity types before create elements
   //PING_PONG_ACTIVITY is just a #define for readability
netlist->activityTypes[PING_PONG_ACTIVITY] = createPingPongActivityType();
```

Next, create the elements, and pass the netlist structure to the creator. It will take pointers to activity types out of the netlist and place them into the in-ports of the elements.

```
elems[0] = createAPingPongElem( netlist ); //use activity types from netlist
elems[1] = createAPingPongElem( netlist );
```

Now, the reset in-port of one of the elements has to be set up to trigger an activity. Every element has a reset in-port, but normally they are set to NULL activity type. Here, we want only one of the two elements to have an activity triggered when the reset signal is sent to start the simulation.

Note that during initialization, all the elements become active, each with its own timeline, but unless an activity is triggered in them they remain idle, with their timeline suspended and not making progress. Only ones that have an activity type attached to their reset in-port will begin to do something in simulated time when simulation starts.

```
   //make reset trigger an action on one of the elements
elems[1]->inPorts[-1].triggeredActivityType =
           netlist->activityTypes[PING_PONG_ACTIVITY];
```

Now, connect the elements together by creating commPath structures. A comm path connects the out-port of one element to the in-port of another. A given port may have many comm paths attached. However, an in-port has only one kind of activity type attached, and all incoming communications fire that same activity. There are multiple kinds of activity, including kinds that have no timing, and so can act as a dispatcher. These end themselves with a continuation activity, which is chosen according to the code in the behavior function. So, a commPath only connects an out port to an in port.

This code sets fixed timing on the comm paths. It also uses a macro for setting the connections. The format is: sending elem-index, out-port, dest elem-index, in-port:

```
      //elem 0, out-port 0 to elem 1, in-port 0
   commPaths[0]= malloc(sizeof(HWSimCommPath));
   setCommPathValuesTo(commPaths[0],0,0,1,0);
   commPaths[0]->hasFixedTiming  = TRUE;
   commPaths[0]->fixedFlightTime = 10; //all time is stated in (integer) units

      //elem 1, out-port 0 to elem 0, in-port 0
   commPaths[1]= malloc(sizeof(HWSimCommPath));
   setCommPathValuesTo(commPaths[1], 1,0,0,0);
   commPaths[1]->hasFixedTiming  = TRUE;
   commPaths[1]->fixedFlightTime = 10; //all time is stated in (integer) units
```

done building netlist, return it

```
   return netlist;
 }
```

The macro that sets the connections inside a comm path struct

```
#define setCommPathValuesTo( commPath, fromElIdx, outPort, toElIdx, inPort)\
do{\
   commPath->idxOfFromElem     = fromElIdx; \
   commPath->idxOfFromOutPort  = outPort; \
   commPath->idxOfToElem       = toElIdx; \
   commPath->idxOfToInPort     = inPort; \
 }while(0); //macro magic for namespace
```

Creating an element involves creating arrays for the in-ports and out-ports, then configuring the in-ports. The out-ports are automatically filled in during simulation start-up, by HWSim. The most interesting feature is that each in-port is assigned an activity type, which all arriving communications trigger. During the simulation, each incoming communication creates an activity instance, which points to this triggered activity type. The behavior and timing of the instance are calculated by the behavior and timing functions in the activity type. Notice that the activity type pointers are taken from the netlist, so they have to be created before creating the elements.

```
HWSimElem *
createAPingPongElem( HWSimNetlist *netlist )
 { HWSimElem *elem;
   elem = malloc( sizeof(HWSimElem) );
   elem->numInPorts  = 1;
```

```
   elem->numOutPorts = 1;
   elem->inPorts = HWSim_ext__make_inPortsArray( elem->numInPorts );
   elem->inPorts[-1].triggeredActivityType = IDLE_SPAN; //reset port
   elem->inPorts[0].triggeredActivityType  = netlist->activityTypes[PING_PONG_ACTIVITY];
        return elem;
 }
```

Creating an activity type involves setting the pointers to the behavior and timing functions, which are defined inside a separate directory where all the behavior and timing functions are defined. An activity may have behavior set to NULL, or timing set to NULL, and may have fixed timing. The structure has flags to state the combination.

```
HWSimActivityType *
createPingPongActivityType( )
 { HWSimActivityType *pingPongActivityType;
   pingPongActivityType = malloc( sizeof(HWSimActivityType) );

   pingPongActivityType->hasBehavior   = TRUE;
   pingPongActivityType->hasTiming     = TRUE;
   pingPongActivityType->timingIsFixed = TRUE;
   pingPongActivityType->fixedTime     = 10;
   pingPongActivityType->behaviorFn    = &pingPongElem_PingActivity_behavior;
   return pingPongActivityType;
 }
```

2.3 Activities, Behavior, Timing, and Communications

2.3.1 ActivityType and ActivityInst

Not all activities have both behavior and timing. This leads to four kinds of activity:

behavior + timing: the normal case, representing behavior in the hardware

behavior only: used for bookkeeping in the simulation, such as dispatching to other kinds of activity, as continuations of this one.

timing only: useful in network simulations, where there may be no behavior of an element, but timing is a complex function.

neither behavior nor timing: used when only want the reception of the communication to be noted by HWSim, and saved in the in-port. Will later be used by an activity triggered on a different port. For example, flip-flops where the input is ignored until the clock input triggers activity.

The kind of activity is set by the combination of flags inside the activity type structure. Some activities perform internal bookkeeping for the simulation, or dispatch different behaviors based on message contents. These have no timing. See the netlist creation comments above to see the actual flags and setting them.

2.3.2 Behavior Functions and Ending Activities

All behavior functions take a ptr to the activity instance they are executing the behavior of. The instance contains a pointer to the elem, and most behaviors will use the element's elemState field. It holds all the persistent state of the element, which remains between activities.

Here is the behavior function from the ping-pong example:

```
void
pingPongElem_PingActivity_behavior( HWSimActivityInst *activityInst )
 {    //NO_MSG is #define'd to NULL, and PORT0 to 0
   HWSim__send_comm_on_port_and_idle( NO_MSG, PORT0, activityInst );
 }
```

There are four ways a behavior can end:

end, no continuation:

```
HWSim__end_activity_then_idle( HWSimActivityInst *endingActivityInstance )
```

end, with continuation:

```
HWSim__end_activity_then_cont( HWSimActivityInst *endingActivityInstance,
                               HWSimActivityType *continuationActivityType)
```

end by sending a communication, with no continuation:

```
HWSim__send_comm_on_port_then_idle( void *msg, int32 outPort,
                               HWSimActivityInst *endingActivityInstance)
```

end by sending a communication, with continuation:

```
HWSim__send_comm_on_port_then_cont( void *msg, int32 outPort,
                               HWSimActivityInst *endingActivityInstance
                               HWSimActivityType *continuationActivityType)
```

Ending with a continuation means that a new activity instance is created. It's simulated-time starting point is the same as the end point of the instance being ended. The call passes a pointer to an activity type, which is the type of the new activity instance.

2.3.3 Activity Timing Functions

All activity timing functions take a ptr to the activity instance they are calculating the timing of. The instance contains a pointer to the element the activity is in. The behavior function is free to communicate to the timing function by leaving special data inside the element state. The timing function might also simply depend on the current state of the element.

Here's an example:

```
HWSimTimeSpan
sampleElem_sampleActivity_timing( HWSimActivityInst *activityInst )
 {
   return doSomethingWithStateOfElem( sendingActivity->elem->elemState );
 }
```

2.3.4 Calculating the time-in-flight of a communication path

The timing function for a communication path is similar to that of an activity. Except, the timing might also depend on configuration data or state stored inside the comm path struct, so that is passed to the timing function as well.

```
HWSimTimeSpan
commPath_TimeSpanCalc( HWSimCommPath *commPath, HWSimActivityInst *sendingActivity )
 { return doSomethingWithStateOfPathAndElem( commPath, sendingActivity->elem->elemState );
 }
```

Bibliography

[1] ...

[2] ...