DHMIF : DEVS-BASED HARDWARE MODEL INTERCHANGE FORMAT

Jun Kyoung Kim
Young Geol Kim
Tag Gon Kim
Systems Modeling Simulation Lab
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology(KAIST)
373-1 Kusong-dong, Yusong-gu, Taejon 305-701, Korea
E-mail: {jkkim, ygkim, tkim}@smslab.kaist.ac.kr

KEYWORD

CAD, DHMIF, HDL, Co-simulation, HW/SW Co-design, Design Reuse

ABSTRACT

The HW/SW co-design methodology and/or the IP (Intellectual Property) reuse methodology are often employed to implement functionalities on a given chip area with time-to-market requirement. However, digital hardware models used in the methodologies may be expressed in different description languages, thus being difficult, or even impossible, to integrate them in an overall model for simulation. An interchange format for hardware models could solve such a heterogeneous language problem if semantics for the format is capable of describing behavior of all hardware models. This paper introduces one such hardware model interchange format called DHMIF (DEVS-based Hardware Model Interchange Format) which is based on a sound semantics of the DEVS formalism. Being inherited from the underlying DEVS formalism DHMIF can represent any digital circuit, combinational and sequential, modeled in hardware description languages such as Verilog HDL or VHDL. Automatic translation from models in such languages into models in DHMIF makes it possible to simulate an overall model in a unified simulation environment. Case studies for both HW/SW co-design and IP reuse verified correctness of the DHMIF approach in co-simulation of a mix of models in hardware description languages and ones in the DEVSim++ environment.

INTRODUCTION

The progress of silicon technology enables us to implement more functionalities on a given chip area. But, design frameworks are not so efficient as to catch up with the progress of such silicon technology. The inefficiency could not satisfy the time-to-market requirement which is getting more and more important than ever for a company to survive. Therefore, required is an efficient design framework which meets such requirement in design of highly complex systems on a chip.

Many approaches have been made to overcome this problem. The first approach is codesign (Subrahmanayam 1993). The codesign approach can reduce the time-to-market by partitioning the hardware and software components in an early stage of system design, and developing each component independently as shown in the figure 1. The second approach is design reuse (Thomas 1999). Reusing pre-designed models along with newly designed ones can greatly reduce the design time. However, both approaches employ different description languages, thus being difficult, or even impossible, to integrate them in an overall model for simulation.

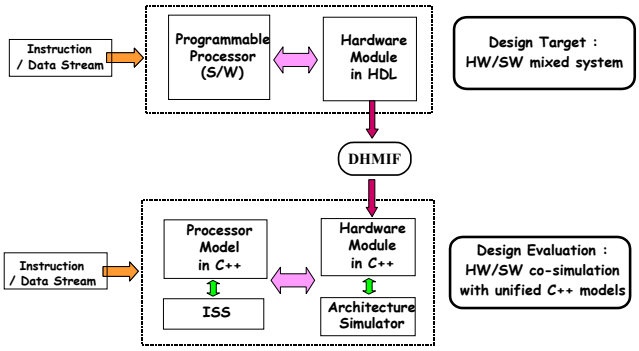


Figure 1: Typical Use of DHMIF in HW/SW Codesign

There are some solutions proposed to solve this language problem. With the assumption that modeling languages used are different, the number of simulation engine determines the taxonomy. The simplest method is to preserve the simulation framework for the respective part and make an interface. In this case, the number of simulation engines is equal to that of the modeling languages. The approach is relatively simple to implement compared to the translation method, which will be explained below. This method remains communication, synchronization and scheduling issues (Hubert 1998). Communication employs techniques such as pipes, socket, RPC(Remote Procedure Call), etc. The handshake is commonly used for synchronization, which degrades the simulation performance. If a simulation engine does not support the way of communication, this method is unavailable, thus it lacks extensibility. Another method is to translate all the models in different language into ones in a single language. This method uses only one simulation engine, thus renders us a faster simulation speed because all the simulation is performed on a simulation engine. A disadvantage of the approach is diffculty in translator implementation. The advantages and disadvantages are compared in Table 1.

Table 1: Comparison of IPC and Translation Method

	IPC	Translation
Advantage	Easy to implement	Fast
Dis-advantage	Synchronization problem Very slow Lack of extensibility	Hard to implement
Number of simulator	Same as the number of modeling language (>1)	One
Examples	CoSim(Valderrama et al.1996), Seamless(Klein and Leef, 1996)	PIA(Passerone et al., 1997)

There are two approaches for the translation method. One is direct translation and the other is semantics-based translation. Semantics plays an important role for the semantics-based translation approach. Sound semantics makes it easy to implement the translator and guarantees its correctness. The PIA approach based on Ptolemy also supports the semantics-based translation, but it restricts the hardware models to be expressed in the PIA language, thus being difficult to support design reuse. To our best knowledge, there is no framework that employs the semantics-based translation method except the PIA approach. Thus, an interchange format for hardware models could solve such a heterogeneous language problem if semantics for the format is capable of describing behavior of all the hardware models. The properties of such semantics are to be investigated with careful consideration of three orthogonal aspects of hardware models to be explained in the next section. This paper proposes a hardware interchange format based on the DEVS (Zeigler et al. 2000) formalism which is baed on sound semantics and can hold all the properties necessary for hardware modeling.

In section 2, we will see the semantics of the hardware that the interchange format must be based on. Section 3 explains DEVS formalism, a base formalism for DHMIF. The mapping relation between hardware and DEVS formalism are shown in section 4. Section 5 shows the syntax of the DHMIF. An application example of the DHMIF will be given in section 6. In section 7, we will remark conclusion and further work.

REQUIREMENT OF SEMANTICS FOR HARDWARE REPRESENTATION

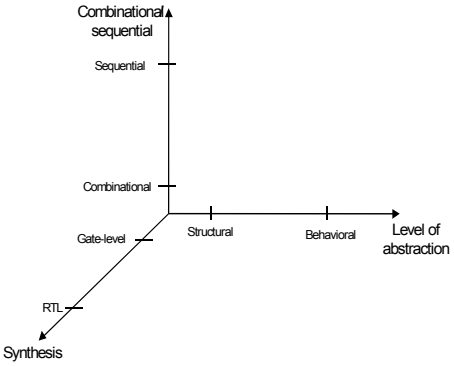


Figure 2: Aspects of Hardware Models

The properties of hardware determine the semantics of DHMIF. To figure out what properties the semantics of DHMIF must have, three orthogonal aspects of hardware are reviewed as in Figure 2.

First, the semantics must be capable of representing both combinational and sequential circuits in digital systems. To represent a combinational circuit, the semantics must represent output transition occurred by input. Some propagation delay may be necessary. On the other hand, a sequential circuit is characterized by state representation, state transition, output transition and the timing property on state transitions.

Second, there is a synthesis aspect. The primitives of RTL include register, arithmetic/logic unit, mux, counter, and so on. Therefore, the state representation, the state transition functions and output function are essential. The gate-level model is an interconnection of various gates, thus expressing the behavior of gates with propagation delay is required. Modeling a combinational logic in RTL does not make sense because a combinational logic has no state.

The third aspect is the level of abstraction. The behavioral level of abstraction requires representation of concurrency in addition to the properties mentioned above. The structural level of abstraction implies that a system can be a part of another complex system in a modular and hierarchical manner. This characteristic is even more important because the reuse methodology is noticed as a solution for designing a very complex hardware within feasible design time.

Semantics must be mathematically formal to apply a formal verification method such as model checking. Moreover, it needs to support the separation of a model and its interface which is an important characteristic for model reuse. The requirements for representing the hardware are summarized in Table 2.

Table 2: Requirements for Representing Hardware Models

Aspect		Requirements
Sequential/ Combinational	Combinational	Output evaluation for the input with delay(feedback free, no state)
	Sequential	State representation, state transition,timing for state transition
Synthesis	RTL description	State representation, state transition with timing property
	Gate-level description	Gate behavior with propagation delay, network of gates
Level of abstraction	Behavioral	Concurrency, state transition
	Structural	Modular/hierarchical modeling
Others	Formal verification, Separation of interface and behavior	

DHMIF SEMANTICS : DEVS FORMALISM

The DEVS(Discrete Event System Specification) formalism introduced by Zeigler is a set-theoretic formalism which provides a means of modeling discrete event system in a hierarchical, modular way. With this DEVS formalism, we can perform modeling more easily by decomposing a large system into smaller component models with coupling specification between them. There are two kinds of model in the DEVS formalism.

Atomic model is a basic model with specification for the dynamics of the model. It describes the behavior of a component, which is indivisible, in a timed state transition level. Formally, an atomic model M is specified by a 7-tuple as follows.

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

where

X : input events set
Y : output event set
S : sequential states set
δ_{int} : S → S is the internal transition function
δ_{int} : Q × X → is the external transition function, where
Q = {(s,e)|s∈S, 0≤e≤ta(s)} is the total state set
e is the time elapsed since last transition
λ : S → Y is the output function
ta : S → $R^{+}_{0,\infty}$ is the time advance function, where
$R^{+}_{0,\infty}$ is the set positive reals between 0 and ∞

The four elements in the 7-tuple, namely δ_{int}, δ_{ext}, λ and ta, are called the DEVS characteristic functions.

The second form of the model, called a coupled model(or coupled DEVS), tells how to couple several component models together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus giving rise to the construction of complex models in a hierarchical fashion. A coupled model DN is defined as follows.

DN = < X, Y, M, EIC, EOC, IC, SELECT >

where

X : input events set
Y : output events set
M : set of all component models in DEVS
EIC ⊆ X × $\cup_i X_i$: external input coupling relation
EOC ⊆ $\cup_i Y_i$ × Y : external output coupling relation

IC ⊆ $\cup_i X_i \times \cup_i Y_i$: internal coupling relation
SELECT : 2^M - ϕ → M is a function which chooses one model when more than 2 models are scheduled simulataneously.

The next section will describe how the DEVS formalism can be qualified as a semantics of a hardware representation.

HARDWARE-TO-DEVS RELATIONSHIP

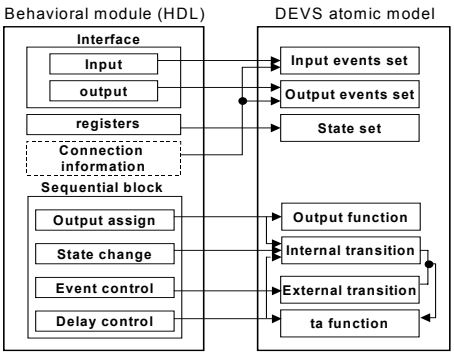


Figure 3: Hardware to Atomic DEVS Mapping: Behavioral Module without Concurrency

We will explain how hardware models in HDL can be translated to the DHMIF models. Figure 3 shows a simple correspondence between HDL and DEVS in which only one concurrent block is assumed. In this case, behavioral model in HDL corresponds to one atomic model in DEVS. The mapping from hardware properties to the DEVS formalism is as follows.

- Registers → States set
- input, inout → Input events set
- output, inout → output events set
- assignment to output → Internal transition, output and time advance function
- time control → Internal transition function and time advance function
- event control → external transition function of the DEVS

Figure 4 shows the mapping from the connected hardware to the DEVS model. Each HDL model should be mapped to DEVS atomic model as explained above and there should be one more model that has the coupling information between the two atomic models.

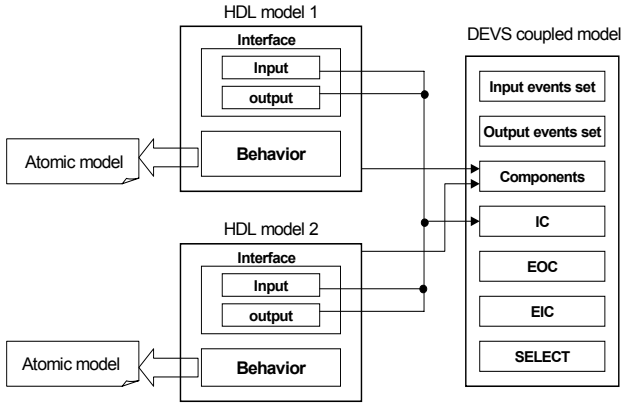


Figure 4: Hardware to Coupled DEVS Mapping: Structural Model

Lastly, let's look at the behavioral model with concurrency as in Figure 5. Each concurrent block is mapped to an atomic model and a coupled model is generated for coupling the atomic models. Input/output events of a hardware model are mapped to input/output events sets of a DEVS coupled model, respectively. Each

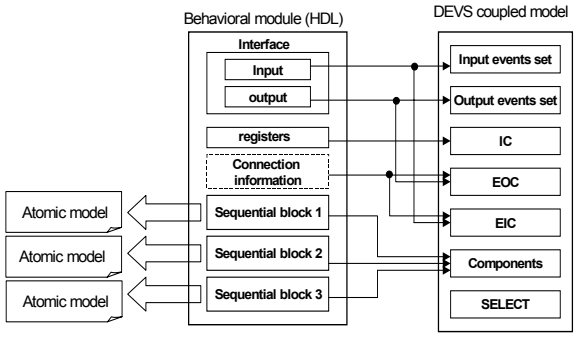


Figure 5: Hardware to Coupled DEVS Mapping: Behavioral Model with Concurrency

atomic model for each concurrent block becomes components of a DEVS coupled model.

A HDL model shares registers between concurrent blocks but DEVS atomic models corresponding to each concurrent block should have their own states set. This case causes a coherency problem, that is, a state change in one atomic model should take effect on states of others. The problem can be simply solved by specifying internal coupling for all the registers.

Adopting the DEVS formalism as semantics for DHMIF has some advantages in modeling and simulation inherited from the previous DEVS research. First and most importantly, DHMIF can represent hardware models in a unified semantics without loss of information on models described in any hardware description languages. Second, it is possible to apply the formal verification methods such as equivalence checking or reachability. Third, we can exploit the verified abstract simulation algorithms associated with the formalism, thereby efficiently constructing a simulation environment for hardware models in DHMIF. Fourth, the modular characteristic enables us to separate the interface and the behavior of the models, which is highly desirable to control design complexity. Lastly, the hierarchical models development framework underlying the DEVS formalism is well compatible to one used in hardware description languages such as Verilog HDL (Donald and Philip 1991). Such framework provides a sound means for composition of a larger system from previously constructed smaller components in a recursive manner.

PHASE REPRESENTATION of DEVS FORMALISM

We have seen that all the sequential machines can be represented by the DEVS formalism. However, for modeling convenience, a concept of phase variable needs to be introduced. This is because even a simple hardware with 4 4-bit wide registers leads to the number of $2^4 \times 2^4 \times 2^4 \times 2^4 = 65536$ states. Phase may reduce the number markedly.

<Definition> Triggering input event
If an input event causes the control flow of a machine to proceed, then it is said to be a triggering input event. Otherwise, it is a non-triggering input event.

For a synchronous system, clock is a triggering input and all the other inputs are non-triggering inputs. On the other hand, all the inputs are triggering inputs in an asynchronous system.

<Definition> Phase
A set of contiguous states is said to be phase if they are equivalent to
(i) non-triggering input events, and
(ii) internal or external state transition functions with basic operations

Figure 6 shows an example of the phase transition which is simpler than the original state transition with equivalent behavior. Suppose a machine with a reset and a clock input. A clock input event is a triggering input event and a reset input event is a non-triggering input event. With the DEVS formalism, there should be 4 states in total as in the state transition of Figure 6(a). The phase transition in Figure 6(b) shows how this state transition can be simplified to the phase transition. In this case, phase P_0 corresponds to the states S_0 and S_1. P_0 is a macro state of states S_0 and S_1 that is insensitive to the change of non-triggering reset input event. Instead, a phase variable should act as a condition variable. The state S_0 corresponds to (P_0, reset=0) state and the state S_1 corresponds to (P_1, reset =1) state of the phase transition.

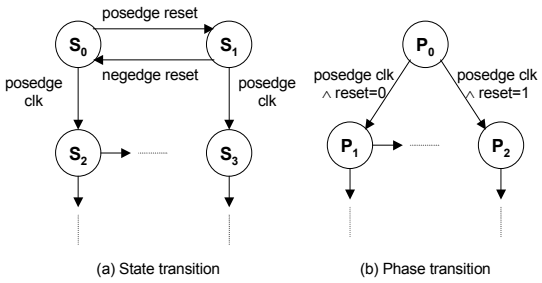
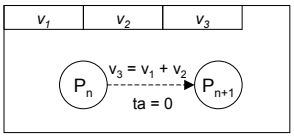


Figure 6: State Transition Diagram and Phase Transition Diagram

Basic operations in the phase representation include various HDL operations such as add, subtract, multiplication and so on. Suppose that there are 3 phase variables, *v1*, *v2* and *v3,* each 16-bit wide. Suppose also that adding *v1* and *v2* is stored in *v3* by internal transition function. Then, the number of different states before the addition is $2^{16} \times 2^{16} \times 2^{16} = 2^{48}$. Figure 7 shows representation of such states in the phase transition. The operation above is modeled with the DEVS formalism as follows.

$\delta_{int}((P_n, v_1, v_2, v_3)) = (P_{n+1}, v_1, v_2, v_3')$ where $v_3' = v_1 + v_2$
$ta((P_n, v_1, v_2, v_3)) = 0$

The phase transition for the external transition



Figures 7: Phase Representation for Basic Operations

function is defined in the same way. Introducing the phase concept provides the DEVS formalism with a syntatic sugar. The next section will show the syntax of DHMIF.

DHMIF SYNTAX

DHMIF is a language whose semantics can cover the semantics of hardware by adopting the DEVS formalism. In this section, the syntax of the DHMIF will be investigated. There are two kinds of DHMIF; atomic DEVS and coupled DEVS.

DHMIF for Atomic Model

DHMIF for an atomic model enables us to describe or express an atomic model of the DEVS with the syntax. As shown in Figure 8 description for interface and implementation of the model is separated. As a model user, one does not need to concern how this model is implemented, but only check if the interface of the model is consistent with other components.

```
interface module_name
    input: input_event_list;
    output: output_event_list;
    wire: wire_list;
end module_name

atomic module model_name
    parameter : parameter_declaration_list;
    state_variable : state_variable_list;
    combinational{ combinational_logic_expression}
    sequential{
        internal_transition: internal_transition_list;
        external_transition: external_transition_list;
        output_function: output_function_list;
        time_advance: time_advance_function_list;
}
end module_name
```

Figure 8: DHMIF Syntax for the Atomic Model

DHMIF for atomic model can describe both combinational and sequential circuits. Four characteristic functions are defined as in Figure 9, where condition and activity, introduced by the phase concept, are also shown. For all the characteristic functions, *phase*condition* corresponds to a state. For the two state transition functions, activity is defined so that basic operations are done on the phase variables.

external transition : *phase*input_event*condition → phase*activity*
internal transition *phase*real_number*condition → phase*activity*
output function : *phase*real_number*condition → output_event*
time advance function : *phase*condition → real_number*

Figure 9: DHMIF Syntax for DEVS Characteristic Functions

DHMIF for Coupled Model

DHMIF for the coupled DEVS also enables us to express coupled DEVS models. The syntax is defined as in Figure 10. This syntax also separates interface and coupling information.

```
interface module_name
    input: input_event_list;
    output: output_event_list;
    wire: wire_list;
end module_name

coupled module model_name
    component : component_list;
    intenal couping :intenal_coupling_list;
   external input couping :eic_list;
   external output coupling : eoc_list;
}
end module_name
```

Figure 10: DHMIF Syntax for the Coupled Model

DHMIF TRANSLATOR IMPLEMENTATION

We implemented a translator that translates Verilog HDL models to DEVSim++ (Hong and Kim 1994) models via DHMIF. Translation is based on the synthesizable subset of Verilog HDL defined at the IEEE1364 standard. Table 3 shows a summary on relationship between DEVS models and Verilog HDL models.

Table 3: Component Mapping between Atomic DEVS and Verilog HDL

DEVS tuple	Verilog HDL components
State Set	Registers
Input event	Variables declared as *input*, *inout*
Output event	Variables declared as *output*, *inout*
Internal transition	*#(delay time)*
	Assignment to output port
External transition	*@(event)*
	wait(event)
	Internal latch of input
Output function	Assignment to output port
ta function	*#(delay time)*

Translation of Verilog HDL models to DEVSim++ ones allows one to simulate hardware models using the DEVSim++ simulator developed at the SMSL(Systems Modeling Simulation Lab) in KAIST (Korea Advanced Institute of Science and Technology). An object-oriented environment DEVSim++ is a realization of the DEVS formalism and associated abstract simulator algorithms in C++. Currently, all the other components except Verilog HDL models are assumed to be DEVSim++ models.

Application Example 1 : Co-simulation

Codesign enables us to design a hardware-software mixed system in an efficient way by separating the hardware development process and software development process. At the co-simulation stage of codesign, independently

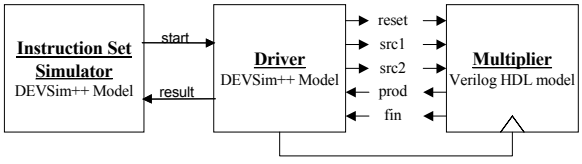


Figure 11: Co-simulation Example

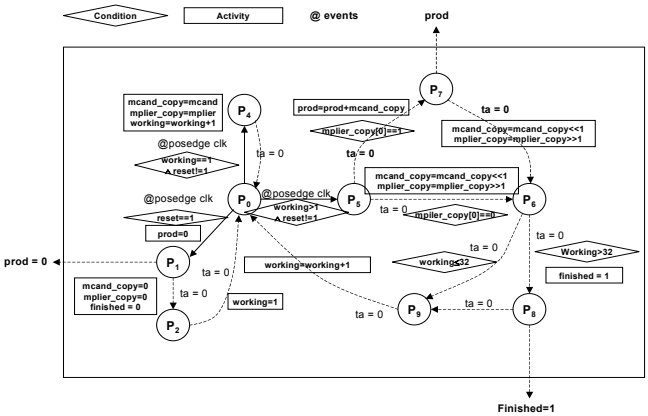


Figure 12: Phase Diagram for Multiplier Model

designed hardware components and software components are integrated and simulated to verify the correctness as well as to evaluate performance. Typically, models for hardware and software components are assumed to be given in Figure 11 as:

- Software is developed in DEVSim++ model
- Hardware is developed in Verilog HDL model

Figure 11 is a simple block diagram of our co-simulation example. Basic instructions are implemented at the ISS component as a DEVSim++ model. The ISS model represents a simplified VLIW(Very-Long Instruction Word) processor (Sima et al. 1997). When there is a multiplication instruction, ISS drives the multiplier hardware model via the interface to perform the instruction.

The co-simulation framework begins with translation of the hardware models in Verilog HDL into DEVSim++ models. The translation gives us all model components in a unified representation of DEVSim++ models. A simple 32bit by 32bit multiplier model is exemplified in Figure 12. This model implements a multiplier by using a simple add-and-shift algorithm and has a triggering input *clk*, non-triggering input events *mcand, mplier, reset* and output events *prod* and *finished*. *prod*, *mcand_copy, mplier_copy, working, finished* are phase variables. The number of states is more than $2^{64}\times2^{32}\times2^{32}\times32\times2^{1}$ which are reduced to 9 phases as shown in Figure 12.

Phase P_0 is an initial phase. When *reset* input is 1 and *clock* goes high, phase P_0 transits to P_1. By internal transitions with zero time advances, P_1 transits to P_2 and P_2 transits to P_0. These transitions are an initialization process. The next time *clk* goes high, non-triggering inputs *mcand* and *mplier* are copied to the phase variable *mcand_copy* and *mplier_copy,* respectively with phase P_4. After that, shifting and adding 32 times perfrom multiplication.

Figure 13 shows the process of translation from the Verilog HDL model to the DEVSim++ model via DHMIF. Figure 13(a) is the original multiplier model in Verilog HDL. The Verilog HDL to DHMIF translator generates the DHMIF model as in Figure 13(b). Note that tuples of the DEVS formalism are shown in the translation. Lastly, this DHMIF model is translated to the DEVSim++ model as shown in Figure 13(c). This DEVSim++ multiplier model is integrated with the ISS model and the driver model which are already written in DEVSim++. The simulation result is given in Figure 14. Figure 14(a) is an assembly program used as a benchmark program and Figure 14(b) is a simulation trace.

The seventh line of the assembly program has 3 multiplication instructions. When the instruction set simulator meets these instructions, it hands over the control to the multiplier. Figure 14(b) shows the simulation trace of first two multiplication instructions. The upper half of the Figure 14(b) is the simulation result for the first multiplication instruction. We verified that the result of multiplying fmul0_lr1(123) and lsu0_br0(700) is correctly stored to the fmul0_r(86100). The lower half of Figure 14(b) also shows the correct simulation result.

This application example demonstrates that with an appropriate interface model software components developed in DEVSim++ can be co-simulated with hardware components described in Verilog HDL, VHDL

```
module multiply(clk, reset, mcand, mplier, prod, fin);

input [31:0] mcand, mplier;
input clk, reset;
output [63:0] prod;
output fin;
reg [63:0] prod;
reg fin;

initial begin
   ... omitted : initialization ...
end

always @(posedge clk)
begin
   if(reset==1) begin
      fin = 0;
      ... omitted ...
   else begin
      if(working == 1) begin
         ... omitted ...
         prod(63,32)=prod(63,32)+mcand
         ... omitted ...
   end
end

endmodule
```

```
interface module multiply
   input:      clk; reset; mcand in 32; mplier in 32;
   output:     prod in 64; fin;
end multiply

atomic module multiply
   state register:   prod in 64; working in 8;
                     mcand_copy in 32; mplier_copy in 32;
                     fin;
sequential{
   internal transition:
      (1)*(0)*(reset==1) => (2)*(NULL);
      (1)*(0)*(!(reset==1)) => (5)*(NULL);
      ... omitted ...
      (9)*(0)*(TRUE) => (10)*(prod(63,32)=prod(63,32)+mcand_copy;);
      ... omitted ...
   external transition:
      (0)*(posedge clk)*(TRUE) => (1)*(NULL);
   output function:
      (2)*(0)*(TRUE) => (fin);
      (3)*(0)*(TRUE) => (prod);
      (6)*(0)*(TRUE) => (prod(31,0));
      ... omitted ...
   time advance function:
      (0)*(TRUE) => (INFINITY);
      (1)*(reset==1) => 0;
      (1)*(!(reset==1)) => 0;
      ... omitted ...
}
end multiply;
```

```
multiply::multiply(char* name):BehavioralModule(name)
{
   clk = 0;
   reset = 0;
   ... omitted ...
}
void multiply::intTransFn()
{
   if(phase==1){
      if(reset==1) phase = 2;
      else if(!(reset==1)) phase = 5;
   ... omitted ...
}
void multiply::activityFn()
{
   ... omitted ...
   if(phase == 9){
      if(TRUE)
         prod(63,32)=prod(63,32)+mcand;
   ... omitted ...
}
void multiply::extTransFn(Port& inPort, BitVector& inValue)
{
   ... omitted ...
   if(!strcmp(inPort, "clk"))
      if(inValue==1) phase = 1;
   ... omitted ...
}
```

(a) Source Verilog Model (b) DHMIF Model (c) DEVSim++ Model

Figure 13: Translation from the Verilog HDL Model to DEVSim++ Model via DHMIF

```
0 : LSU0.ldqi          0
1 : FMUL0.ld lsu0.br0,lr1; FMUL1, ld lsu0.br1,lr1; FMUL2.ld lsu0.br2,lr1; LSU0.ldqi 4;
2 : FMUL0.ld lsu0.br0,lr2; FMUL1, ld lsu0.br1,lr2; FMUL2.ld lsu0.br2,lr2; LSU0.ldqi 8;
3 : FMUL0.ld lsu0.br0,lr3; FMUL1, ld lsu0.br1,lr3; FMUL2.ld lsu0.br2,lr3; LSU0.ldqi 12;
4 : FALU0.ld lsu0.br0,lr1; FALU1.ld lsu0.br1,lr1;FALU2.ld lsu0.br2,lr1;
5 : LSU0.ldqi 257
6 : FMUL0.mul lsu0.br0,lr1;FMUL1.mul lsu0.br0,lr1; FMUL2.mul lsu0.br0,lr1;
7 : FMUL0.mul lsu0.br1,lr2;FMUL1.mul lsu0.br1,lr2; FMUL2.mul lsu0.br1,lr2;
```

(a) Compiled Code

```
================================================
------------------------------------------------
[PC = 14] fmul0mul fmul0_lr1 lsu0_br0
------------------------------------------------
 Before execution
------------------------------------------------
fmul0_r=-4.31602e+008(DEST)
fmul0_lr1=123(SRC)
lsu0_br0=700(SRC)
------------------------------------------------
 After execution
------------------------------------------------
fmul0_r=86100(DEST)
fmul0_lr1=123(SRC)
lsu0_br0=700(SRC)
================================================
------------------------------------------------
[PC = 14] fmul1mul fmul1_lr1 lsu0_br0
------------------------------------------------
 Before execution
------------------------------------------------
fmul1_r=-4.31602e+008(DEST)
fmul1_lr0=234(SRC)
lsu0_br0=700(SRC)
------------------------------------------------
 After execution
------------------------------------------------
fmul1_r=163800(DEST)
fmul1_lr0=234(SRC)
lsu0_br0=700(SRC)
================================================
```

(b) Simulation Trace

Figure 14: Simulation Result

or other hardware description languages.

Application Example II : Design Reuse for ARM Processor

To show the effectiveness of DHMIF for the design reuse framework, we constructed a simple arm processor with some components pre-designed in Verilog HDL. This processor model consists of four pipeline stages, instruction fetch, instruction decode, execution and memory writeback stages. Figure 15 shows the overall block diagram of the ARM processor. The six Verilog HDL models, *if_id_reg.v*, *pc_part.v*, *alu_mul.v*, *controller.v*, and *clock.v,* with gray boxes in the figure are Verilog HDL models which are already in existence and

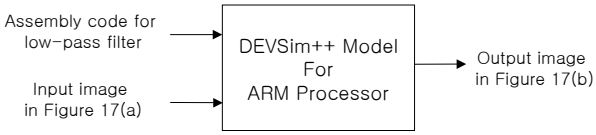


Figure 16: DEVSim++ Simulation of ARM Processor

resused. All other components are newly developed DEVSim++ models. By translating the Verilog HDL models to DEVSim++ models via DHMIF and integration with the existing DEVSim++ models, an overall processor model was constructed. The processor model was simulated using the DEVSim++ simulator with an input assembly code of a 2-D image low-pass filter as shown in Figure 16. The simulation result verified correctness and usefulness of the DHMIF for the design reuse framework. Figure 17(a) shows the input image for simulation and Figure 17(b) is the image after lowpass filtering through the simulator. Note that the effect of lowpass filtering made the original image to be blurred at boundaries

(a) Original image (b) Image after filtering

Figure 17: Lowpass Fitering with Simulated ARM Processor

CONCLUSION AND FURTHER WORK

This paper proposed a DEVS-based hardware model interchange format, DHMIF that supports a formal means to integrate hardware models described with different languages into models in a unified representation. The translation method based on the DHMIF semantics gunrantees correctness of hardware models interchange into DEVSim++ models. Moreover, the use of a single simulation engine in C++ with C++ hardware models increases simulation speed markedly.

Based on the DHMIF semantics a translator was implemented which translates Verilog HDL models into

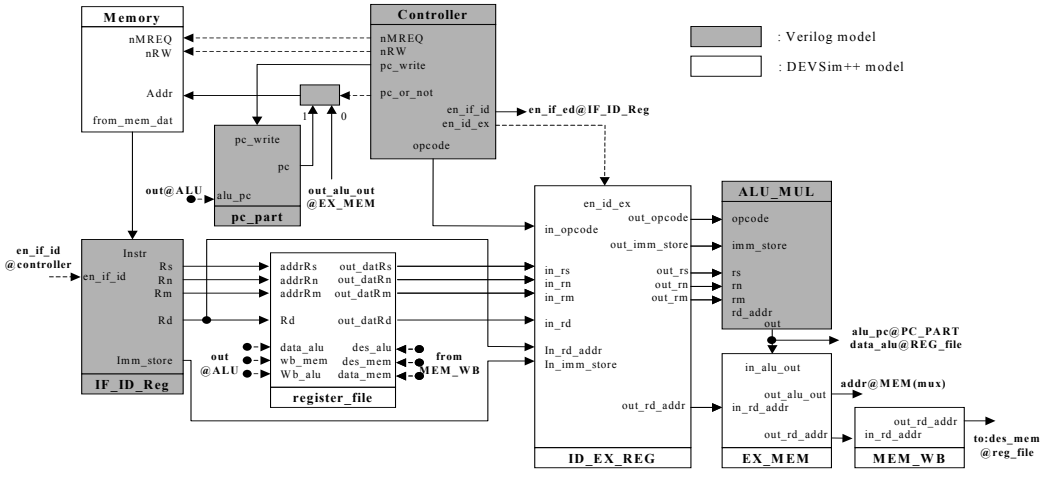


Figure 15: Block Diagram of the Deisgn Resue Example

DEVSim++ models. Simulation of a mixture of Verilog HDL models and DEVSim++ models was performed and produced correct result. Applications of such simulation were demonstrated in two important areas in SOC(system-on-chip): HW/SW co-desing and IP resue.

DHMIF can be used as a common representation for different hardware description languages. Bi-directional translation based on the representation may support automatic transformation of one hardware model(i.e. Verilog HDL) to another(i.e. VHDL). Such transformation may be required to interchange hardware models developed in different description languages before integrated simulation. Because DHMIF is based on the formal semantics of DEVS, a formal verification method, such as equivalence checking or reachability analysis, can be applied. Such transformation and formal verification of models in DHMIF remains as future work.

REFERENCES

B.P. Zeigler, H. Praehofer and T.G. Kim. 2000. *Theory of Modeling and Simulation, 2nd edition*, Academic Press

C.A. Valderrama, Francois Nacabal, Pierre Paulin, A.A. Jerraya, 1996. "Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems: An Industrial Experience", *7th International Workshop on Rapid Systems Prototyping*, Greece

C.Passerone, L.Lavagno, C.Sansoe, M.Chiode, A.Sangiovanni-Vincentelli, 1997. "Trace-off Evaluation in Embedded System Design via Co-simulation", *Proceedings of the ASP-DAC'97*, Chiba, Japan

Desco Sima, Terence Fountain, Peter Kacsuk, 1997. *Advanced Computer Architectures – A Design Space Approach*, Addison-Wesley

Donald E. Thomas, Philip R. Moorby. 1991. *The Verilog Hardware Description Language*, Kluwer Academic Publishers, Boston

Gyung Pyo Hong, Tag Gon Kim. 1994. "The DEVS formalism : a framework for logical analysis and performance evaluation for discrete event systems", *AI, Simulation, and Planning in High Autonomy Systems*. pp170 –175

H. Hubert, 1998. *A Survey of HW/SW Cosimulation Techniques and Tools*, M.S. Thesis, Royal Institute of Technology, Sweden

P.Subrahmanayam, 1993. "Hardware-Software Codesign : Cautious Optimism for the Future", *IEEE Computer*, 26(1), pp84-85

Russel Klein, Serge Leef, 1996. "New Technology Links Hardware and Software Simulators", *Electronic Engineering Times*

Thomas, T., 1999. "Technology for IP reuse and portability", *IEEE Design and Test of Computers,* Volume 16, Issue 4, Oct. 1999, pp 7-13

AUTHOR BIOGRAPHY

JUN-KYOUNG KIM was born in Cheong-ju, Korea and received B.S. degree in electronic engineering from Yonsei University. He graduated from KAIST with M.S. in electrical engineering and is a Ph.D student at KAIST. His research interests include discrete event systems modeling/ simulation, processor design, co-design and processor description language. His e-mail address is jkkim@smslab.kaist.ac.kr.

YOUNG GEOL KIM was born in Seoul, Korea and received B.S. and M.S in electrical engineering from KAIST. He is a Ph.D student at KAIST. His research interests include architecture description language design, retargetable simulator and codesign framework. His e-mail address is ygkim@smslab.kaist.ac.kr.

TAG GON KIM received his Ph.D. in computer engineering with specialization in methodology for systems modeling/simulation from University of Arizona, Tucson, AZ, 1988. He was a Full-time Instructor at Communication Engineering Department of Bukyung National University, Pusan, Korea between 1980 and 1983, and an Assistant Professor at Electrical and Computer Engineering at University of Kansas, Lawrence, Kansas, U.S.A. from 1989 to 1991. He joined at Electrical Engineering Department of KAIST, Tajeon, Korea in Fall, 1991 as an Assistant Professor and has been a Full Professor since Fall, 1998. His research interests include methodological aspects of systems modeling simulation, analysis of computer/communication networks, and development of simulation environments. He has published more than 100 papers on systems modeling, simulation and analysis in international journals/conference proceedings. He is a co-author (with B.P. Zeigler and H. Praehofer) of the book *Theory of Modeling and Simulation* (2nd ed.), Academic Press, 2000. He is the Editor-in-Chief of *Transactions of Society for Computer Simulation* published by Society for Computer Simulation International(SCS). He is a senior member of IEEE and SCS and a member of ACM and Eta Kappa Nu. His e-mail address is tkim@ee.kaist.ac.kr.