DEVS Formalism: Reusable Model Specification in an Object-Oriented Framework

Tag Gon Kim
Korea Advanced Institute of Science and Technology

The Discrete Event Systems Specification (DEVS) formalism developed by Zeigler supports specification of discrete-event models in hierarchical, modular form. Such hierarchical, modular specification offers a basis for the reusable model base, thereby enhancing productivity and quality in modeling engineering. The DEVS-Scheme implements the DEVS formalism in a LISP-based object-oriented programming environment that supports development of highly reusable model bases by using the DEVS formalism and object-oriented framework. This article describes a methodology for developing reusable models in the DEVS-Scheme environment. To develop highly reusable models in a hierarchical, modular manner, the methodology exploits inheritance and encapsulation from the object-oriented paradigm and the hierarchical, modular framework from the DEVS formalism.

---

DEVS formalism, object-oriented simulation, reusable model, multiprocessor system model, hierarchical simulation, DEVS-Scheme

1 INTRODUCTION

It is accepted widely that the repetitive nature of software construction is indeed striking. Software reuse concerns the reapplication of a variety of knowledge about one system to other similar systems. The objective of such reuse is to reduce the effort of development and maintenance of other systems, thereby improving software development productivity and quality.

Developing simulation models is a software development process in that models eventually are to be implemented as working computer programs. In this sense, model specification is software specification, and model verification is software debugging. As recognized in [1], for a long time simulation has employed concepts of software reusability in the model development life cycle.

There are two approaches to software reusability, depending on the nature of the components being reused [2]. One approach is to apply composition technologies; the other is to apply generation technologies. The principle of model reuse applies more composition technologies than generation ones. In composition technologies, the model components to be reused are largely atomic, which are passive elements operated upon by an external agent. However, to achieve high reusability, building blocks can be complex, composition models from which more complex models are constructed.

Correspondence and requests for reprints should be sent to Tag Gon Kim, Department of Electrical Engineering, KAIST, 373-1 Kusong-dong, Yusong-gu, Taejon 305 701, Korea. E-mail: (tkim@ee.kaist.ac.kr/)
The Discrete Event Systems Specification (DEVS) formalism, developed by Zeigler, supports specification of discrete-event models in a hierarchical, modular manner [3–5]. Within the formalism, two classes of models, namely, atomic and coupled models, are to be specified. An atomic DEVS model is one that cannot be decomposed into components, whereas a coupled one can be decomposed into component models. The formalism specifies an atomic DEVS model in terms of timed state transition. A coupled DEVS model is specified in terms of specification for each component and connection information between components and the coupled one, and among components.

The discrete-event systems world view considers a system as a collection of entities and relationships between entities. Such a world view is recognized as being compatible with the object-oriented world view. Compatibility between the two views motivates development of a discrete-event simulation system in an object-oriented programming environment. Indeed, the DEVS-Scheme is one such environment that implements the DEVS formalism in a LISP-based, object-oriented environment [6–8]. The environment supports development of highly reusable discrete-event models specified by the DEVS formalism within the object-oriented programming system on Scheme [9].

This article describes a methodology for developing reusable models in the DEVS-Scheme environment. To develop highly reusable models in a hierarchical, modular manner, the methodology exploits inheritance and polymorphism from the object-oriented paradigm, and the hierarchical, modular framework from the DEVS formalism. Thus, the methodology reuses functions to develop atomic models by exploiting inheritance and encapsulation from the object-oriented paradigm and reuses model components to construct complex hierarchical models supported by the DEVS formalism. Such a level of reusability cannot be achieved by simply employing object-oriented programming techniques without using hierarchical, modular specification semantics, such as the DEVS formalism, in model development. Moreover, models developed in a hierarchical, modular form need a hierarchical scheduling algorithm for simulation, as shown in Section 7.

This article is organized as follows. Section 2 presents a brief review of the object-oriented modeling and simulation concepts. Section 3 presents the model base concepts. Section 4 describes the DEVS formalism and its realization in an object-oriented environment. Sections 5 and 6 show model reusability at the atomic model level and the coupled model level, respectively. Hierarchical simulation in the DEVS-Scheme is explained in Section 7, and concluding remarks are given in Section 8.

2 OBJECT-ORIENTED MODELING AND SIMULATION

The object-oriented programming paradigm, originated from the discrete-event simulation language SIMULA [10], is a technique in which a software system is decomposed into subsystems based on objects. In such a paradigm, computation is done by objects sending messages among themselves. The object-oriented programming technique enhances software quality by supporting maintainability, extensibility, and reusability [11].
In a real-world system, it is natural to think that components are distributed throughout the system and interact with one another. Accordingly, a simulated model of such a system would have distributed components, each of which may be a self-contained unit working concurrently. It follows that conventional programming techniques are not well suited to modeling and simulation of such a system. Thus object-oriented programming provides a natural means of representing knowledge in a form suited for simulation modeling.

In object-oriented simulation modeling, a simulation model consists of objects, called components, which are connected to form a composite model. We consider that each real component has a set of states that changes over time when components interact with each other. Object models of these components can also have state variables represented by instant variables whose values are changed when messages are communicated and methods are applied. This sets up a correspondence between real components, their states, and their interactions on one hand; and object counterparts, state variables, and message-passing communication on the other. Such a correspondence is natural provided that the real system interactions can be represented conveniently by message communication.

If simulation implements a form of discrete-event simulation, state changes are brought about as events scheduled on a time base. In our object-oriented simulation approach, each model object has an associated object, called an abstract simulator, which interprets the model's dynamics and manages simulation time. This idea has been implemented in the DEVS-Scheme environment for discrete-event systems modeling and simulation described in Section 4.2. Whenever a modeler creates a model object in the DEVS-Scheme environment, an object for an associated abstract simulator is created implicitly and attached to each other by the environment. Thus, the connection between abstract simulators in the DEVS-Scheme is the same as that for models. Simulation in the DEVS-Scheme precedes by means of message-passing between abstract simulators.

3 MODEL BASE CONCEPTS

3.1 Modular System and Model Base

A module in a software system is a program text that can function as a self-contained, independent unit addressing only a single, logically coherent task. Similar to a circuit board, such a module unit should be totally independent of the source of input, the destination of output, and the history of activation of the module. The important reason for keeping modules independent has to do with local modification and efficient testing of each module with the software system.

The model base is an organized library that contains a set of reusable models in a modular form. Models in a model base can be either atomic or coupled. New models can be saved in, and saved models can be retrieved from, the model base. Models so retrieved may be reused to construct a hierarchical model that can be reused as a component model in construction of yet more complex, hierarchical models. Again, the complex, hierarchical model is saved in the model base and can be reused later, and so on.
Figure 1. Coupling scheme and model base concepts.

Figure 1 shows the fundamental concepts of modularity and model base. Assume that two atomic models, BUF (buffer) and PROC (processor), are developed and saved in the model base of Figure 1(a). If these model descriptions are in a proper modular form, we can create a new model PEL (processing element) by assembling BUF and PROC by using an operation called coupling (see Section 3.2). The resulting model PEL shown in Figure 1(b) is a coupled model, which is also in modular form. As can be seen, the term modularity, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated.

Note that in the coupled model, PEL, BUF, and PROC do not know how they are coupled; only PEL knows coupling information of its components. Representing coupling information in such a way allows modelers to construct a complex model in a hierarchical manner. Once placed in the model base in Figure 1(c), PEL can itself be employed to construct yet larger models in the same manner used with its component models. This property, called closure under coupling, enables hierarchical construction of modular models.

3.2 Specification of Coupling Scheme

We now describe coupling specification to construct a hierarchical model by using reusable component models in a model base. The coupling scheme (CS) is specified by a set of three relations—external input coupling (EIC), external output coupling (EOC), and internal coupling (IC)—each of which is represented by a set of ordered
pairs of ports. Formally, an ordered pair of ports of the form (M1.p1, M2.p2) means that the output port p1 of model M1 (M1.p1) is connected to the input port p2 of model M2 (M2.p2). In this specification, “M1.p1 is connected to M2.p2” means that the information flows only from M1.p1 to M2.p2. Thus, the coupling scheme of any model can be represented by the collection of three relations, namely, CS = (EIC, EOC, IC).

External input coupling is the relation of the input ports of the coupled model to those of the component models. It indicates how the input ports of the composite model are connected to the input ports of the components. External output coupling is the relation of the output ports of the coupled model to those of the component models. It specifies how the output ports of the composite model are connected to the output ports of the component models. Finally, internal coupling is the relation of the output ports of the components to the input ports of other components. It designates how the components inside the coupled model are interconnected by indicating how the output ports of some components are connected to input ports of other components. Figure 1(d) shows a coupling scheme of the model PEL.

4 DEVS FORMALISM: A BRIEF REVIEW

4.1 DEVS Formalism and Abstract Simulator

A set-theoretic formalism, the DEVS formalism [4, 5], specifies discrete-event models in a hierarchical, modular form. Within the formalism, one must specify (1) the basic models from which larger ones are built, and (2) how these models are connected together in hierarchical fashion. A basic model, called an atomic model (or atomic DEVS), has specification for dynamics of the model. An atomic model M is specified as follows [5]:

\[ M = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \tau a \rangle \]

\[ X = \text{input events set} \]
\[ S = \text{sequential states set} \]
\[ Y = \text{output events set} \]
\[ \delta_{\text{int}} = \text{internal transition function: } S \rightarrow S \]
\[ \delta_{\text{ext}} = \text{external transition function: } Q \times X \rightarrow S \]
\[ Q = \text{total state of } M = \{ (s, e) | s \in S, 0 \leq e \leq \tau a(s) \} \]
\[ \lambda = \text{output function: } S \rightarrow Y \]
\[ \tau a = \text{time advance function: } S \rightarrow \text{Real} \]

We call the internal transition function, external transition function, time advance function, and output function the DEVS characteristic functions. Note that the DEVS semantics requires modelers to specify the characteristic functions in a separated form. As we shall see later, such separated specification exploits reusability of the characteristic functions through inheritance in our object-oriented simulation environment of the DEVS-Scheme.

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

\[
DN = <D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, \text{SELECT}>
\]

\( D \) = component names set;
for each \( i \) in \( D \),
\( M_i \) = DEVS for component \( i \) in \( D \);
\( I_i \) = set of influences of \( i \);
for each \( j \) in \( I_i \),
\( Z_{i,j} = Y_i \rightarrow X_j : i\text{-}to\text{-}j output translation function; \)
SELECT = subsets of \( D \rightarrow D : \) tie-breaking selector.

Detailed descriptions for the definitions of the atomic and coupled DEVS can be found in [5].

The abstract simulator introduced in [5] is a conceptual device capable of interpreting the dynamics of DEVS models. Two classes of the abstract simulator have been defined: one for the atomic DEVS and the other for the coupled DEVS. Within the DEVS-Scheme, a pair of a model and an abstract simulator is created, and association of the two is made for simulation.

The job of an abstract simulator for an atomic model is to schedule the next event time and execute the model's transition functions and output function timely as simulation proceeds. The responsibilities of the abstract simulator for a coupled DEVS model are to synchronize the component abstract simulators for scheduling the next event time and to route external event messages to component simulators. The complete algorithms for both abstract simulators and proof for the correctness for the algorithms can be found in [3]. The architectures and performance of distributed simulation systems, derived from the abstract simulator concept, have been studied intensively in [12] and some were implemented by multiprocessor computer systems [13, 14].

4.2 DEVS-Scheme Environment

The DEVS-Scheme is a LISP-based, object-oriented simulation environment that realizes the DEVS formalism and its associated abstract simulator concepts. To realize them, first, the DEVS-Scheme defines two general classes: models for DEVS models and processors for abstract simulators. Such classes are defined as the subclasses of a universal class called entities. The class entities provides tools—such as constructor and destructor—to manipulate objects not only for the class itself but also for the two subclasses defined earlier. The class models has two subclasses to realize two model classes defined in the DEVS formalism. The two subclasses are atomic-models realizing atomic DEVS models and coupled-models realizing coupled DEVS models.

The class atomic-models realizes the atomic level of the DEVS formalism by means of its class/instance variables and methods. Four instance variables for modelers to specify are \( S \), int-transfn, ext-transfn, and outputfn, which correspond to the sequential state, internal transition function, external transition function,
and output function of an atomic DEVS, respectively. The class *coupled-models* realizes the *coupled DEVS* which embodies the hierarchical model composition of the DEVS formalism. *Coupled-models* has a specification for its component models (also called children) and coupling scheme. Instance variables corresponding to children and coupling relations, and methods that manipulate the variables, realize the formalism.

The class *processors* realizing the abstract simulator concepts is specialized into three classes: *simulators*, *coordinators*, and *root-coordinators*. The *simulators* and *coordinators* are assigned to handle *atomic-models* and *coupled-models* in a one-to-one manner. The *model-processor* pairing is recorded by instance variables of models and processors; processors have an instance variable, *devs-component*, and models have an instance variable, *processor*. A *root-coordinator* manages the overall simulation and is linked to a *coordinator* of the outmost coupled model. Figure 2 shows the class hierarchy in the DEVS-Scheme. Details for the DEVS-Scheme environment can be found in [7] and [8].

## 5 ATOMIC MODEL DEVELOPMENT

### 5.1 Atomic Models in DEVS-Scheme

The DEVS-Scheme allows modelers to develop discrete-event models in a hierarchical, modular manner by using the DEVS formalism within an object-oriented environment. Thus, the DEVS-Scheme supports model reusability by exploiting both the object-oriented paradigm and the DEVS semantics. This section describes model reusability in the atomic model development within the DEVS-Scheme.

Figure 3 shows the structure for the atomic-models class. As shown, the atomic-
models class has instance variables $S$, int-transfn, ext-transfn, and outfn. These instance variables correspond to the state set, the internal transition function, the external transition function, and the output function in the DEVS formalism, respectively. Methods in the DEVS-Scheme to manipulate these variables are set-$S$, set-ext-transfn, set-int-transfn, set-outputfn, and others [7]. Developing an atomic model in the DEVS-Scheme is to define $S$, int-transfn, ext-transfn, and outfn for the model.

5.2 Specialization of the Class Atomic-Models

Consider that we develop classes of buffer models that employ different service disciplines. For simplicity, consider development of two specialized classes of buffer models, namely, the fifo (first-in–first-out) class and the lifo (last-in–first-out) class. Call the fifo class FIFO and the lifo class LIFO. Because there are some common operational properties in the two classes, such properties can be shared in the object-oriented paradigm. For that, first we define a class called BUFFERS, which has such common properties. Then we develop two specialized classes of FIFO and LIFO based on the class BUFFERS. Figure 4 shows the class hierarchy in the DEVS-Scheme environment for such development.

Let us outline the DEVS specification for a buffer model, an object of the class BUFFERS. The buffer model BUF, as shown in the model base in Figure 1(a), is commonly used in computers and/or communication systems. It controls flow of incoming problems or packets that are to be sent to a cascaded processor or transmitter. To specify the model at an atomic level we need to define four
Figure 4. Specialization of atomic-models.

```
;; Definition of the class BUFFERS in DEVS-Scheme
;; superclass: atomic-models
;; We only define the internal transition function and output function.
;; The external transition function is left undefined.
;; input ports: "in", "ready"
;; output port: "out"
;; state variables: sigma, phase, waiting-line, proc-free

(define-class BUFFERS
  (classvars
   (instvars
    (ind-vars '(sigma phase waiting-line proc-free)) ;; state variables
    (int-transfn buf_intrans) ;; internal transition function
    (outputfn buf_outfn) ;; output function
   ) ;; superclass
  )
  (mixins atomic-models)
  (options gettable-variables
   settable-variables
   inittable-variables
  )
)
(compile-class BUFFERS)
```

Figure 5. Definition of class BUFFERS.

DEVS characteristic functions: external transition function, internal transition function, output function, and time advance function. The external transition function specifies how a buffer model responds to the external stimuli. Such stimuli include jobs incoming from the outside world and signals from the cascaded processor acknowledging that it is free. The internal transition function specifies how a buffer model updates its waiting line containing such jobs when the cascaded processor is free after elapsing a period of time specified by the time advance function. Finally, the output function specifies how a job is sent to the cascaded processor just before its internal transition.

To find some common operations between fifo and lifo buffers, let us relate
the preceding DEVS characteristic functions to the well-known operations on a buffer, namely, insert, delete, and first. Clearly, the external transition function is related to the insert operation, the internal transition function to the delete operation, and the output function to the first operation. Note that the time advance function specifying sojourn time is embedded in the external and internal transition functions. Define the insert operation of a fifo buffer as insertion of an incoming job at the end of the waiting line and a lifo buffer at the first of the line. Then, the delete operations of both buffers are the same: deletion of the first job in the waiting line. In addition, the first operations for the two buffers are the same. Thus, the internal transition functions and output functions are the same for both classes. We shall define such functions in the class BUFFERS which can be reused for the classes FIFO and LIFOS through inheritance.

Based on the preceding discussion, we now develop the class BUFFERS in the DEVS-Scheme. First, we define BUFFERS as a subclass of atomic-models in the DEVS-Scheme. Next, we define the internal transition function and the output function, while leaving the external transition function undefined. Figure 5 shows such definition in the DEVS-Scheme environment.

5.3 Model Reuse Through Inheritance

Let us develop the classes FIFO and LIFOS by reusing the class BUFFERS defined previously. To do so, first we define the classes FIFO and LIFOS as subclasses of BUFFERS in the DEVS-Scheme. Such definition inherits the internal transition

```
;; Definition of the class FIFO in DEVS-Scheme
;;
;; superclass: BUFFERS
;; We only define the external transition function and output function.

(define-class FIFO
  (classvars)
  (instvars
   (ext-transf fifo_extras))
  (mixins BUFFERS)
  (options gettable-variables
   settable-variables
   inittable-variables)
  )
(compile-class FIFO)

;; external transition function ;;
(define (fifo_extras S e X)
  (case (content-port X)
    ((in)
     ;; when receive an input at port "in"
     (set! (state-waiting-line S) (append (state-waiting-line S) (list (content-value X))))
     (if (= (length (state-waiting-line S)) 1)
      (hold-in 'SEND sending-time)))
    ((ready
      ;; when receive an input at port "ready"
      (set! (state-proc-free S) #true)
      (if (state-waiting-line S)
       (hold-in 'SEND sending-time))
      (else (bkpt "no such a port exists: " S)))

Figure 6. Definition of class FIFO.
```
function and the output function from the superclass BUFFERS to its subclasses of FIFOS and LIFOS. We then define the external transition functions specific to FIFOS and LIFOS that were not defined in the class BUFFERS. Figure 6 shows the definition of the FIFOS class including its external transition function.

The procedure explained here can be applied recursively to the development of subclasses of FIFOS or LIFOS if needed. For example, a class of priority queues can be defined as a subclass of FIFOS. In such a case, we need to define only the DEVS characteristic functions for the subclass that are different from those defined in FIFOS. The functions defined in the subclass can override ones defined in its superclass.

The approach shown for developing atomic models is novel. The combination of the object-oriented paradigm and the DEVS semantics makes it possible to support the approach in atomic model development. Once atomic models are developed based on the preceding approach, such models can be saved in the model base for later reuse in developing coupled DEVS models. The following section describes how such models are to be reused as independent objects.

6 COUPLED MODEL DEVELOPMENT

6.1 Coupled Model Specification

A coupled DEVS model can be developed by using the class digraph-models defined in the DEVS-Scheme. Figure 7 shows the class definition for the digraph-models.

```
digraph-models

instance variables:
  children
  composition-tree
  influence-digraph
  priority-list
    ⬤

methods:
  specify-children
  add-couple
  set-priority
    ⬤

superclass: coupled-models
```

Figure 7. Class digraph-models in the DEVS-Scheme.
As with the atomic model development, users need to fill the slots of instance variables of the class.

In coupled model development, model reusability is achieved by exploiting hierarchical composition technologies supported by the DEVS formalism. Because object models in the model base are self-contained, it is possible to take them out from the model base and reuse them as components.

To explain such reusability in more detail, consider performance modeling of a bus-oriented multiprocessor system shown in Figure 8. The system consists of \( n \) processing elements, a single global bus, and a shared memory. The workload assumption for the system is as follows. Each processor initiates its local processing. After a random processing time, the processor issues a bus request and waits until connection to the shared memory is established. Once the connection is made, the processor accesses the shared memory in a random time. After the memory access is completed, the bus is released and the processor returns to its local processing.

Let us develop a simulation model of Figure 8. Assume that each processing element consists of a buffer and a processor. Then, we can use the model PEL, as shown in Figure 1(b), as a model of each processing element. Again, assume

![Diagram of Shared Memory Multiprocessor System](image)

**Figure 8.** Shared memory multiprocessor system.

```
; A Digraph Model PEL
; components: BUF, PROC
; input ports: "bus_connected", "prob_in"
; output port: "bus_request"

(make-pair digraph-models 'PEL) ;; (1)
(send PEL specify-children (list BUF PROC)) ;; (2)
(send PEL add-couple PEL BUF 'prob_in 'in) ;; (3)
(send PEL add-couple PEL PROC 'bus_connected 'smem_access) ;; (4)
(send PEL add-couple PROC PEL 'out 'bus_request) ;; (5)
(send PEL add-couple BUF PROC 'out 'in) ;; (6)
(send PEL add-couple PROC BUF 'done 'ready) ;; (7)
(send PEL set-priority (list BUF PROC)) ;; (8)
```

**Figure 9.** Digraph model PEL in the DEVS-Scheme.
that BUF and PROC are developed and saved in the model base. Then, the DEVS-Scheme codes in Figure 9 construct PEL.

In Figure 9, line (1) creates a coupled model PEL from the class digraph-models and associated coordinator C:PEL in the DEVS-Scheme. Line (2) specifies that PEL has BUF and PROC as components. Lines (3)–(5) specify the external input and external output coupling schemes. Lines (6) and (7) specify the internal coupling scheme. Finally, line (8) specifies a tie-breaking rule between components. As shown in the specification, all information concerning the coupled model construction is attached to PEL, not to BUF or PROC. This approach allows us to construct other coupled models using PEL as a component without referring to BUF or PROC. Again, we save PEL in the model base for later reuse.

6.2 Model Reuse Through Hierarchical Assembling

To complete the modeling of the multiprocessor system, let us call models of the overall system, a collection of processing elements, the bus, and the shared memory, MUL-PROC, br-PELs, BUS, and SMEM, respectively. Note that br-PELS is a coupled model, each element of which is the digraph model PEL.

Since all PELs in br-PELs are connected to BUS, the class broadcast-models in the DEVS-Scheme can be used to specify such a coupling scheme. The following DEVS-Scheme codes construct the broadcast model br-PELs shown in Figure 10.

(make-broadcast PELs) ;; create a broadcast model br-PELs
(send br-PELs make-members 'PEL n) ;; make n PELs for br-PELs

![Figure 10. Broadcast model of PELs.](image-url)
Note that no coupling scheme is specified in br-PELs. The DEVS-Scheme implicitly establishes a coupling between a broadcast model and its components. More specifically, the DEVS-Scheme connects each input port of a broadcast model to the input port of each element and each output port of its elements to an output port of the broadcast model.

Assume that we save br-PELs in the model base for later reuse, and that BUS and SMEM are in the model base. Then the overall model MUL-PROC can be constructed as a digraph model consisting of br-PELs, BUS, and SMEM. Figure 11 shows the model and its DEVS-Scheme codes. Note again that hierarchical assembling, supported by the DEVS-Scheme environment, can reuse models saved in the model base.

We argue that our approach of hierarchical model development would be impossible in other simulation languages or environments unless their simulation engines support hierarchical simulation based on the concepts of abstract simulators introduced in Section 4.1. To support the argument, the following section first examines two simulation strategies, namely, event-oriented and process-oriented approaches used in discrete-event simulation languages, conventional or object-oriented ones. It then explains our approach of hierarchical simulation based on abstract simulator concepts.

```
;;; An Overall Digraph Model MUL-PROC of A Multiprocessor System
;;; components: br-PEL, BUS, SMEM
;;; input ports: "user_request" 
;;; output port: none 

(make-pair digraph-models 'MUL_PROC)
(send MUL-PROC specify-children (list br-PELs BUS SMEM))
(send MUL-PROC add-couple MUL-PROC br-PELs 'user_request 'prob_in)
(send MUL-PROC add-couple br-PELs BUS 'bus_request 'in)
(send MUL-PROC add-couple BUS br-PELs 'out 'bus_connected)
(send MUL-PROC add-couple BUS SMEM 'out 'in)
(send MUL-PROC add-couple SMEM BUS 'out 'smem_done)
(send MUL-PROC set-priority (list SMEM BUS br-PELs))
```

Figure 11. Overall digraph model MUL-PROC of Figure 8.
7 HIERARCHICAL SIMULATION IN THE DEVS-SCHEME

7.1 Discrete-Event Simulation Strategies

Let us examine two typical discrete-event simulation strategies. In the event-oriented simulation strategy, an event list consisting of event routines and associated simulation times is maintained within the environment as a global variable in chronological order. An event routine is the description of a timed state transition and output specification for the model to be simulated. In this sense, an atomic DEVS model can be specified by an event routine [4]. Then, a collection of such models can form an event list. Simulation precedes by executing events in the list, one by one in order. However, as we shall show in the following, a hierarchically coupled DEVS model having components of either coupled or atomic models cannot be specified by an event routine. Thus, the event-oriented simulation strategy fails to simulate a hierarchically coupled DEVS model without changing its hierarchical structure to a nonhierarchical one.

In the process-oriented simulation strategy, a process is a sequence of related events, each separated by its own delay time. Because a model is represented by a process in this approach, models may not be reusable to construct a hierarchical model, hierarchical in the sense that a process has other processes as its components. Because of the same reason as in the event-oriented strategy, the process-oriented strategy cannot be employed to simulate a coupled DEVS model.

7.2 Hierarchical Simulation of DEVS Models

To explain our approach of hierarchical simulation, consider simulation of the coupled model of PEL explained earlier. Figure 12(a) shows a model to be simulated, and Figure 12(b) shows the architecture for hierarchical simulation used in the DEVS-Scheme. In Figure 12(a), GEN is a generator model that generates input events for simulation of the PEL model. Note that the overall simulation model is a coupled model MP that consists of an atomic model GEN and a coupled model PEL which, in turn, consists of two atomic models, BUF and PROC.

Recall that the hierarchical simulation in the DEVS-Scheme is based on concepts of abstract simulators associated with the DEVS formalism [5]. As shown in Figure 12(b), attached to each model is an associated abstract simulator, either a simulator or a coordinator. More specifically, three atomic models—GEN, BUF, and PROC—have associated simulators of S:GEN, S:BUF, and S:PROC, respectively. Two coupled models—PEL and MP—have associated coordinators of C:PEL and C:MP, respectively. Finally, R:MP is the root-coordinator whose job it is to manage the overall simulation clock. In the DEVS-Scheme, a macro “make-pair” creates a pair of a model and an associated abstract simulator. Let us explain the jobs of such abstract simulators in hierarchical simulation.

The job of a simulator is to schedule the next event time and execute the DEVS characteristic functions of the associated atomic model. The job of a coordinator is to route messages to component abstract simulators and synchronize them. To
perform such jobs, both a simulator and a coordinator use two input event types: \((x, t)\) and \((*, t)\). \((x, t)\) is an external input event arrived randomly, whereas \((*, t)\) is an internal one generated from the root-coordinator when the scheduled time arrives. Figures 13 and 14 show abstract simulator algorithms for atomic and coupled DEVS models, respectively.

Let us now explain how hierarchical simulation precedes in Figure 12(b). Assume that simulation starts by generating an event from GEN at \(t = 0\). To do so, the initial values of the next event times (\(tN\)s) for the simulators S:GEN, S:BUF, and S:PROC are set to zero, infinity, and infinity, respectively. Such initialization is done by the DEVS-Scheme, not by a modeler. However, the modeler must initialize the values of time advance functions at initial states for GEN, BUF, and PROC, accordingly. Once \(tN\)s for S:BUF and S:PROC are so initialized, the DEVS-Scheme sets the \(tN\) of C:PEL (\(tN(C:PEL)\)) to the minimum of the \(tN\)s of two component simulators. Thus, \(tN(C:PEL)\) is set to zero. Likewise, \(tN(C:MP)\) is set to the minimum of \(tN(S:GEN)\) and \(tN(C:PEL)\) which is zero. When \(tN(C:MP)\) is
when receive an input \((x, t)\)
if \(tL \leq t \leq tN\)
    request \(M_a\) to execute its external transition function
    update \(tL := t\)
    request \(M_a\) to execute its time advance function
    schedule \(tN\) based on the time advance function
    send \((\text{done}, tN)\) to its coordinator
else
    error "incorrect simulation time in external event"

when receive an input \((*, t)\)
if \(t = tN\)
    request \(M_a\) to execute its output function
    send the output \((y, t)\) produced by \(M_a\) to its coordinator
    request \(M_a\) to execute its internal transition function
    update \(tL := t\)
    request \(M_a\) to execute its time advance function
    schedule \(tN\) based on the time advance function
    send \((\text{done}, tN)\) to its coordinator
else
    error "incorrect simulation time in internal schedule"

Figure 13. Abstract simulator for atomic DEVS.

when receive an input \((x, t)\)
if \(tL \leq t \leq tN\)
    route \((x, t)\) to component simulators
    wait until all component simulators done
    update \(tL := t\)
    schedule \(tN\) as the minimum of components \(tN_i\)’s
    send \((\text{done}, tN)\) to its parent coordinator
else
    error "incorrect simulation time in external event"

when receive an input \((*, t)\)
if \(t = tN\)
    find all imminent simulators \(i\) such that \(tN_i = tN\)
    apply the priority rule and select one \(i^*\)
    send \((*, t)\) to \(i^*\)
    translate \((y, t)\) of \(i^*\) to \((x, t)\)
    send \((x, t)\) to all influences of \(i^*\)
    wait until all influences done
    update \(tL := t\)
    schedule \(tN\) as the minimum \(tN_i\)’s of \(i^*\) and all its influences
    send \((\text{done}, tN)\) to its parent coordinator
else
    error "incorrect simulation time in internal schedule"

Figure 14. Abstract simulator for coupled DEVS.

initialized to zero, \(C:MP\) sends a \((\text{done}, tN(C:MP) = 0)\) message to \(R:MP\) to inform that scheduling has been done. Once \(R:MP\) receives the done message, it sends a \((*, t = 0)\) message to \(C:MP\). Once \(C:MP\) receives the message, it routes the message to its component, whose \(tN\) is the same as zero. In this case, \(C:MP\) routes the \((*, t = 0)\) message to \(S:GEN\). \(S:GEN\), when it receives the message, it requests \(GEN\) to produce an output and execute its internal transition function
followed by its time advance function. After such a request, S:GEN updates its $tN$ based on the GEN's time advance function. S:GEN now transmits the GEN's output message, $(y, t = 0)$, to C:MP. When C:MP receives the message, it translates the message in an input message of $(x, t = 0)$ and sends it to C:PEL. When C:PEL receives the $(x, t = 0)$ message, it routes it to S:BUF. Because the message is an external one, S:BUF requests BUF to execute the BUF's external transition function followed by its time advance function. S:BUF updates its $tN$, which, in turn, updates $tN$(C:PEL). Then $tN$(C:MP) is set to the minimum of $tN$(C:PEL) and $tN$(S:GEN). When R:MP receives the (done, $tN$(C:MP)) message from C:MP, R:MP generates another $(*, t = tN$(C:MP)) message, which C:MP routes either to S:GEN or to C:PEL depending on their $tNs$. If the message is routed to C:PEL, it is traversed in a path of S:BUF, C:PEL, S:PROC, and C:PEL. In the meantime, S:BUF, S:PROC, and C:PEL update their $tNs$. Such updating, in turn, updates $tN$(C:MP) by which R:MP generates a $(*, tN$(C:MP)) message, and so on.

In our hierarchical scheduling approach, no information about scheduling for component simulators is known to the outside world. Only the coordinator of a coupled model knows the schedules for its component simulators. Because component schedules determine the parent coordinator's schedule recursively, scheduling is done in a bottom-up manner. Thus, it is clear that the scheduling approach explained here is totally different from those used in the event-oriented and process-oriented simulation. This explains why our model-based approach, supported by the hierarchical DEVS formalism and DEVS-Scheme, achieves more reusability of models than any other approach.

8 CONCLUDING REMARKS

The DEVS-Scheme environment supports development of highly reusable models in an object-oriented framework. Exploiting such reusability in development of complex models markedly enhances productivity in model engineering. The reusability is achieved by combining three methodologies: the model base, the DEVS formalism, and the object-oriented programming. Such a combination allows us to exploit model reusability through inheritance and encapsulation from the object-oriented paradigm and the composition technologies supported from the DEVS formalism and the model base methodology.

The hierarchical simulation algorithms based on abstract simulator architecture make it possible to simulate DEVS models developed in hierarchical, modular form. Such simulation management may not be employed in simulation languages, conventional or object-oriented. Based on the hierarchical simulation management, the DEVS-Scheme allows users to develop user-defined classes as subclasses of existing classes [15], thereby enhancing reusability of complex, hierarchical models by means of inheritance and components assembly.

REFERENCES


