Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

A HETEROGENEOUS SIMULATION FRAMEWORK BASED ON THE DEVS BUS AND THE HIGH LEVEL ARCHITECTURE

Yong Jae Kim
Tag Gon Kim

Department of Electrical Engineering
Korea Advanced Institute of Science and Technology
373-1, Kusung-Dong, Yusung-Gu, Taejun, 305-701, KOREA

ABSTRACT

We describe a heterogeneous simulation framework in which conventional simulation models and the DEVS (Discrete Event Systems Specification) models can be interoperable. The framework conceptually consists of three layers: the model layer, the DEVS layer, and the HLA (High Level Architecture) layer. The model layer has a collection of heterogeneous simulation models, such as DEVS, CSIM, SLAM, and so on, to represent various aspects of a complex system. The DEVS layer provides a common framework, called the DEVS BUS, so that such simulation models can communicate with each other. Finally, the HLA layer is employed as a communication infrastructure, which supports several good features for distributed simulation. The DEVS BUS has been implemented on the HLA and a simple example of communicating two heterogeneous models has been developed to validate the DEVS BUS.

1 INTRODUCTION

A heterogeneous simulation includes many simulators having different simulation methodologies, each of which is dedicated to an aspect of a complex question. For example, simulation for a manufacturing system may include a scheduler, a harbor, a traffic, a factory, an AS/RS, and an ecological simulator. The simulators run concurrently for answering the complex question.

High Level Architecture (HLA) has been defined in the DoD M&S sub-objective 1-1 (DoD 1995): “Establish a common high-level simulation architecture to facilitate the interoperability of all types of models and simulation among themselves and with C4I systems, as well as to facilitate the reuse of M&S components". The HLA, however, gives no formal way to model a system.

When an existing simulation model such as CSIM, SLAM, and so on, wants to join a federation, the simulation model should be modified so that the model can send(receive) external messages to(from) the other federates. Such modifications seem difficult and sometimes may be impractical. In this paper, we propose an alternative way to heterogeneous simulation using the DEVS BUS approach, in which existing simulation models need not to be modified.

Kim and Kim (Kim and Kim 1996b) proposed the DEVS BUS that virtually connects the supervisory simulation model and node simulation models. They also proposed a very simple protocol conversion method that can be used only for server models. In this paper, we refine the DEVS BUS and develop a general protocol converter using a system theoretic approach (Kim and Kim 1998).

The rest of this paper is organized as follows. Section 2 describes an overview of the framework. Section 3 reviews the DEVS formalism and describes the DEVS BUS architecture. Section 4 develops a DEVS/CSIM simulation protocol converter with which a CSIM model can be attached to the DEVS BUS. Sections 5 and 6 present an implementation and an execution of a DEVSim-HLA environment, respectively. Finally, some conclusions and future works are given.

2 OVERVIEW OF THE FRAMEWORK

The goal of the proposed framework is to provide a common simulation infrastructure for heterogeneous simulation, in which constructive simulations, live components, and human interactions can be interoperable. The infrastructure should have a simple, well-defined interface so that a simulator can easily participate in a heterogeneous simulation.

We propose the DEVS BUS approach as shown in Figure 1. There are conceptually three layers: the model

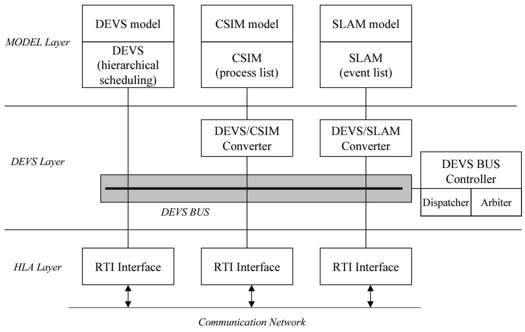

Figure 1: DEVS BUS Approach

layer, the DEVS layer, and the HLA layer. Each simulation node, called a *federate*, consists of a simulation model, a simulation protocol converter, and an HLA simulation facility. Federates communicate with each other via a communication network.

The model layer has a collection of heterogeneous simulation models, such as DEVS, CSIM, SLAM, and so on, to represent various aspects of a complex system. There are two main advantages of using heterogeneous simulation models: modeling power and reusability. The layer enables a modeler to build a global model with a combination of world views to answer a complex question. Also, well-developed simulation models can be reused so that we can build rapidly an overall simulation model.

The DEVS layer provides a common framework so that such simulation models could communicate with each other. Simulation models participating in a heterogeneous simulation, however, may not communicate directly with each other due to different simulation protocols: simulation protocol conversion is required. A protocol converter is an interface module between different simulation protocols. For example, a DEVS/CSIM converter translates DEVS requests into CSIM messages and vice versa. Note that such a converter does not translate each simulation model into a DEVS model but enable simulation models to communicate with each other. The DEVS BUS is virtually located between simulation models. Instrumented with protocol converters, the DEVS BUS coordinates communications between simulation models while preserving causal relationships of events.

Finally, the HLA layer is employed as a communication infrastructure, which supports several good features for distributed simulation. For example, the Run-Time Infrastructure(RTI) of the HLA supports a time advance mechanism based on the conservative approach and several message delivery schemes such as receive and time stamp ordered. Moreover, the layer enables us to enlarge our simulation framework to include live components and human interactions.

3 DEVS BUS

Before describing the DEVS BUS, we briefly review the DEVS formalism and abstract simulators for DEVS models.

3.1 DEVS Formalism

DEVS (Discrete Event System Specification) is a set-theoretic formalism to specify discrete event systems (Zeigler 1984). There are two kinds of models, atomic and coupled. An atomic model, called AM, specifies the dynamics of a model and is defined as:

Definition [AM]

$AM = < S, X, Y, \delta_{ext}, \delta_{int}, \lambda, ta >$
where
S: sequential states set,
X: input events set,
Y: output events set,
δ_{ext}: $Q \times X \rightarrow S$, external transition function
where Q is the total state set of
$Q = \{(s, e) | s \in S \, and \, 0 \leq e \leq ta(s)\}$,
δ_{int}: $S \rightarrow S$, internal transition function,
λ: $S \rightarrow Y$, output function,
ta: $S \rightarrow \mathcal{R}^+_{0,\infty}$, time advance function
where the $\mathcal{R}^+_{0,\infty}$ is the non-negative real numbers with ∞ adjoined.

The interface of an atomic model is defined by X and Y. The model can process events defined at X and produce events defined at Y. δ_{ext} and δ_{int} specify how to change the states of the model. An output event is produced at a state according to λ. Finally, a sojourn time of a state is defined by ta.

A coupled model provides the way of composition of several atomic and/or coupled models. When we want to specify a complex system, we can specify each of subcomponents individually and construct big one using the coupled model, which has only structural informations and is defined as:

Definition [CM]

$CM = < X, Y, \{M_i\}, EIC, EOC, IC, SELECT >$
where
X: input events set,
Y: output events set,
M_i : component basic model,

$EIC \subseteq X \times \cup X_i$: external input coupling relation,
$EOC \subseteq \cup Y_i \times Y$: external output coupling relation,
$IC \subseteq \cup Y_i \times \cup X_j$: internal coupling relation,
SELECT : $2^M - \phi \rightarrow M_i$, tie-breaking selector.

M_i can be an atomic and/or coupled model. EIC specifies how to route external messages to M_i and EOC how to route output events of M_i to the outside of CM. An output event of M_i is sent to M_j according to IC. Finally, SELECT is a tie breaking function.

3.2 DEVS Abstract Simulator

Attached to each DEVS model is an associated abstract simulator, either a *simulator* for an atomic model or a *coordinator* for a coupled model (Zeigler 1984). Consider Figure 2, where a solid line with $?event$ corresponds to an external state transition by an external input event and dashed one with $!event$ represents an internal state transition with an external output event. Two simulators, S1 and S2, are managed by a coordinator, COOR, which is not shown in the Figure. Assume that S1 wants to send an output event to S2. After receiving $(*, t)$, S1 produces an output (y, t) by executing λ and sends it to COOR. Then, S1 changes its state as defined in δ_{int}, calculates a sojourn time, $tN1$, of a new state using ta, and sends a $(done, tN1)$ to COOR. After receiving (x, t), S2 updates its state according to δ_{ext} and sends $(done, tN2)$ to COOR. The abstract simulator algorithm is a composition of that of S1 and S2, as shown in the lower part of Figure 2.

The hierarchical simulation algorithm for a coupled model, PEL, which has two atomic models, BUFF and PROC, is shown in Figure 3. BUFF and PROC have associated simulators of S:BUFF and S:PROC, respectively. The coupled model, PEL, has the associated coordinator of C:PEL. Finally, R:PEL is the *root-coordinator* whose job is to manage the overall simulation clock.

Assume that the next simulation time is 10 and BUFF produces an output at 10. First, R:PEL sends $(*, t = 10)$ to C:PEL. C:PEL routes the message to its component, whose tN is 10. In this case, C:PEL routes $(*, 10)$ to S:BUFF. S:BUFF requests BUFF to execute consecutively the output function, the internal transition function, and the time advance function of BUFF while producing an output message, $(y, 10)$. S:BUFF sends $(y, 10)$ to C:PEL. Then, C:PEL translates $(y, 10)$ into an input message, $(x, 10)$, and sends it to S:PROC. After receiving the input message, S:PROC requests PROC to execute the external transition function and the time advance function. Then, S:PROC reports $(done, tN1 \geq 10)$ to C:PEL, which indicates the next event time of S:PROC is $tN1$. Also, S:BUFF reports its next event time by sending $(done, tN2 \geq 10)$ to C:PEL.

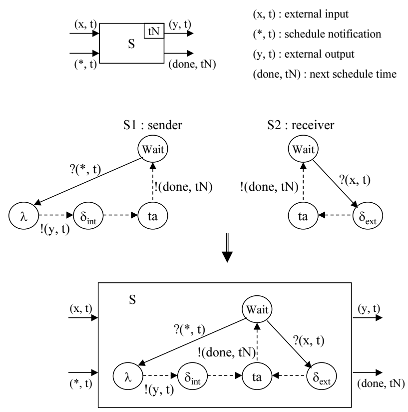

Figure 2: DEVS Simulator Algorithm

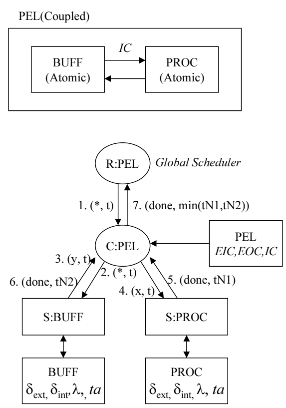

Figure 3: Hierarchical Simulation Algorithm for DEVS Coordinator

The new tN of C:PEL is set to the minimum of two tNs and reported to R:PEL by sending $(done, min(tN1, tN2))$. Once R:PEL receives the message, it updates the simulation clock into $min(tN1, tN2)$ and sends $(*, min(tN1, tN2))$ to C:PEL.

There are four kinds of message in the algorithm: $(*, t)$, $(done, tN)$, (x, t) and (y, t). The former two messages are used for simulation scheduling and the latter two for data transfer.

3.3 DEVS BUS Architecture

A computer bus serves as a shared communication link between various parts of a computer system. A *master* is a device that can initiate a communication with a responding device, which is called a *slave*. A bus has multiple masters when there are multiple CPUs or when I/O devices can initiate a communication. An *arbitration* is a mechanism to resolve conflicts that arise when more than two masters try to use the bus at the same time. A device that is dedicated to the arbitration is called a *bus arbiter*.

The bus has the two major advantages: low cost and versatility (Hennessy and Patterson 1990). The cost is low because a single set of wires is shared by several devices. We can add new devices to the bus by implementing a single interconnection scheme already well defined. On the other hand, a communication bottleneck is the major disadvantage of the bus. If the bus is in use, a device that is newly trying to use it should wait until it becomes free.

The basic idea of the DEVS BUS is the same as that of the hardware bus. The approach may arise a bottleneck problem as the hardware bus and also has the advantage of the common interface. When a simulator wants to send a message to others, it should wait until granted to use the bus. When a simulator wants to join a heterogeneous simulation, it comes true if the simulator just implements the DEVS BUS protocol.

Figure 4 shows the DEVS BUS architecture. There are four communication paths between the DEVS BUS controller and node simulators. (x, t) and (y, t) are for data transfer. $(*, t)$ corresponds to a *bus grant* of the hardware bus. $(done, tN)$ has the composite meaning of a *bus release* and a *bus reservation*. The DEVS BUS controller consists of a dispatcher and an arbiter. Basically, the dispatcher is a coupling scheme of a coupled DEVS and the arbiter is the root coordinator of the hierarchical simulation algorithm for the coupled DEVS. The dispatcher receives data from source model and forwards it to destination model. The arbiter selects a simulator among several simulators so that the simulator exclusively use the DEVS BUS for an instant.

Once a simulator receives $(*, t)$, it use the bus and eventually sends $(done, tN)$ as a bus release/reservation. The bus reservation reports it to the dispatcher that the simulator should be scheduled at the next event time tN. So, whenever a simulator receives $(*, t)$ or (x, t), it sends $(done, tN)$ to the dispatcher. It differs from a *bus request* of a common hardware bus, in which a master want to use the bus not later but immediately.

An addressing scheme should be considered to correctly transfer data. Actually, a hardware bus arbiter only deals with control signals. Data read and write operations are performed between a master and a slave. The master should select the designated slave among several slaves according to the predefined addressing scheme. On the other hand, in the DEVS BUS, the bus dispatcher determines the destination simulator. The dispatcher has all connection information, called a coupling scheme. The coupling scheme is a relation in which all pairs of source and destination models are specified. When a simulator

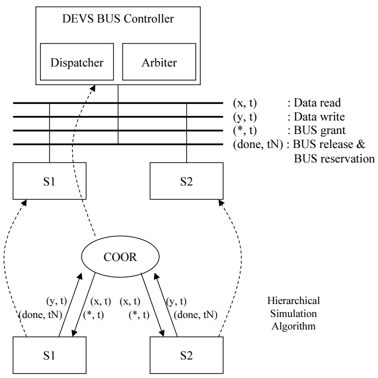

Figure 4: DEVS BUS Architecture

produces (y, t), the bus dispatcher translates it into (x, t) and forwards (x, t) to the destination simulator as specified in the coupling scheme.

Specification of connection information in the coupling scheme, not in models, gives much flexibility in changing destination simulator. Consider that some models of $S2$ are moved into $S3$. There is no need for $S1$ to know the movement. $S1$ just sends (y, t) to the dispatcher not to directly S2 or S3.

A possible scenario of the DEVS BUS arbitration is shown in Figure 5. Initially, both simulators report their tNs to the arbiter. When the arbiter receives both messages, it determines that a simulator with the smaller tN, S1, can use the bus. The arbiter sends $(*, 3)$ to S1. Once S1 receives the message, it produces $(y, 3)$ to the dispatcher. Then, the dispatcher translates it into $(x, 3)$ and forwards $(x, 3)$ to S2. After receiving $(x, 3)$, S2 reports its tN to the arbiter by sending $(done, tN = 5)$. Also, S1 produces $(done, tN = 10)$. Then, the arbiter generates $(*, 5)$ so that S2 can use the bus and so on.

Table 1 shows a comparison between the DEVS BUS and a common hardware bus. There are two differences between them: bus request and addressing. Scheduled is a bus request of the DEVS BUS, by which a simulator reserves the bus for a future use. On the other hand, immediate is that of the hardware bus, by which a master can use the bus right away if granted. Because the dispatcher in the DEVS BUS controller has all addressing information, the simulator just sends data to the dispatcher. In the hardware bus, however, the arbiter only controls bus arbitration and the master should know a destination address. Specification of connection information in the dispatcher, not in models, gives much flexibility in changing the destination address. We did not define a bus protocol for data transfer such as timing requirements used for a

Table 1: Comparison between DEVS BUS and Hardware BUS

Feature	DEVS BUS	Hardware Bus
Bus request	$(done, 0)$	BR
Bus reservation	$(done, tN)$	N/A
Bus grant	$(*, t)$	BG
Bus release	$(done, tN)$	BREL
Addressing Method	Dispatcher base	Source base

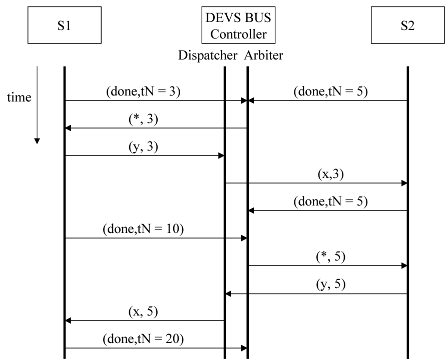

Figure 5: DEVS BUS Arbitration

hardware bus. Those requirements are considered useless for our purpose.

4 SIMULATION PROTOCOL CONVERSION

4.1 DEVS/CSIM Simulation Protocol

Conventional simulation environments can be easily added to the DEVS BUS by using a dedicated simulation protocol converter. DEVS models are interpreted using the hierarchical simulation algorithm. Simulation methodologies for conventional simulation models, however, differ from that of the DEVS models. When a DEVS model wants to communicate with a conventional model, a *simulation protocol mismatch* exists and should be resolved.

In this section, we consider a heterogeneous simulation environment that consists of a DEVS simulation model and a CSIM simulation model. We design a DEVS/CSIM simulation protocol converter to resolve the mismatch using a system theoretic protocol conversion methodology (Kim and Kim 1998). Generally speaking, the protocol conversion problem is to find a missing component that is connected with two end components while satisfying a given high level specification. In the protocol conversion methodology, the two components and the high level specification are described in the DEVS formalism and a protocol converter, the missing component, is found algebraically.

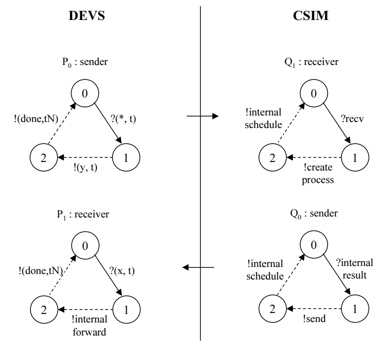

Figure 6: DEVS-CSIM Communication

Figure 6 describes the DEVS and the CSIM simulation protocol. The DEVS simulation protocol consists of a sender, P_0, and a receiver, P_1. Q_0 and Q_1 are those of the CSIM simulation protocol. The simulation protocols only include capabilities on communication with other simulation models. We abstracted the details such as interactions with models and scheduling algorithms.

Assume that P_0 sends a message to Q_1 at $t = 0$. Initially, the DEVS BUS arbiter, Arb, receives two different messages, $(done, tN_P = 0)$ and $(done, tN_Q > 0)$, from the DEVS and the CSIM simulator, respectively. (Let's assume $(done, tN_Q)$ can be sent.) The next scheduling time, tN, is set to the minimum of tN_P and tN_Q, that is $tN = 0$. Once tN is determined, Arb grants one of two simulators to use the bus by sending $(*, tN = 0)$ to the simulator. In this case, the DEVS simulator is granted. After receiving $(*, 0)$, P_0 produces an output event, $(y, 0)$, which is eventually sent to Q_1. Then, P_0 reports its next scheduling time to Arb by sending $(done, tN_{P'})$. When P_1 receives an external input message, (x, t), from Q_0, it

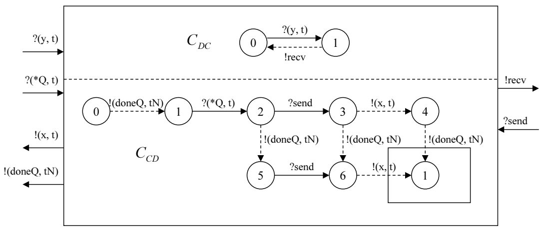

Figure 7: DEVS/CSIM Protocol Converter

forwards the message to the destination model and sends $(done, tN_{P''})$ to Arb.

On the other hand, Q uses $send$ and $recv$ messages instead of $(done, tN)$, $(*, t)$, (x, t), and (y, t). When Q_1 receives an external $recv$ message, it creates a process to perform jobs for the message and is internally rescheduled. If there is an internal result, Q_0 produces a $send$ message and is rescheduled.

Evidently, the simulation methodology of the DEVS model is different from that of the CSIM simulation model: their simulation protocols are mismatched. Because of the protocol mismatch, the DEVS model can't directly communicate with the CSIM model. We should develop a simulation protocol converter, which makes the CSIM model interoperable with the DEVS model.

4.2 DEVS/CSIM Simulation Protocol Conversion

We build a protocol converter that can be decomposed into two separate parts (Figure 7). C_{DC} is for the communication path from P_0 to Q_1 and C_{CD} from Q_0 to P_1. C_{DC} and C_{CD} are individually found using the system theoretic approach (Kim and Kim 1998). The resulting converter, C, is constructed by composition of C_{DC} and C_{CD}. When C is to be constructed directly from P and Q, the complexity may be high. The decomposition of C into C_{DC} and C_{CD} is efficient.

5 IMPLEMENTATION OF DEVSIM-HLA

In this section, we describe the connection between the DEVS layer and the HLA layer. The environment is developed on the RTI version 1.0.3 (DMSO 1997) using the D-DEVSim++ simulation environment (Kim et al. 1996a) and the CSIM environment (Schwetman 1988).

5.1 Implementation of the DEVS BUS Protocol

In the RTI, federates communicate with each other in two ways: object and interaction. An object represents a simulation entity and has several attributes for states of the entity. On the other hand, an interaction is best suited to represent a message between federates.

The DEVS Bus protocol has four kinds of messages, each of which corresponds to an interaction. To route the message correctly, we add some routing informations such as address and port information. The interactions are considered as reliable TSO messages.

5.2 Time Management

There are two factors to determine time management service in RTI: time constrained and time regulating (DMSO 1996). Time constrained indicates whether the federate will be constrained by the logical time of other federates; time regulating indicates whether the federate proposes to participate in determining the logical time of other federates. Time constrained federates can receive time-stamp ordered (TSO) messages and time regulated federates can send them.

Among four possible different services of time management according to the two factors, we use the logical time synchronized service so that a federate participates in other federate's time advance decisions and accepts such participation from other federates. The federates can send and/or receive TSO messages.

There exists a semantic gap between the RTI time management and the time advance mechanism of the DEVS BUS protocol. Consider a logically synchronized federate. In the RTI, the fact that the current time of the federate is 2 means that there is no more external TSO message with a time-stamp less than or equals to 2, that is $ts \leq 2$. The federate can only generate messages with $ts \geq 2 + lookahead$. On the other hand, the DEVS BUS protocol uses next schedule times, tNs. $tN = 2$ means that the arbiter makes $(*, 2)$. Once a simulator receives $(*, 2)$, it sends $(done, tN)$ after a set of executions of the output function, the internal transition function, and the time advance function. At this time, another $tN = 2$ is possible if a *zero time advance* is modeled. So, in the DEVS BUS, there may be possible $tN = 2$ after processing $(*, 2)$.

The problem is more difficult when we consider a (y, t) message routing. Assume that a coupled model, c_0, which is mapped into a federate A, consists of two atomic models, a_1 and a_2 which are mapped into another federate B. Consider a_1 wants to send $(y, 2)$ to a_2. Then, a_1 should send the message to c_0 because in the DEVS formalism, a basic model in a coupled model can not directly send an output message to another in the coupled model. Once c_0 receives the message at $t(FedA) = 2$, c_0 should send $(x, 2)$, the translated message of $(y, 2)$, to a_2 at $t(FedA) = 2$. In the RTI, however, it's illegal because $ts = 2 < 2 + lookahead(> 0)$.

We solve the problem using two epsilons scheme, which uses predefined small values, $\epsilon 1$ and $\epsilon 2$, while preserving the overall logical sequence of events. $\epsilon 1$ is used to resolve the zero time advance problem by adding $\epsilon 1$ to tN whenever a zero time advance occurs. $\epsilon 2$ is used for the (y, t) message problem. When the message time of a (y, t) is the same as the federate's current time, $\epsilon 2$ is added to the request message to the RTI, while preserving t of (y, t). $\epsilon 1$ is slightly modified from the ϵ-delay scheme (Kim et al. 1997) and $\epsilon 2$ is from the $EPSILON$ of the RTI (DMSO 1997).

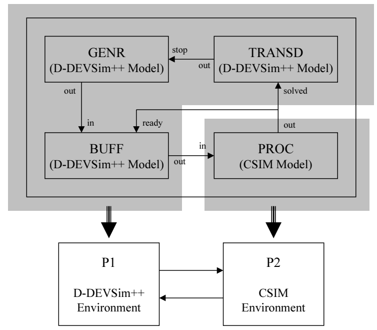

Figure 8: EF_PEL Model

6 AN EXECUTION

We develop a simple example, called EF_PEL (Figure 8), which consists of four different components. The generator produces jobs at a predefined rate and sends them to the buffer. Once receiving a job, the buffer forwards it to the processor if the processor is free, otherwise the buffer saves it until the processor is available. After finishing the job, the processor reports a result to the transducer and sends a message to the buffer so that another job can be sent. When a termination condition meets, the transducer sends a stop message to the generator so that no more jobs are generated.

We build the EF_PEL simulator using two federates. The processor model is developed as a CSIM model and mapped into the Federate P2. The others are DEVS models and mapped into the Federate P1. To enable communication between two simulation models, we use the DEVS/CSIM protocol converter constructed at the previous section.

The DEVS simulator and the CSIM simulator run concurrently. (x, t) and (y, t) messages are well passed obeying timing constraints. The simulation goes well so that every jobs generated are processed in the processor model and finally reported to the transducer model. We can get the statistics of facilities of the processor model from the CSIM environment and the overall performance results from the DEVS environment.

7 CONCLUSIONS AND FUTURE WORKS

We have described a software bus, called the DEVS BUS, as a common simulation infrastructure for heterogeneous simulation. The DEVS BUS provides a well-defined interface so that a simulator could be easily added to heterogeneous simulation just by implementing the interface. The DEVS BUS controller consists of a dispatcher and an arbiter. Basically, the dispatcher is a coupling scheme of a coupled DEVS and the arbiter is the root-coordinator of the hierarchical simulation algorithm associated with the coupled DEVS.

We have implemented the DEVSim-HLA, a heterogeneous simulation environment based on the DEVS BUS and the High Level Architecture. Currently, the environment consists of the D-DEVSim++ environment and the CSIM environment on the Run-Time Infrastructure of the HLA. A DEVS/CSIM simulation protocol converter is implemented to provide the DEVS BUS. The EF_PEL model showed that the framework is a feasible solution to heterogeneous simulation.

To show advantages of our framework, we'll evaluate a large, complex example including more than two federates. Live components and human interactions are also considered.

REFERENCES

Data Modeling and Simulation Office (DMSO). 1996. HLA Time Management Design Document Version 1.0, August

Data Modeling and Simulation Office (DMSO). 1997. High Level Architecture Run-Time Infrastructure Programmer's Guide Version 1.0, May

Department of Defense (DoD), USA. 1995. Modeling and Simulation (M&S) Master Plan, October

Hennessy, J. L. and D. A. Patterson. 1990. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc.

Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1996a. Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time

Warp Globally. *TRANSACTIONS of The Society for Computer Simulation* 13(3): 135-154.

Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1997. Ordering of simultaneous events in distributed DEVS simulation. *Simulation Practice and Theory* 5(3): 253-268.

Kim, Y. J. and T. G. Kim. 1996b. A Heterogeneous Distributed Simulation Framework Based on DEVS Formalism. In *Proceedings of the Sixth Annual Conference On Artificial Intelligence, Simulation and Planning in High Autonomy Systems*, La Jolla, California, USA, 116-121.

Kim, Y. J. and T. G. Kim. 1998. A Circuit Theoretic Approach to Protocol Conversion. (in preparation.)

Schwetman, H. 1988. Using CSIM to model complex systems. In *Proceedings of the 1988 Winter Simulation Conference*, San Diego, California, USA, 246-253.

Zeigler, B. P. 1984. *Multifacetted Modeling and Discrete Event Simulation*. Academic Press Inc.

AUTHOR BIOGRAPHIES

YONG JAE KIM is a Ph.D. candidate in the Department of Electrical Engineering at Korea Advanced Institute of Science and Technology (KAIST). He received a B.S. degree in electrical engineering from Yonsei University, Korea, and a M.S. degree in electrical engineering from KAIST.

TAG GON KIM is a professor in the Department of Electrical Engineering at KAIST. He received B.S. and M.S. degrees in electronics engineering from Pusan National University, Korea, and Kyungpook National University, Korea, respectively. He received a Ph.D. degree in computer engineering from the University of Arizona, Tucson, AZ. He is a senior member of IEEE, and a member of ACM, AAAI, SCS, and ETA Kappa Nu. He is an associate editor of *Simulation* and *TRANSACTIONS of SCS*.