HLA to Simulate Computer Systems at the Hardware Platform Level

Dr. Trevor Pearce
Amir Saghir
Dr. Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
Ottawa, ON. Canada
{pearce, saghir, gwainer}@sce.carleton.ca

Keywords: hardware simulation, DEVS, HLA

ABSTRACT*: The use of embedded computer systems to solve application problems requires an understanding of both the hardware platform involved and the software that customizes the hardware for the particular application. Modeling and simulation is used increasingly in developing the hardware and software of such systems. We have developed a generic framework to simulate the computer systems at the hardware platform level that can be used by the designers at different stages of the system development. For modeling platform components, the DEVS (Discrete event system specification) formalism is used. The simulation framework for these models is defined, following the specifications of the HLA (High level architecture) standard. We then describe the proposed framework of the simulator and how it interacts with general models of basic hardware platform components. We finally discuss a case study where this proposed simulation methodology is implemented on a specific hardware platform along with the results of the case study.*

1. Introduction

The use of embedded computer systems to solve application problems requires an understanding of both the hardware platform involved, and the software that customizes the hardware for the particular application. Modeling and simulation is used increasingly in developing the hardware and software of such systems. The models created by hardware engineers while developing hardware components often contain details that are irrelevant to software developers. Software developers would prefer a programmer's model of the hardware platform, which would abstract away irrelevant hardware details and focus only on the information relevant to software development. Ideally, the more abstract programmer's models would have larger grained simulations that are less computationally expensive than the hardware models.

This paper develops a framework to simulate computer systems at the hardware platform level, with the goal of supporting designers at different stages of system development. Within the framework, platform components are modeled using the Discrete EVent System specification (DEVS) formalism. The simulation framework for these models is defined, based on the specifications of the High Level Architecture (HLA) standard. Section 2 provides background information about the HLA and DEVS, including a layered approach to using DEVS and the HLA for a simulator. Section 3 describes the proposed simulation framework and how interacts with the general models of basic hardware platform components. Section 4 presents a case study where the proposed simulation methodology was applied to a specific hardware platform. Section 5 presents conclusions.

2. Background

This section introduces the HLA, DEVS, and some related work. A detailed discussion of DEVS is outside of the scope of this paper and can be found elsewhere [1]. This section briefly introduces those aspects of DEVS that are relevant to the component interactions, and describes other research that has mapped DEVS onto the HLA.

The HLA was originally developed by the US Department of Defense, and has evolved into IEEE standard 1516-2000. The architecture provides a component-oriented framework within which simulation developers can structure and describe their simulation application. In particular, the HLA addresses two key issues: promoting interoperability among simulations, and aiding the reuse of models. In the terminology of the HLA [2], the simulation system created by combining constituent simulations is a federation, and each constituent simulation is a federate.

The baseline of the HLA includes three specifications. The HLA Rules [3] define the responsibilities and relationships among the components of an HLA federation. The HLA Interface Specification [4] provides a specification of the functional interface between the HLA federates and the HLA RunTime Infrastructure (RTI). The RTI is a middleware that provides common services to federates, and enables a collection of federates to interact dynamically as a federation. This specification defines the RTI services API and identifies "callback" functions that must be provided by each federate. The HLA Object Model Template (OMT) [5] provides a common documentation standard for Simulation and Federation Object Models (SOM/FOM). A SOM defines the objects and interactions relevant to a single federate, while a FOM defines object models, communication between federates, and other information relevant to the interoperation of all federates in a federation.

The HLA Interface Specification divides the services provided by the RTI into six management areas, as summarized in table 1.

Figure 1 shows the major components of the RTI used in this research. The RTI library (libRTI) provides the RTI services specified in the HLA Interface Specification. The federation executive (FedExec) manages the federates within a single federation. It allows federates to join and to resign, and each federate joining the federation is assigned a federation wide unique handle. The handle facilitates data exchange between participating federates.
The RTI Executive (RtiExec) manages multiple federation executions in a network. It helps in initializing RTI components for each federation executive (FedExec), and also ensures that each FedExec has a unique name.

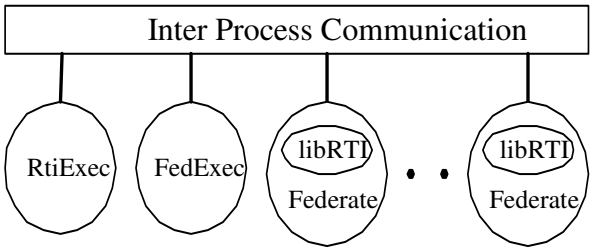

Figure 1: RTI Major Components [13].

DEVS is a system's formalism for describing hierarchical, modular models in a discrete event simulation. The DEVS formalism focuses on the changes of state variable values and generates time segments that are piecewise constant and continuous. All user-defined models are developed as either atomic or coupled models. An atomic model is a state machine, and the state of the model is changed by external and internal events that occur as time elapses. Input and output ports allow models to be interconnected to create new coupled models. When an output port on one model, say model A, is connected to the input port on another, say model B, then output events generated through the output port by model are communicated as external events to model B. Coupling creates a hierarchical tree, with atomic models at the leaves.

Various software tools have been developed to implement the DEVS formalism, such as DEVS-C++ [6], CD++ [7] and DEVSim++ [8]. These tools map DEVS model descriptions to a simulation engine, and provide libraries and methods that enable the DEVS models to interact as a hierarchical simulation based on the coupling hierarchy.

Zeigler et al [9] designed and developed an HLA compliant simulation environment called DEVS/HLA. In their approach, DEVS models are mapped onto the RTI using the layered approach shown in figure 2. The

Management Area	**Activities Supported**
Federation Management	Manages federation execution. Initializes name space, transportation, routing spaces etc.
Declaration Management	Specifies the data a federate sends and/or receives.
Object Management	Creates, modifies and deletes objects and interactions. Facilitates object registration and distribution. Coordinates attribute updates among federates. Accommodates various transportation and time management schemes.
Ownership Management	Supports transfer of ownership for individual object attributes. Offers both "push" and "pull" based transactions.
Time Management	Establishes or associates events with federate time. Regulates interactions, attribute updates, object reflection or object deletion by federate time scheme. Supports interaction between federates having different time schemes.
Data Distribution Management	Supports efficient routing of data.

Table 1: RTI Management Areas partitioned in FedExec life cycle [13].

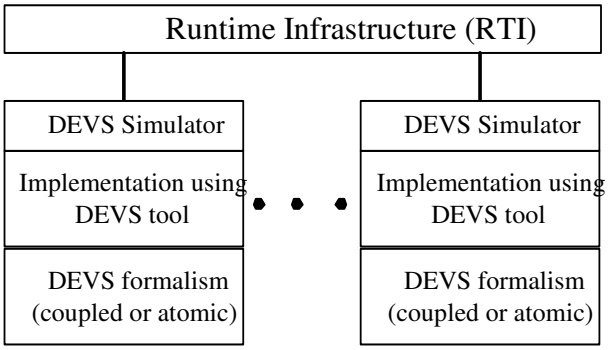

Figure 2: Layered approach to implement DEVS/HLA. [9]

mapping exploits the information contained within the DEVS models to automate as much as possible of the programming work required in constructing the HLA compliant simulations. The goal is to facilitate a bi-directional transfer of information between the OMT Development Tool (OMDT) that captures OMT information and the DEVS model description.

The DEVS formalism layer of figure 2 is used for defining the models. The DEVS-C++ tool includes a library to interface DEVS and the RTI in a C++ environment. This library contains methods for attribute updates, attribute reflections, interaction updates, interaction receive, object discovery and quantizers. A quantizer object is associated with each attribute that the modeler would like to publish, and the quantizer checks for the attribute value crossing programmed thresholds. Update messages are only sent when a value crosses a threshold. Quantizers reduce message update traffic, and the size of the quantizers is directly related to the accuracy and the speed of the computation required.

The simulator/implementation layer of figure 2 is a simulation engine, which takes care of all time and data interactions among the DEVS models. It also acts as a message translator between the DEVS models and the RTI.

The research reported below shares some similarities with previous work by Fay and Holoway [10]. They model a hardware platform, and simulate the execution of programs using the self-contained federation paradigm. In this paradigm, the federation executes as a single process on a single machine. They obtain significant performance gains using the alternate paradigm, since they eliminate the need for communication overhead and simplify the runtime coupling of models. Significant differences in the research reported here include the use of DEVS to model components, and the use of a standard HLA federation model.

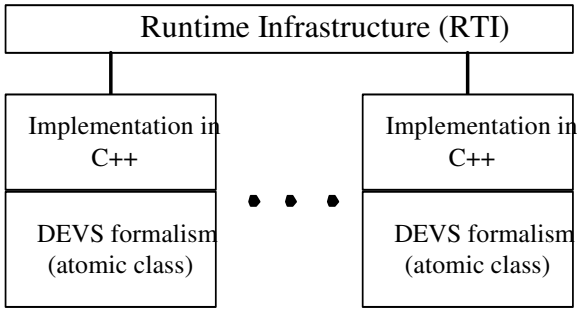

Figure 3: A model of the proposed solution.

3. Simulation Framework

This section presents a simplified system level framework for the hardware platform simulator using DEVS and HLA/RTI. The discussion starts with a system level block diagram, and includes a generic simulation flow for a federate. The description of the DEVS atomic models used for each hardware component is outside of the scope of this paper and can be found elsewhere [11].

A key simplification that simplifies the proposed framework is that only DEVS atomic models are used. The rationale for this simplification include controlling atomic models directly from the RTI, therefore an extra layer of DEVS middleware to support coupled models is not required. This approach effectively removes model coupling and flattens the DEVS modeling hierarchy. The resulting simplified structure is shown in figure 3, where the DEVS simulator/tool present in figure 2 is no longer required.

Figure 4 shows a system level block diagram for a simple hardware platform simulator. It contains the following basic hardware components: Processor module, Bus controller module, Memory module, Interrupt controller module and Timer module. The Processor module is further divided into three units: Execution Unit (EU) that is responsible for decoding and executing all instructions, Bus Interface Unit (BIU) that is responsible for performing all external bus operations and Control Unit (CU) that is responsible for minimum/maximum mode and interrupt signals from other devices. This version of the framework has a single bus master (the processor); and therefore a bus arbiter module is not included. Each of these components is modeled as a DEVS atomic model, and is realized as a unique federate. The Runtime Infrastructure (RTI) provides the simulation platform for this federation.

To create a specific hardware platform model, the DEVS atomic models of each component must be constructed. The atomic models must include input and

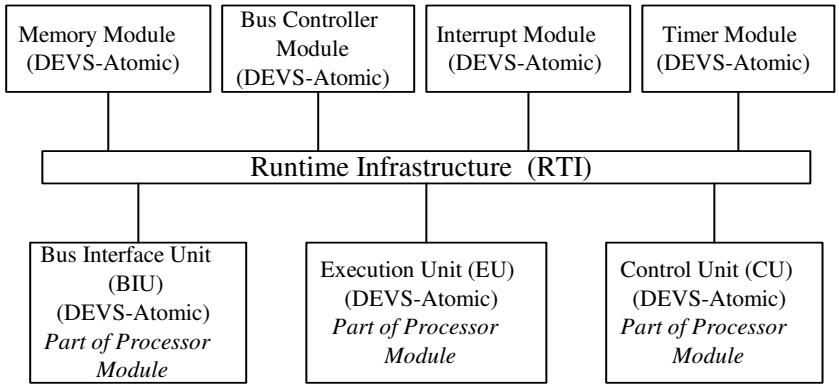

Figure 4: System-level block diagram for Hardware platform simulator.

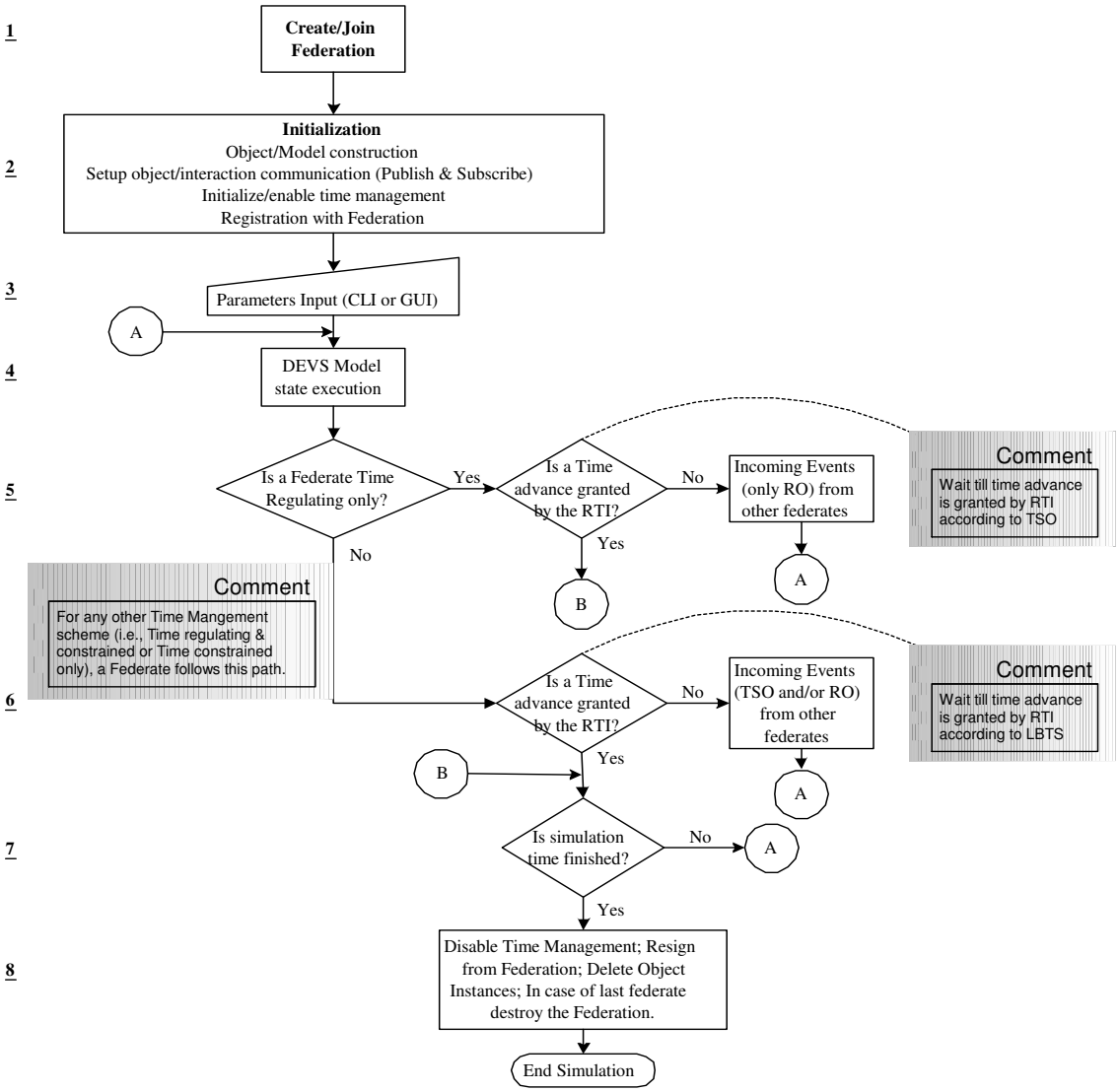

Figure 5: Simulation flow chart of a federate.

output ports to accommodate interactions among the components. Ideally, the component models would accumulate in a repository for reuse in different platform models. Once the component models are constructed, they must be coupled (logically) to indicate the information paths in the system and the types of information flowing on the paths. The coupling is logical (as opposed to creating a coupled model in the DEVS sense) in that the coupling information is used to identify data attributes that will be shared via the RTI, and also to specify the federates that will publish and subscribe to the attributes. A simple mapping from DEVS external events to RTI update calls is used to realize communication between components.
The time advance requests issued in a platform simulation depend on the level of abstraction of the component models. If the models are accurate to bus cycles, the input/output ports would correspond to bus-level interconnections, and information exchanges would follow the bus protocols for the components. At this level, time advances would typically be the same magnitude as the bus clock cycle. If more abstract models are implemented, then the inputs/outputs and interaction protocols can also be more abstract. For example, information transfers involving multiple bus lines might be modeled using a single port, information paths that might be multiplexed across the real bus might me modeled as separate paths, bus signals that are not relevant at the level of abstraction could be ignored, and the number of steps in bus protocols could be reduced. By reducing the number of steps in abstract interaction protocols, the resulting simulation will not experience as many time advance requests, and the magnitudes of requested time advances are likely to be integer multiples of the bus clock cycle. These more abstract models would not have bus cycle accuracy; however, they would result in simulations with faster real-time performance.

Figure 5 shows the generic simulation flow of a federate. The left side of the figure is labeled with reference numbers corresponding to steps in the flow chart. The reference numbers simplify the following discussion of the flow.

Step 1 is the initial step of the simulation in which a federate creates or joins a federation. If the federation does not exist, a new one is first created and then the federate is added to it. If a federation already exists, then the federate joins that federation.

At step 2, the federate is initialized. Initialization consists of the Object/Model construction, defining the information to be published or/and subscribed (can be re-defined during simulation), defining the federate as time regulating and/or time constrained (can be re-defined during simulation) and the registration of the federate with the federation executive. The FOM must also contain these published and subscribed attributes and parameters. The registration process registers the federate with the federation execution and returns an HLA object handle. The handle identifies the federate during subsequent interactions within the federation.

Step 3 allows a user to input parameters for a federate before starting the simulation loop. This step is optional, as some federates do not need any user input.

Step 4 is the first step in the simulation loop. In this step, a federate executes state transition functions according to its DEVS atomic model.

Following step 4, a federate will perform either step 5 or 6, depending upon its time management scheme. Step 5 is performed if a federate is time regulating only (i.e. only capable of sending Time Stamped Order (TSO) messages). The RTI grants a time advance according to the TSO message time. On the other hand if there is some incoming Receive Order (RO) message (i.e. a message with no time stamps) from another federate, then the RTI delivers this message to the federate without granting a time advance. Whether a time advance is granted or not, the simulation proceeds and the DEVS model transitions from one state to another. Step 6 is performed if a federate is time constrained (i.e. capable of receiving TSO messages). The RTI grants a time advance according to the Lower Bound Time Stamp (LBTS). The LBTS is the maximum time to which a time constrained federate may advance. The LBTS is determined by the RTI based on the lookahead values (a time period during which a federate does not send out any message) for each federate and TSO messages from all the time regulating federates. On the other hand if there is some incoming message (either TSO or RO) from another federate then the RTI delivers this message to the federate without granting a time advance. If a time advance is granted, the federate proceeds to step 7. If a time advance is not granted, the federate returns to the top of the simulation loop for another DEVS model state transition.

At step 7, the federate checks whether the simulation loop will continue or stop. At step 8, a federate removes itself from the federation by disabling time management, resigning from the federation, deleting object instances, and, in the case where it is the last federate, destroys the federation.

4. Case Study

The proposed simulation framework, as discussed in section 4, for hardware platform modeling using the DEVS/HLA approach is verified by the case study. A hardware platform containing a processor (a simplified version of Intel 8088), a basic memory unit, a bus controller (a simplified version of Intel 8288), a basic interrupt controller and a basic timer unit is modeled

using DEVS. This case study is for a synchronous platform, but no clock module is modeled. Each module assumes synchronous interactions and requests to schedule the future bus events as an integral number of clock ticks in the future.

For this case study, a simple simulator is developed to run the instruction codes shown in tables 2 and 3. The simulator generates 10 even numbers, from 0 to 18, and stores them in the memory module. The instructions, their encoding, and clock timing information [12] are shown in table 2. Instruction execution times are determined by taking the number of clocks cycles required per instruction plus any effective address (EA) time required for the operand. The EA for the indexed operand in the MOV [BX],CH instruction is 5 clocks.

During the execution of the main program, the interrupt controller module sends interrupt signals to the processor, resulting in the execution of the interrupt routine (shown in table 3). The processor's interrupt behavior includes checking the interrupt flag; pushing the IP, CS and IF; sending the interrupt acknowledge to the interrupt controller; reading the vector type from the interrupt controller; calculating the starting address for the interrupt routine residing in the memory and clearing the instruction queue. The interrupt routine used in the case study increments a variable stored in the memory. The instruction details for the interrupt routine are shown in table 3 [12].

For this case study one simulation time unit represents one bus clock cycle, and one bus cycle is assumed to

Opcode (Hex)	Program Instructions	Clock Cycle
B500	MOV CH , 0	4
BF0000	MOV DI , 0	4
B102	MOV CL , 2	4
BB0000	xyz : MOV BX , 00	4
03DF	ADD BX , DI	3
882F	MOV [BX] , CH	9+EA = 9+5 = 14
47	INC DI	2
02E9	ADD CH , CL	3
83C70A	CMP 10 , DI	4
750D	JNE xyz	4 (when not executed) 16 (when jump executed)
F4	HLT	2

Table 2: Main program for the case study

Op-code (Hex)	Program Instructions	Clocks
53	PUSH BX	15
BB0020	MOV BX , 32	4
8A07	MOV AL , [BX]	8+EA = 8+5 = 13
FEC0	INC AL	3
8807	MOV [BX] , AL	9+EA = 9+5 = 14
5B	POP BX	12
CF	IRET	32

Table 3: Interrupt routine's opcodes for the case study

```
_____________________________________________________________________
// Setting up the data structure required to send this object's state to the RTI.
RTI::AttributeHandleValuePairSet* Memory::CreateNVPSet()
{
   RTI::AttributeHandleValuePairSet* pMemoryAttributes = NULL;

   //-------------------------------------------------------
   // Set up the data structure required to send this object's state to the RTI.
   //-------------------------------------------------------
   pMemoryAttributes = RTI::AttributeSetFactory::create( 1 );

   pMemoryAttributes -> add( this -> GetDataFromMemRtiId(),
                             (char*) &this -> GetDataFromMem(), (sizeof(int)) );
   return pMemoryAttributes;
}
_____________________________________________________________________
```

Figure 6: Creating an AHVPS for communication among Module/Federates.

```
//-------------------Setting the lookahead time
  lookahead = RTIfedTime(0.5);
//-------------------
  RTIfedTime requestTime(1.0); // requestTime = timeStep = 1.0
  requestTime += grantTime; // grantTime is the time granted by the RTI during the last
time elapse
  // log the request
  out2file << "\n\nRequest time = " << requestTime << endl;
  timeAdvGrant = RTI::RTI_FALSE;
  rtiAmb.nextEventRequest( requestTime );
  while( timeAdvGrant != RTI::RTI_TRUE )
    {     //-------------------------------------------------------
   // Tick will turn control over to the RTI so that it can process an event.
   //-------------------------------------------------------
          rtiAmb.tick();
    }
```

Figure 7: Time elapse requested by a Federate using the RTI service methods.

consist of four bus clock cycles. The models are abstract, and within a bus cycle the timing details are not implemented accurately. For the execution unit of the processor the timing details are implemented with one clock cycle accuracy.

The FOM for the case study identifies the time management parameters and the attributes used by each federate. The attributes correspond to information exchanged through the input/output ports of the DEVS models of the components. For example all the attributes which the processor component sends to the memory component should be defined as 'published' by the processor federate and 'subscribed' by the memory federate.

Attributes are sent from one federate to the other in the form of AttributeHandleValuePairSet (AHVPS). AHVPS is a set comprised of attribute handles, values and the size of the values. To create the AHVPS, a method CreateNVPSet(), extracted from the memory module's software code, is shown in figure 6. The RTI method "AttributeSetFactory::create(1)" is used to create the AHVPS for only one AHVP, as there is only one attribute to send out i.e., "DataFromMem".

Figure 7 shows a time elapse request being sent to the to the RTI according to the time calculated (represented by the "requestTime") for the next event. The lookahead value (see step 6 figure 5) is set to 0.5, meaning that the federate will not send any output until this period is elapsed. The time step value is set to 1.0, indicating that this is the maximum limit for the time elapse requested to the RTI. The final "requestTime" is calculated by adding the time step value of 1.0 and the "grantTime", which is the time granted in the previous time elapse request. In other words the "grantTime" is the current simulation time. The RTI method "nextEventRequest(requestTime)" requests for the time elapse until the "requestTime". In return, the RTI grants the time advance until the next event. The tick( ) function gives control to the RTI for processing the ongoing events/tasks.

A logging capability (out2file in figure 7) is developed for all component models, and each federate has its own log file. Whenever a federate receives or updates an attribute it sends this information to its log file along with the with the simulator's time stamp. The status of the internal registers for the processor federate and memory contents for the memory federate are also logged.

The case study simulator was verified by analyzing the logs generated by the components during a simulation run and comparing these results with the theoretical values provided by tables 2 and 3. The processor is accurate to one clock cycle, and hence the simulation results for the execution timing for each program instruction should match the theoretical timing values.

The following example analyzes the logs to verify the execution of MOV [BX], CH. Decimal values are used throughout the analysis. Table 2 shows that the instruction requires 14 clock cycles for the EU and the BIU to execute this instruction. Figure 8 shows a snap shot of the processor module's log, when MOV [BX],CH is at the top of the Instruction queue (line 4). Executing the instruction should move the value in the CH register to the memory location at the address calculated by ((DS×16)+BX). Lines 2 and 3 show the values of the BX, CH and DS registers. Hence the execution should result in the value 2 being moved to the memory location at address 4×16+1 = 65. Line 1 shows that the execution start time is 138.0.

Figure 9 shows a snap shot of the memory module's log corresponding to the completion of the instruction.

```
--------------------------Processor log-------------------------------
Line 1: FED_HP: Time granted (timeAdvanceGrant) to: 138.0000000000
Line 2: Zero Flag=0 INT Flag=1 AL=0 DI=1 BX=1 CL=2 CH=2
Line 3: IP=14 CS=1 SS=7 SP=31 DS=4
Line 4: Contents of the queue: 136 47
----------------------------------------------------------------------
```

Figure 8: Snap shot of the Processor's log for the analysis of the MOV [BX],CH instruction.

```
-------------------------------Memory log-------------------------------
Line 1: Control Signal From Pro To Mem = 2
Line 2: Address from Processor = 65
Line 3: Data from Processor = 2
Line 4: FED_HP: Time granted (timeAdvanceGrant) to: 152.0000000000
Line 5: 30 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
Line 6: 181 0 191 0 0 177 2 187 0 0 3 223 136 47 71 2
Line 7: 233 131 199 10 117 13 244 0 0 0 0 0 0 0 83 187
Line 8: 0 32 138 7 254 192 136 7 91 207 0 0 0 0 0 0
Line 9: 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
----------------------------------------------------------------------
```

Figure 9: Snap shot of the Memory's log for the analysis of the MOV [BX],CH instruction.

Lines 1 to 3 indicate that the Memory is commanded to write (control signal = 2) the value 2 at address 65. Line 4 shows that the command was processed at time 152.0, which represents the end of the execution of the MOV [BX],CH instruction. Lines 5 through 9 show a memory dump after performing the command. Each line shows the values of 16 bytes of memory, starting sequentially at address 0 in line 5. The content of memory at address 65 (represented by the second byte of line 9) is shown to be 2, indicating that the write was performed. A more complete analysis would also compare a complete memory dump before the operation to a complete memory dump after the operation to ensure that no other memory contents had changed.

The difference between the execution start time (138.0 from line 1 in figure 8) and end time (152.0 from line 4 of figure 9) is 14 clock cycles. This result matches the theoretical clock cycle value (line 6 of table 2) required for the execution of the MOV [BX],CH instruction.

5. Conclusions

The HLA has been used to simulate a computer system at the hardware platform level. Hardware components are modeled as DEVS atomic models, and the HLA/RTI is used as a simulation platform in place of the simulation engines inherent to traditional DEVS tools. The approach has been verified in a case study to show that results can be accurate to within a clock cycle. The HLA-specific details that should be visible in component models include the attributes for DEVS input and output ports that should be published/subscribed to the RTI and defined in the HLA compliant FOM.

6. References

[1] Zeigler, B. P., Praehofer, H., Kim, T. G.: Theory of Modeling and Simulation, Academic Press, 2000

[2] Dahmann, J.S.: "High Level Architecture for Simulation" First International Workshop on Distributed Interactive Simulation and Real Time Applications, 1997. Page(s): 9 –14

[3] IEEE standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Framework and Rules. IEEE Std. 1516-2000, 2000

[4] IEEE standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Federate Interface Specification IEEE Std 1516.1-2000, 2001

[5] IEEE standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Object Model Template (OMT) Specification IEEE Std 1516.2-2000 , 2001

[6] Zeigler, B., Cho, H.; Lee, J.; Sarjoughian, H.: "The DEVS/HLA Distributed Simulation Environment And Its Support for Predictive Filtering", DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.

[7] Rodriguez, D., Wainer, G.: "New Extensions to the CD++ tool". Proceedings of SCS Summer Multiconference on Computer Simulation, 1999.

[8] Kim, T. G.: "DEVS Research at KAIST CORE LAB: From Theory to Practice". Computer Engineering (CORE) Lab. Department of Electrical Engineering, KAIST, Korea. July 3, 1995. Online report URL: http://sim.kaist.ac.kr/~tkim/devsim.html

[9] Zeigler, B.P., et al. “Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions”, Proceeding of the 1999 Spring Simulation Interoperability Workshop, March 1999. Orlando,FL

[10] Fay, J.F, Holoway, T.R.: “Electronic Circuit Simulation Using The High-Level Architecture”, Proceedings of the 2002 Fall Simulation Interoperability Workshop, September, 2002, Orlando, FL

[11] Saghir, A., Pearce, T.W., Wainer, G., “Modelling Computer Hardware Platforms using DEVS and HLA Simulations”, 2004 Summer Simulation Conference, July 25 – 29, 2004, San Jose, California

[12] The Intel iAPX 88 Book. Intel Corporation, July 1981.

[13] High-Level Architecture / Runtime Infrastructure. RTI 1.3-Next Generation, Programmer’s Guide Version 3.2; 7 September 2000. www.dmso.mil