Modeling Computer Hardware Platforms using DEVS and HLA Simulation

Amir Saghir
Trevor Pearce
Gabriel Wainer
Department of Systems and Computer Engineering
Faculty of Engineering,
1125 Colonel By Drive, Carleton University
Ottawa, Ontario, K1S 5B6, Canada.

Key words: Discrete event simulation, DEVS, High Level Architecture, HLA, Runtime infrastructure, RTI.

Abstract

We introduce the use of DEVS and the HLA for the development of embedded computer systems. Modeling and simulation can play an important role in the development of such systems by allowing earlier feedback on component integration. Our objective is to describe how general hardware components can be modelled and integrated at a level of abstraction that can be used by systems and software developers, and how these general models can be programmed for a specific hardware platform. The differing focus of hardware and software development tools often results in a significant gap in the levels of abstraction of the resulting component models. In the proposed approach, a set of general hardware components is designed using DEVS. The general components are then programmed for specific hardware components and used to simulate a specific hardware platform. The hardware models interact with each other in a simulation environment based on the HLA (High Level Architecture). A unique feature of the resulting simulation is that coupled DEVS models are not used, and as a result, the DEVS simulation middleware can be simplified by direct use of HLA services. The proposed approach is verified by a case study in which an Intel 8088-based computer system is modelled and simulated.

1. INTRODUCTION

The development of embedded computer systems is a complex process. Application requirements must be implemented in a combination of software and hardware components, and delays in integrating the components can lengthen the development lifecycle significantly. Modeling and simulation can play an important role in the development of such systems by allowing earlier feedback on component integration.

When developing hardware/software platforms, it is useful to model components at various levels of abstraction and complexity. Hardware developers require models that interact using digital signals and signalling protocols. Internally, the hardware components are often modelled at the register transfer level, and in terms of individual electronic devices and manufacturing technologies. In contrast, software development environments attempt to minimize the visibility of hardware details in favour of emphasizing application-level software solutions. The differing focus of hardware and software development tools often results in a significant gap in the levels of abstraction of the resulting component models. The gap can hamper simulation when integrating components, particularly during early development stages when feedback would be most beneficial.

The hardware/software interface is where a computer system implementation realizes the integration of hardware and software components. From a software perspective, hardware models often over-specify the interface by including signalling details associated with hardware component interactions. It may be possible to simulate software executing at this level; however, the resulting simulation is often too large and cumbersome to produce detailed results in reasonable timeframes. Furthermore, locating software flaws can be next to impossible due to the overbearing amount of hardware detail.

Software developers would prefer a programmer's model of the hardware/software interface. This model is more abstract, and views hardware at a more coarse level. Hardware details are often reduced by assuming that bus interactions are atomic, and therefore, the behaviours of hardware components can be described with less concern for signalling and circuitry implementation. Simulations at the programmer's model level would be larger grained than traditional hardware simulations, and therefore less computationally expensive and less cluttered by hardware details.

Our objective is to describe how general hardware components can be modelled and integrated at a level of abstraction that can be used by systems and software developers, and how these general models can be programmed for a specific hardware platform. A set of general hardware components is designed using DEVS (Discrete Event Sys-

tems specifications) [1]. The general components are programmed for specific hardware components and used to simulate a specific hardware platform. The hardware models interact with each other in a simulation environment based on the HLA (High Level Architecture) [7]. This proposed approach is verified by a case study in which an Intel 8088-based computer system is modelled and simulated.

This next section provides a brief introduction to DEVS and the HLA. Section 3 describes how we model general hardware components using DEVS, and how these models interact with using RTI services. The case study in Section 4 shows the refinement of the general models for a simple Intel 8088-based platform. The simulation results of the case study are then used to verify the proposed approach.

2. DEVS and the HLA

DEVS is a well-defined modeling and simulation framework that can express hierarchical, modular, discrete event models. A DEVS model evolves by changes in state variables through the occurrence of events and elapsed time. The DEVS formalism focuses on the changes of the variable values, and generates time segments that are piecewise constant. An important aspect of the formalism is that the time intervals are continuous. There are two types of models: atomic (behavioural) and coupled (structural) [1]. Since coupled models are not used in the research, they are not discussed in any detail here.

A DEVS atomic model (AM) specifies a system's response to input events in terms of states, transitions, timing constraints, and output events. An atomic model is defined formally as:

$$AM = < X , S , Y , \delta_{int} , \delta_{ext} , \delta_{con} , \lambda , ta >$$

Where

X is the input events set.
S is the state set.
Y is the output events set.
$\boldsymbol{\delta_{int}}$ is the internal transition function.
$\boldsymbol{\delta_{ext}}$ is the external transition function.
$\boldsymbol{\delta_{con}}$ is the confluent transition function
λ is the output function.
ta is the time advance function.

The formal definition can be interpreted using the following operational rules. When an input event arrives, the δ_{ext} function accepts the input and changes the instance variable. If no further inputs arrive during a time interval specified by the ta function, then the δ_{int} function changes the instance variable. Should an input arrive before the time interval elapses, then the δ_{ext} function accepts the input and changes the instance variable. The δ_{con} function resolves situations where events occur simultaneously. δ_{con} might specify, for example, the order in which the relevant functions should be performed. The λ function produces the output Y from the instance variables.

Various software tools have been developed to implement the DEVS formalism, including DEVS-C++ [4], CD++ [5] and DEVSim++ [6]. These tools provide methods to implement DEVS models and map them to a simulation engine so that the DEVS atomic and coupled models can interact with each other in a hierarchical simulation environment.

The HLA is an IEEE standard [7] for the interoperability of component-based simulations. The HLA promotes simulation interoperability, and encourages the reuse of models. The HLA provides a general framework, which defines the relationships among the components (called *federates*) of an HLA simulation (called a *federation*). The HLA Interface Specification defines the functional interface to the HLA RunTime Infrastructure (RTI) [8]. The RTI is a middleware layer that provides HLA services to federates and federations.

Previous efforts have been made to create HLA-compliant DEVS models, and to integrate DEVS simulators with the RTI [2]. These efforts allow re-use of the models and simulators, but require the DEVS simulation engine as an additional middleware layer. We propose a new distributed approach where all the hardware components are modelled as DEVS atomic models, and interact with each other using a HLA-compliant simulation engine. Although DEVS has already been used successfully for the modeling and simulation of hardware architectures [3], we wanted to experiment with a more direct approach to the interaction of DEVS models in an HLA simulation.

Figure 1.a) shows a general approach to modeling and simulation using DEVS and the HLA/RTI. Two layers of simulation middleware (e.g. the RTI and the DEVS simulator) are on top of each other. The DEVS simulation layer is

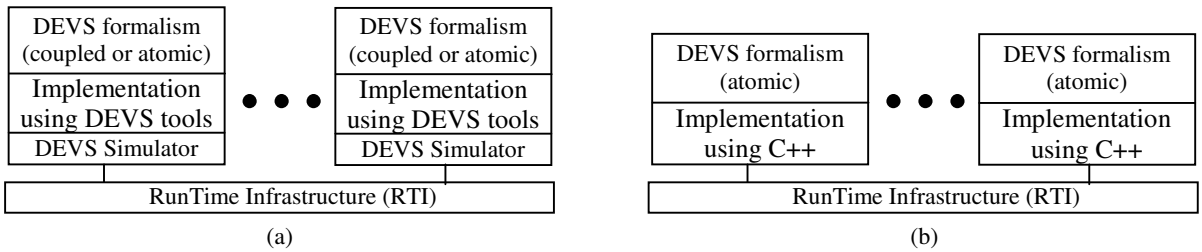

Figure 1: (a) DEVS/HLA general approach. (b) A model of the proposed solution.

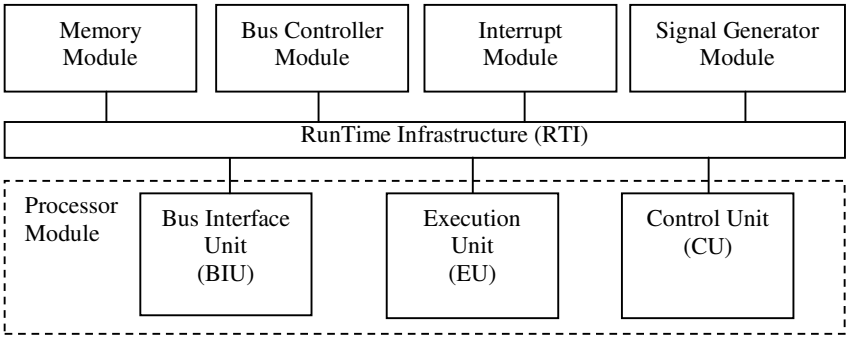

Figure 2: System-level block diagram for a Hardware platform simulator.

required to support the execution of DEVS coupled models. DEVS tools include complex simulation and coordination engines in order to manage/synchronize the different components. Such engines require additional message translation functionality to work with the RTI layer. Zeigler [2] has shown how to adapt the DEVS-C++ tool to the HLA/RTI structure. This is quite useful to reuse old simulations developed under the DEVS-C++ tool but suffers from the extra layer of DEVS middleware.

Figure 1(b) shows our proposed approach to model general hardware components using DEVS, and then simulate compositions of components above the RTI. We use only DEVS atomic models, and as a result, RTI services can be used to control the models directly. By using only DEVS atomic models, the extra layer of DEVS middleware to support coupled models is not required. Atomic models are a simple concept, and easily reusable for future developments.

3. HARDWARE PLATFORM MODELS

Figure 2 shows a system level block diagram for a simple hardware platform simulator consisting of the Processor, Bus controller, Memory, Interrupt Controller and Signal Generator modules. The Processor module has been shown divided into the: Execution Unit (EU) that is responsible for decoding and executing all instructions, the Bus Interface Unit (BIU) that is responsible for performing all external bus operations, and the Control Unit (CU) that is responsible for possible processor configuration modes (e.g. the maximum/minimum modes of the Intel 8088) and interrupt signals from other devices. Only a single master device is present in the system, and therefore a bus arbiter component has not been included. The framework assumes atomic bus cycles, hence none of the detailed signalling protocols within a bus cycle are addressed in the component models. Each of the modules is specified as a DEVS atomic model, and is realized as an HLA federate. The federates collectively form a federation interacting on the RTI.

As a representative organization of the modules, the DEVS atomic model of the memory module is given below (models of the remaining components can be found in [9]), and shown as a state machine in Figure 3.

$$\text{Memory} = < X , Y , S , \delta_{int} , \delta_{ext} , \delta_{con} , \lambda , ta >$$

X – Input events

- Address bus $\in \{0 \ldots 2^{AB\_Size}-1\}$, where AB_Size is a natural number defined in the processor module's parameters. To support access to all memory locations, Memory size (MS) $\leq 2^{AB\_Size}$.
- Data bus $\in \{0 \ldots 2^{DB\_Size}-1\}$, where DB_Size is a natural number defined in the processor module's parameters.
- Read control signal $\in \{0,1\}$ where 1 = Read and 0 = no operation specified.
- Write control signal $\in \{0,1\}$ where 1 = Write and 0 = no operation specified.

Y – Output event set

- Data bus $\in \{0 \ldots 2^{DB\_Size}-1\}$, where DB_Size is a natural number defined in the processor module's parameters.
- Data acknowledge signal (DTACK): informs the processor that the bus cycle has ended during an asynchronous processor's mode (e.g., Motorola 68000 series). DTACK $\in \{0,1\}$ where 1 = ACK and 0 = no operation.

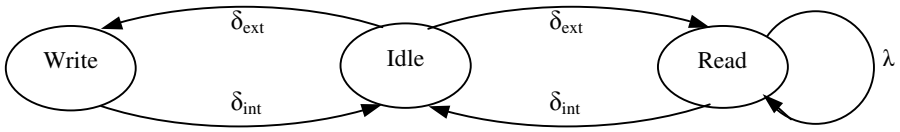

Figure 3: State diagram for the DEVS model of the Memory Module

S – States

- Read state: In read state the memory module acquires data from the memory location as addressed by the address bus {0… 2^{AB_Size}-1}, waits until the ta function is elapsed and writes data to the data bus {0… 2^{DB_Size}-1}.
- Write state: In write state the memory module writes data, provided by the data bus {0… 2^{DB_Size}-1}, to a memory location which is addressed by the address bus {0… 2^{AB_Size}-1}. The module then waits until the ta function is elapsed.
- Idle state: In this state the module waits for the next δ_{ext}. The delay function associated with this state $\in R^+$.

δ_{ext} – External transition function

The δ_{ext} starts with the arrival of read/write signal in the module. In case of write signal, the δ_{ext} acquires the information from the address and data bus. In the case of a read signal, the δ_{ext} only acquires the information from the address bus. As an example the state diagram for memory read and memory write during a minimum operation mode is shown in figure 3.

ta – Time advance function

The ta function introduces the time delay before scheduling the next action by the component. At this level of abstraction, the ta functions associated with read and write states are equivalent to one bus cycle. Figure 3 shows an example where the ta function is introduced in the read state before carrying out the λ function.

λ – Output function

The λ outputs the data on the data bus during the read cycle. For the case of Motorola 68000 series processors the DTACK signal is also sent out. As an example, see figure 3.

δ_{int} – Internal transition function

The δ_{int} changes the internal state from Read or Write to the Idle state. The module stays in the Idle state until the next δ_{ext}.

δ_{con} – Confluent transition function

The δ_{int} has higher priority than the δ_{ext}

Figure 3 represents the DEVS model for the memory module behaviour graphically. The module stays in the Idle state until there is an input event. The input event either causes a transition to the Read state or the Write state. In the Read state, the memory module gets data from the requested memory location, waits for a time representing the memory access time to elapse, and then performs output (λ) by sending data to the data bus. After sending the output, the module uses an internal transition (δ_{int}) to return to the Idle state. In the Write state, the module reads the address and data buses, updates the specified memory locations, waits for a time representing the memory access time to elapse, and then returns to the Idle state.

4. 8088-BASED CASE STUDY

The general hardware component models were refined to simulate a 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 timing signal generator module. This case study is for a synchronous platform, but no clock module is modeled. Each module assumes synchronous interactions and requests to schedule future bus events as an integral number of clock ticks in the future. The implementation assumes that the memory module does not introduce wait states.

The simulator was developed in sufficient detail to execute the assembly code shown in tables 1 and 2. The assembled code is shown in the first column of table 1 along with the mnemonic code and any necessary timing information [10]. Instruction execution times are determined by taking the number of clock cycles required per instruction plus any time required to access the operand based on the effective address (EA) used in addressing modes. For instance, 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,

Op-code (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 16 (when jump executed)
F4	HLT	2

Table 1: Main program code 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 2: Interrupt routine code for the case study

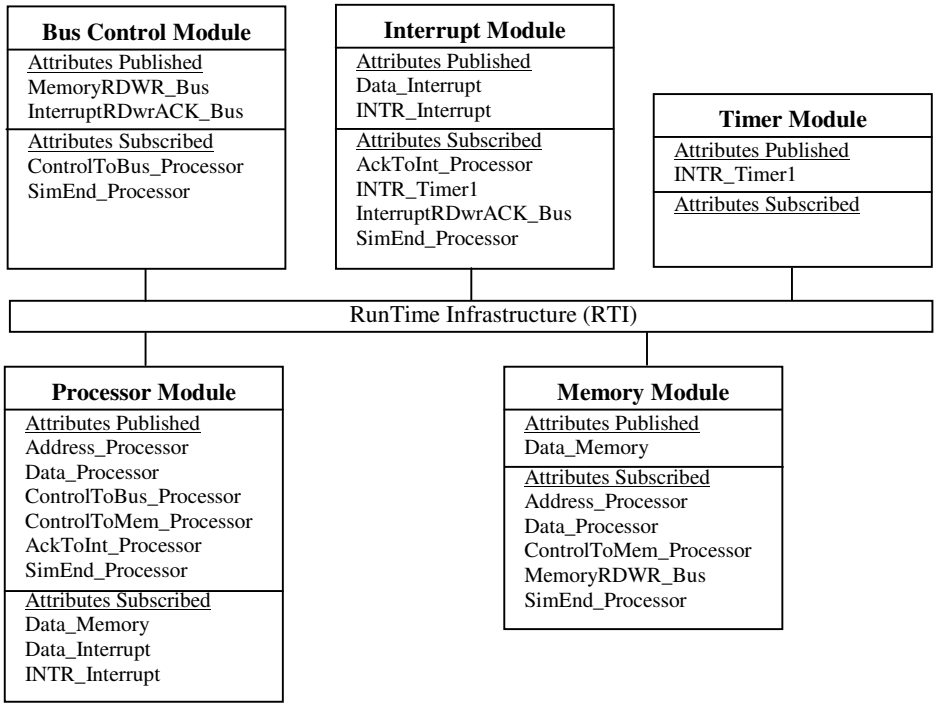

Figure 4: Block diagram showing time publish/subscribe attributes

resulting in the execution of the interrupt routine shown in table 2. The processor's interrupt behaviour 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 timing information for the interrupt routine is also shown in Table 2 [10]. A time unit of the simulator is assumed to be one (system) clock cycle, and one processor bus cycle is assumed to be four clock cycles. The timing details of the execution unit of the processor are implemented to one clock cycle accuracy. Within a processor bus cycle, the system timing details are not implemented accurately. This lack of accuracy is irrelevant, since the processor control unit dictates the overall timing behaviour of the system.

Components of the simulator are defined as HLA federates, and figure 4 shows the attributes (information) communicated through input and output ports of the DEVS models of the components. For instance, the memory module publishes data values (the *Data_Memory* attribute) as outputs, to which the processor module subscribes as inputs.

The federates exchange attributes using the RTI publish and subscribe services. When a federate publishes attribute data, all federates subscribed to that attribute are notified. This allows the coupling of the DEVS atomic models to be realized using RTI services.

The memory module implementation is based on the DEVS specification presented in Section 3. RTI services are used to communicate events between atomic models and to manage time advances. Therefore, the implementation does not require a DEVS simulation engine for this support.

The memory module is passive, and responds to events from the RTI. In the Idle state, the module has no pending work to perform. The memory module's δ_{ext} function is activated by the arrival of subscribed data. The function is implemented as a case statement that performs the appropriate processing based on the attribute received. For example, suppose that the memory module is Idle when the processor initiates a memory read operation. The processor does this by publishing a *ControlToMem_Processor* attribute, which is one of the attributes subscribed by the memory module. The attribute includes information about whether the access is a read or write operation, and the relevant memory address. The memory model must not return the requested data immediately, but must follow the bus protocol. The module enters the Read state, and waits for one memory read bus cycle (4 clock ticks) to lapse. Upon entering the Read state, ta function is activated to request a time advance of 4 clock ticks using the RTI time management services. The RTI controls time advance among the federates, and when appropriate, sends a *TimeAdvanceGrant* to the memory module indicating that it has waited the requested amount of time. The memory module responds by activating the λ

function to output data back to the processor (achieved by publishing a *Data_Memory* attribute), and then the internal transition function (δ_{int}) is activated to go back to the Idle state.

The simulator is instrumented to log events, and the case study was verified by comparing the theoretical values provided by tables 1 and 2 with the logs generated by the simulator. The execution unit is designed to reflect timing accurately to one clock cycle. Hence, the simulation results for the execution timing of each program instruction should match the theoretical timing values.

The simulation was run to execute the program shown in tables 1 and 2. The length of the simulation run included the occurrence of a (simulated) timer interrupt. The logs of every instruction's execution were verified to agree with the theoretical values, and the timer interrupt was verified to have occurred at the correct time.

4. CONCLUSIONS & FUTURE WORK

We have presented a new DEVS/HLA approach to model and simulate computer hardware platforms at the bus cycle level of abstraction. DEVS is used to model general hardware components as atomic models. The general components are then refined for a specific hardware platform. HLA services are used to support simulation coupling, event communication and time management. As a result, the generic DEVS simulation engine (middleware) found in other DEVS-based approaches is not required. An 8088-based case study has demonstrated and verified the approach.

Ideally, the larger-grained hardware platform models provided by the approach may yield simulation tools better suited to the needs of systems and software developers.

Future work is progressing in many directions:

1. The level of detail in the general DEVS models is being expanded to include: more instruction addressing modes, multiple bus masters, and alternate architectures such as DSP processors.
2. Tools are being developed to hide application developers from low-level details by generating the HLA access code automatically.
3. The automatic generation of HLA-compatible signalling protocols is being developed to allow the reuse of the larger grained simulations with finer-grained signalling protocols. This research is using XML representations of protocol timing diagrams to generate a more detailed layer of interaction behaviour.

REFERENCES

[1] Zeigler, B.; Kim, T.; Praehofer, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.

[2] Zeigler, B.P. "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions", Simulation Interoperability Workshop, March 1999. Orlando, FL.

[3] Daicz, S.; Troccoli, A.; Wainer, G. "Experiences in modeling and simulation of computer architectures using DEVS". In *Transactions of the Society for Modeling and Simulation International*. Vol. 18, No. 4. December 2001. pp. 179-202.

[4] 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.

[5] Wainer, G. "CD++: a toolkit to define discrete-event models". In *Software, Practice and Experience*. Wiley. Vol. 32, No.3. November 2002. pp. 1261-1306.

[6] Kim, T. G. "DEVSIM++ User's Manual", CORE Lab, EE Dept, KAIST, Taejon, Korea. 1994.

[7] IEEE standard for modeling and simulation (M&S;) high level architecture (HLA) - Framework and Rules. IEEE Std. 1516-2000 , Sep. 2000; Page(s): i -22

[8] IEEE standard for modeling and simulation (M&S;) high level architecture (HLA) - Federate Interface Specification. IEEE Std 1516.1-2000 , 2001; Page(s): 1 –467.

[9] Saghir, Amir "Computer System Modeling at the Hardware Platform Level" M.A.Sc. Thesis. Dept. of Systems and Computer Engineering, Carleton University. 2002

[10] Triebel A. Walter and Singh, Avtar "16-bit Micro-Processors. Architecture, Software and Interface Techniques."; Prentice-Hall, Inc, Englewood Cliffs, NJ. 2001.