A Heterogeneous Distributed Simulation Framework Based on DEVS Formalism

Yong Jae Kim and Tag Gon Kim

Dept. of Electrical Engineering
Korea Advanced Institute of Science and Technology
373-1, Kusong-Dong, Yusong-Gu, Taejon, 305-701, KOREA
yjkim@coregate.kaist.ac.kr
tkim@eekaist.kaist.ac.kr

Abstract

This paper proposes a heterogeneous distributed simulation framework based on the DEVS formalism. A software bus called the DEVS bus is proposed, which virtually connects the DEVS models and conventional non-DEVS models developed by different simulation languages such as SIMAN, SLAM, SIMSCRIPT, and so on. For the DEVS bus protocol, the hierarchical simulation algorithm proposed by Zeigler is used. For communicating between DEVS models and non-DEVS models connected on the DEVS bus, a protocol converter is proposed. The converter is realized by transformation of non-DEVS models into an equivalent DEVS models at the I/O level.

1 Introduction

Parallel and distributed discrete event simulation(PDES) [1] has been widely studied as a promising technology. PDES has been mainly concentrated on how to achieve reasonable speedup while guaranteeing that events are processed in chronological order by using a synchronization algorithm. PDES has been conventionally developed in homogeneous simulation environments. So, there is no opportunity to use models developed by different simulation languages.

We propose a heterogeneous distributed simulation framework based on the DEVS formalism [2]. Because of its great expressive power to represent discrete event systems, the DEVS formalism is selected as a basic structure of the framework. The DEVSim++ environment, a realization of the DEVS formalism in C++ [3], is used to develop the supervisory simulation model. As node simulation models, we use simulation models developed by different simulation languages such as SIMAN [4], SLAM [5], SIMSCRIPT [6] , and so on. The proposed framework makes it possible for models developed by different simulation languages to communicate each other.

In Distributed Interactive Simulation [7], each simulator communicates with others through passing Protocol Data Units predefined for the military domain. Unlike DIS, our framework is general enough to be applicable to several domains.

2 The Proposed Framework

The framework consists of a supervisory simulation model, node simulation models, and the DEVS bus system. The overall system architecture of the framework is shown in Figure 1. We use the DEVSim++ environment to represent the supervisory simulation model, because the environment allows us to model and simulate various target systems in a general and convenient manner. Widely available conventional simulation models such as SIMAN, SLAM, and SIMSCRIPT models, can be used as node simulation models. A node consists of a protocol converter and a node simulation model. The DEVS bus system is developed by using the concept of the hierarchical simulation algorithm proposed by Zeigler [2].

To simulate the whole simulation model in a heterogeneous distributed environment, the first step is to partition the model into a supervisory model and node models. After partitioning, client models are assigned to the supervisory simulation model and server models are allocated to node simulation models. A node simulation model acts as a server to process requests from the supervisory simulation model.

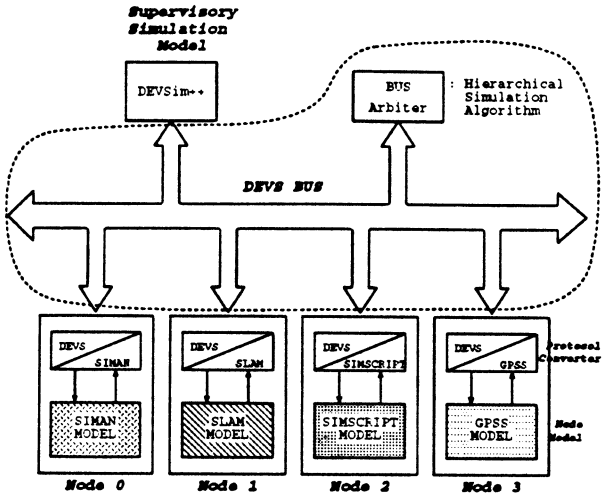

Figure 1: Overall System Architecture

The DEVS software bus system virtually connects the supervisory simulation model and node simulation models. We use the DEVS formalism and its associated abstract hierarchical simulation algorithm. The root-coordinator of the hierarchical simulation algorithm can be regarded as the bus arbiter of the DEVS bus system.

As described above, the supervisory simulation model acts as a client and node simulation models act as servers. When the supervisory simulation model sends a request message which is one of the (*, t) and (x, t) message, a node simulation model sends a result message as a reaction of the request. So, we can think node simulation models as resources and the supervisory simulation model as a processor.

Note that in our simulation scheme a node simulation model cannot send messages to the other node simulation models directly. Instead the message should first be sent to the supervisory simulation model. Then the supervisory simulation model routes the message to the destination model. Moreover, the bus system can multicast a message to several node simulation models at a time as a hardware bus system does.

The purpose of the framework is to support simulation of DEVS models together with models in conventional simulation languages. However, simulation methodologies for the conventional simulation models differ from that of the DEVS models. To overcome such difference may require a protocol conversion mechanism. The components DEVS/SIMAN, DEVS/SLAM, DEVS/SIMSCRIPT, and DEVS/GPSS shown in Figure 1 are protocol converters. These converters translate DEVS requests into messages which the specified simulation model can understand. For example, DEVS/SIMAN converter translates DEVS requests into SIMAN messages and vice versa. These converters are defined by an equivalent I/O transformation of node simulation models into DEVS models.

3 Hierarchical Simulation Algorithm of DEVS Models

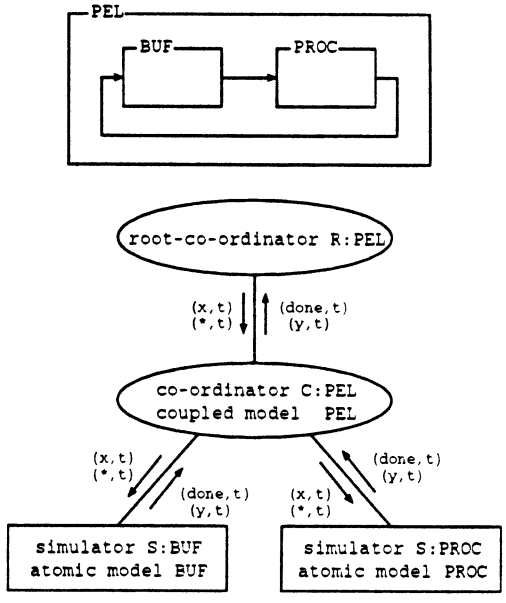

Figure 2: Hierarchical Simulation Algorithm

The hierarchical simulation algorithm for the coupled model, PEL, which has two atomic models, BUF and PROC [8], is shown in Figure 2. Attached to each DEVS model is an associated abstract simulator, either a *simulator* or a *coordinator*. Two atomic models, BUF and PROC, have associated *simulators* of S:BUF 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. These abstract simulators can be thought as virtual processors.

Assume that simulation starts by generating an event from BUF at $t = 0$. To do so, the initial values of the next event times (t_Ns) for the simulators S:BUF and S:PROC are set to zero and infinity, respectively. Once t_Ns for S:BUF and S:PROC are so initialized, the t_N of C:PEL (t_N(C:PEL)) is set to the minimum of the t_N of two component simulators.

When t_N(C:PEL) is initialized to zero, C:PEL sends a (done, t_N(C:PEL) = 0) message to R:PEL to inform that scheduling has been done. Once R:PEL receives the done message, it sends a (*, t = 0) message to C:PEL. Once C:PEL receives the message, it routes the message to its component, whose t_N is the same as zero. In this case, C:PEL routes the (*, t = 0) message to S:BUF. S:BUF requests BUF to produce an output and execute its internal transition function followed by its time advance function. After such a request, S:BUF updates its t_N based on the BUF's time advance function. S:BUF now transmits the BUF's output message, (y, t = 0), to C:PEL. When C:PEL receives the message, it translates the message in an input message of (x, t = 0) and sends it to S:PROC. Because the message is an external one, S:PROC requests PROC to execute the PROC's external transition function followed by its time advance function. S:PROC updates its t_N, which, in turn, updates t_N(C:PEL). When R:PEL receives the (done, t_N(C:PEL)) message from C:PEL, R:PEL generates another (*, t = t_N(C:PEL)) message, which C:PEL routes either to S:BUF or to S:PROC depending on their t_Ns.

To send messages to the destination model correctly, the supervisory simulation model must have the global view of the overall simulation model. This is obtained by the hierarchical simulation algorithm because the root-coordinator knows where the other coordinators are located.

We call the hierarchical simulation algorithm *the DEVS bus protocol.* In a common hardware bus, when a processor wants to access a resource, the processor should first send a bus request signal to a bus arbiter. Then the arbiter sends a bus grant signal to the processor if there is no conflict. After receiving the bus grant, the processor can access the resource. In the DEVS bus system, we can think the root-coordinator as a bus arbiter.

When a node model wants to send a message to another node model, it should send a (done, t_N) message to the root-coordinator. When the simulation time of the system is t_N, the root-coordinator sends a (*, t_N) message to the node model. Then, the node model sends a (y, t_N) message to the root-coordinator which routes the message to the destination node model.

4 Protocol Conversion

The proposed framework supports the reuse of conventional simulation models already developed by different simulation languages as node simulation models. When the node simulation models are integrated with the supervisory simulation model on the DEVS bus system, the mismatch between simulation protocols have to be conquered. This is because the node simulation models do not use the hierarchical simulation algorithm explained in the previous section.

The DEVS bus system uses four kinds of messages, (*, t), (x, t), (done, t) and (y, t) message. If an atomic DEVS model receives an external message, (x, t), it cannot immediately send a result message to its influencees. Only after receiving an (*, t) message from the root-coordinator, the bus arbiter of the DEVS bus system, the atomic model can send an output message by executing its output function.

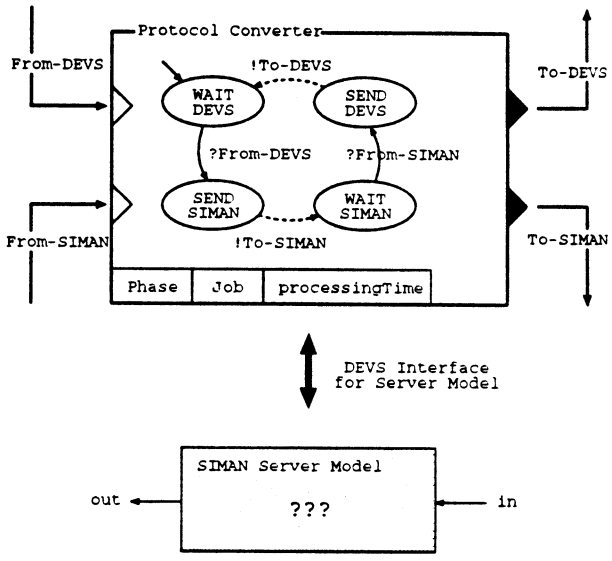

Figure 3: Protocol Converter

However, in conventional node simulation models such as SIMAN, SLAM, and SIMSCRIPT, there is no time advance function and no classification of messages in an explicit form. To join the node simulation models to the DEVS bus system, we propose a simple protocol conversion method. The protocol converter has the same I/O structure of the original simulation model but different internal structure.

Consider a SIMAN simulation model which acts as a server for a job. The model receives a job, processes it, and finally outputs a result. A protocol converter for the SIMAN server model is depicted in Figure 3. With the help of the protocol converter, a simulation protocol of a node simulation model is converted into the DEVS bus protocol.

This simple method works as follows. When a protocol converter receives an (x, t_1) message from

the From-DEVS port at the *WAIT_DEVS* phase, it executes an external state transition function which makes the model's phase *SEND_SIMAN*. It routes the message to the SIMAN model within the same node through the To-SIMAN port and waits a result from the SIMAN model at the *WAIT_SIMAN* phase. The SIMAN model processes the message which comes from the protocol converter. After processing the message, for a sojourn time t_{proc}, it sends a result to the From-SIMAN port of the protocol converter. Then the protocol converter saves the result and the time as its state variable and changes its phase into *SEND_DEVS*. As a result of this external state transition, the protocol converter sends a (done, t_N) message to its parent. In this case, the next scheduling time of the protocol converter, t_N, is $t_1 + t_{proc}$. When the protocol converter at the *SEND_DEVS* phase receives an $(*, t_N)$ message from the From-DEVS port, the protocol converter sends the result which was saved as the state variable to its parent.

The protocol conversion mechanism of other simulation models such as SLAM or SIMSCRIPT, is the same as the SIMAN model. Therefore, the simulation protocol of a SIMAN model is easily converted into that of the DEVS bus system.

To route messages from a protocol converter to a node simulation model, several mechanisms can be possible. For example, interprocessor communication, event handling with user written simulation code, and file I/O, may be possible. Selecting one of the schemes is an implementation issue. The user code option provides the flexibility to develop discrete event models. For example, the SIMAN language allows the user to include lower-level-language functions and routines in the simulation model.

5 Modeling Example : A Simple Manufacturing System

Consider a simple manufacturing system depicted in Figure 4. The system consists of a flexible manufacturing cell (FMC), an inspection station, a painting station, and a packaging station.

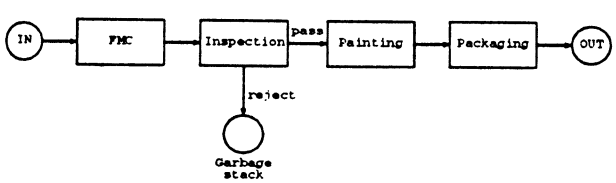

Figure 4: Simple Manufacturing System

The FMC consists of 10 horizontal milling machines which can perform any of three operations. Five of the milling machines are dedicated to perform operation A, one of the machines is dedicated to perform operation B and two of the machines are dedicated to perform operation C. Two of the 10 milling machines are classified as flexible. The inspection station tests parts which are transferred from the FMC. If the parts are found to be functioning improperly, they are routed to a garbage stack. After the parts are completely processed at the inspection station, they are transferred to the painting station and then they are packed at the packaging station. Finally, they are transferred to a staging area where they exit the system. Figure 5 is a DEVS representation of the target system with no experimental frame.

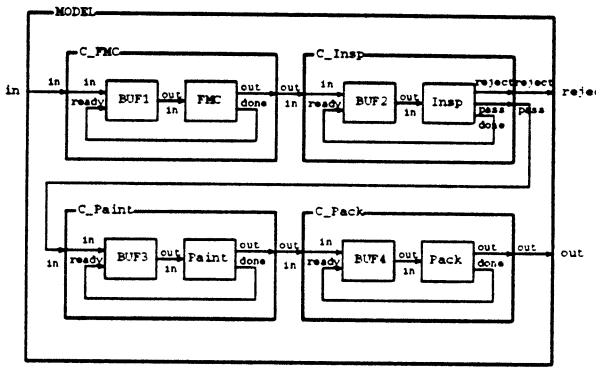

Figure 5: DEVS Model For Simple Manufacturing System

Assume that we have a FMC model and an inspection station simulation model already developed in SLAM II and a painting station model and a packaging station model in SIMAN IV , respectively. Then our framework proposed in this paper reuses these simulation models in the following manner.

First, the whole simulation model is partitioned into node models and a supervisory model. Let us allocate each station coupled model to a node and an experimental frame to the supervisor.

Then, using the protocol conversion method, a process atomic model of each node simulation model is converted into a DEVS model which has the same I/O structure as the original process atomic model.

As a result of these two steps, we can partition the whole simulation model as shown in Figure 6. The other models, BUF, GEN, and TRANSD, are implemented on the DEVSim++ environment.

A protocol converter for each station model can be easily developed. In this example, protocol convert-

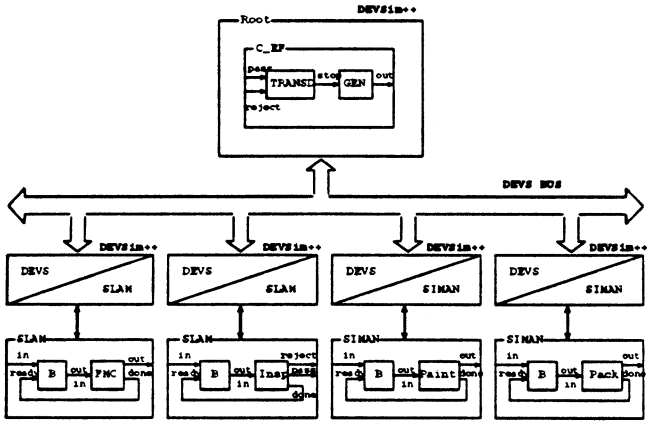

Figure 6: Model Partition

ers as protocol converters for the FMC, the painting station, and the packaging station should be the same as one in Figure 3 because these station models have the same I/O structure. The inspection station, however, has an input port and three output ports. The protocol converter for the station can be developed as shown in Figure 7. A pseudo DEVSim++ code for the protocol converter is shown in Figure 8.

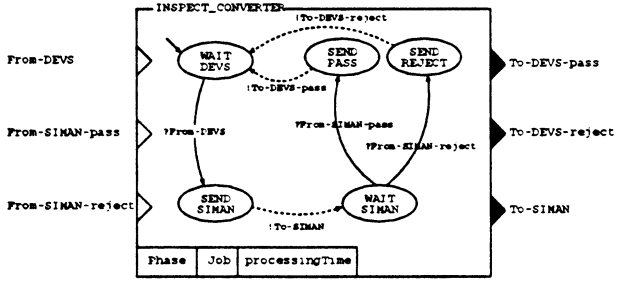

Figure 7: Protocol Converter For Inspect Station

6 Summary

We propose a heterogeneous distributed simulation environment based on DEVS formalism. To reuse models developed by different simulation languages, we propose a simple simulation protocol conversion method between simulation methodologies for the conventional simulation models and that of the DEVS models. The protocol converter is implemented on the DEVSim++ environment. The supervisory simulation model and conventional simulation models are connected on the DEVS bus system.

The protocol conversion method suggested in this paper is very simple. The method will be formally improved in the future. Currently, we are developing the overall system architecture on Unix PVM and Windows PVM environment [9] [10].

```
// external transition function
void INSPECT_CONVERTER_ext_transfn(State_vars& s,
    const timeType& e, const Messages& message)
{
  switch(*message.get_port()) {
    case From-DEVS :
      s.set_value("Phase", SEND_SIMAN);
      break;
    case From-SIMAN-pass :
      s.set_value("Job", (Job&)*message.get_job());
      s.set_value("processingTime",
          (timeType&)*message.get_time());
      s.set_value("Phase", SEND_PASS);
      break;
    case From-SIMAN-reject :
      s.set_value("Job", (Job&)*message.get_job());
      s.set_value("processingTime",
          (timeType&)*message.get_time());
      s.set_value("Phase", SEND_REJECT);
      break;
    default :
      ERROR;
  }
}

// internal transition function
void INSPECT_CONVERTER_int_transfn(State_vars& s)
{
  switch(s.get_value("Phase")) {
    case SEND_SIMAN :
      s.set_value("Phase", WAIT_SIMAN); break;
    case SEND_PASS :
    case SEND_REJECT :
      s.set_value("Phase", SEND_DONE);  break;
    default :
      ERROR;
  }
}

// output function
void INSPECT_CONVERTER_outputfn(const State_vars& s,
    Messages& message)
{
  switch(s.get_value("Phase")) {
    case SEND_SIMAN :
      message.set_port_val("To-SIMAN", job);
      break;
    case SEND_PASS :
      message.set_port_val("To-DEVS-pass", job);
      break;
    case SEND_REJECT :
      message.set_port_val("To-DEVS-reject", job);
      break;
    case SEND_DONE :
      message.set_port_val("To-DEVS-done", dummy);
      break;
    default :
      ERROR;
  }
}

// time advance function
void INSPECT_CONVERTER_time_advancefn(const State_vars& s)
{
  switch(s.get_value("Phase")) {
    case SEND_SIMAN :
    case SEND_DONE :
      return 0;
    case SEND_PASS :
    case SEND_REJECT :
      return s.get_value("processingTime");
    case WAIT_DEVS :
    case WAIT_SIMAN :
      return Infinity;
    default :
      ERROR;
  }
}
```

Figure 8: Pseudo DEVSim++ Code Of Protocol Converter For Inspect Station

References

[1] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of ACM.*, Vol. 33, No. 10, October, pp. 30-53, 1990.

[2] B. P. Zeigler, "Multifacetted Modelling and Discrete Event Simulation," Academic Press, Orlando, FL, 1984.

[3] T. G. Kim and S. B. Park, "The DEVS formalism: Hierarchical modular system specification in C++," *Proc. 1992 European Simulation Multiconf.*, York, UK, pp. 152-156, 1992.

[4] C. Dennis Pegden, Robert E. Shannon, and Randall P. Sadowski, "Introduction to Simulation Using SIMAN," McGraw-Hill, Inc., 1990.

[5] A. Alan B. Pristker, "Introduction to Simulation and SLAM II," 3rd Edition, Halsted Press Book, 1986.

[6] Edward C. Russell, "Building Simulation Models with SIMSCRIPT II.5," C.A.C.I., 1983.

[7] "Standard for Distributed Interactive Simulation - Application Protocols," IST-CR-94-50, Institute for Simulation and Traning, 1994.

[8] T. G. Kim, "DEVS formalism: Reusable Model Specification in an Object-Oriented Framework," International Journal in Computer Simulation 5, pp. 397-416, 1995.

[9] V. S. Sunderam, G. A. Geist, and J. Dongarra, "The PVM concurrent computing system: Evolution, Experiences, and Trends," Parallel Computing 20, pp. 531-545, 1994.

[10] Alexandre Alves, Luis Silva, Joao Carreira, and Joao Gabriel Silva, "WPVM: Parallel Computing for the People," Proceedings of HPCN'95, High Performance Computing and Networking Conference, in Springer Verlag Lecture Notes in CS., pp. 582-587, Milan, Italy 1995.