Hierarchical Development of Model Classes in the DEVS-Scheme Simulation Environment

TAG GON KIM

University of Kansas, Lawrence

Abstract—DEVS-Scheme is a realization of Zeigler's DEVS (Discrete Event System Specification) formalism in a LISP-based, object-oriented environment which supports specification of discrete event models in hierarchical, modular fashion. Within the DEVS-Scheme environment, the modeler can develop new model classes in an incremental manner by defining new classes, specific to his application domains, based on the existing ones provided by the environment. Concepts of the model–simulator pairing associated with the DEVS formalism and polymorphism inherited from the implementation of such concepts in the object-oriented paradigm make it possible to develop new model classes in such an incremental manner. This paper describes how the concepts of the model–simulator pairing and polymorphism can be exploited in the development of new model classes in DEVS-Scheme. The development of subclasses, suited for simulation modeling for parallel computer systems, of the class kernel-models in DEVS-Scheme is exemplified.

1. INTRODUCTION

THE DISCRETE EVENT SYSTEM SPECIFICATION (DEVS) formalism introduced by (Zeigler, 1976, 1984) provides a means of formal specification for a mathematical object, called a system. Within the formalism, a system has a time base, inputs, states, outputs, and functions for determining next states and outputs and time advance, given current states and inputs (Concepcion & Zeigler, 1988).

The DEVS-Scheme environment is a realization of the DEVS formalism in a LISP-based, object-oriented framework, which enables the modeler to specify models in a manner that is closely parallel the DEVS formalism (Kim & Zeigler, 1987, 1990). DEVS-Scheme supports building models in a hierarchical, modular manner, a systems-oriented approach that is not possible in conventional languages.

Simulation management in DEVS-Scheme is based on the principles of abstract simulator, a conceptual device capable of interpreting dynamics specified by using the DEVS formalism. The principles are implemented by three specialized classes for abstract simulators. Thus, whenever a model object is created, an associated abstract simulator object needs to be created from one of three classes and attached to the model. Such a model–simulator pair is recorded by their instance variables so that the model knows its simulator and the simulator knows its model. However, simulators do not know any information inside the models. Thus, during the simulation the abstract simulator consults with its model to know various information necessary to manage the simulation such as the state transition functions and destinations of received messages. The consultations are based on message passings between a pair of the abstract simulator and its associated model.

DEVS-Scheme is designed such that the classes for models can be developed as subclasses of the existing classes without defining new classes for the associated abstract simulators. Developing new classes in such a manner is based on the ability of models of different classes to respond to the same messages received from abstract simulators of the same class. The ability is called polymorphism that is inherited from the object-oriented language on which DEVS-Scheme is implemented.

This paper describes how polymorphism can be exploited in the development of new model classes in DEVS-Scheme. Specifically, the development of subclasses, suited for modeling parallel computer systems, of the class coupled-models in DEVS-Scheme is exemplified to show importance of polymorphism. Section 2 reviews the DEVS formalism and its implementation DEVS-Scheme. Section 3 describes class development in DEVS-Scheme based on the coupling relations. Section 4 presents polymorphism in message-based simulation within the DEVS-Scheme environment. Finally, concluding remarks follow in section 5.
2. THE DEVS FORMALISM AND DEVS-SCHEME

We shortly review the hierarchical, modular DEVS formalism and its associated abstract simulator concepts. Also, a realization of the formalism in a LISP-based, object-oriented environment will be briefly described.

2.1. The DEVS Formalism and Abstract Simulator

The DEVS (Discrete Event System Specification) formalism, developed by Zeigler (1976, 1984), specifies discrete-event models in 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. Such specification has the following information: the set of input ports, the set of output port, the set of state variables, the internal and external transition functions, the output function, and time advance function. 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 hierarchical construction. A coupled model contains the following information: the set of input ports, the set of output ports, the coupling scheme, and the selection function. Formal definitions of atomic and coupled DEVS and detail descriptions for the definitions can be found in (Zeigler, 1984).

The abstract simulator (Zeigler, 1984) is a conceptual device capable of interpreting the dynamics specified by the DEVS formalism. Two types of the abstract simulator have been defined; one for the atomic DEVS and the other for the coupled DEVS. The architectures and performance of distributed simulation systems, derived from the abstract simulator concept, have been intensively studied (Baik, 1985) and some of which were implemented by multiprocessor computer systems (Concepcion, 1985).

The abstract simulator for an atomic DEVS interprets the dynamics of the atomic DEVS. Figure 1 shows the algorithm for the abstract simulator (Zeigler, 1984). The algorithm is divided into two parts: “when receive \((x, t)\)” part and “when receive \((*, t)\)” part. The function of the “when receive \((*, t)\)” part for the simulator is to schedule internal events and execute transitions due to such events. The function of the “when receive \((x, t)\)” part is to execute transitions caused by external events. The correctness of the simulator has been proved in Concepcion and Zeigler (1988).

The responsibilities of the abstract simulator for a coupled DEVS is to synchronize the component abstract simulators and to handle external events. As with the abstract simulator for the atomic DEVS, the algorithm for the coordinator is divided into two phrases of the “when receive . . .” parts. Figure 2 shows the algorithm for the abstract simulator (Zeigler, 1984). The correctness of the algorithm has been also proved in (Concepcion & Zeigler, 1988).

2.2. The DEVS-Scheme Environment

DEVS-Scheme is an object-oriented environment which realizes the DEVS formalism and its associated abstract simulator concepts. To realize them, DEVS-Scheme first 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 above.

The class models has two subclasses to realize two types of models defined in the DEVS formalism. The two subclasses are atomic-models realizing atomic DEVS models and coupled-models for coupled DEVS models. The class atomic-models realizes the atomic level of the DEVS formalism by use of its variables and methods that correspond to the components of the structure in the formalism. Four instance variables of the atomic-models, namely int-transfn, ext-transfn, outputfn, and time-advancefn realize the internal transition function, external transition function, output function, and time-advance function of atomic DEVS, respectively, when they are evaluated. Methods of atomic-models and their examples are described in detail in Zeigler (1987).

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 desired communication links among the children. Instance variables corresponding to children and coupling relations, and methods that manipulate the variables, realize the formalism. Methods, get-children, get-influences, get-receivers, and translate are available for the coupled-models (Kim & Zeigler, 1990).

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.

3. CLASS DEVELOPMENT IN DEVS-SCHEME

Once the basic classes for models and the associated simulator classes are provided in DEVS-Scheme, the modeler can develop new model classes for his own applications. Often, the modeler needs to develop different model classes based on the different couplings among model components.

3.1. Specification of the Coupling Scheme

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). Figure 3 shows a coupling scheme of a model AB.

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. For example, external input coupling, EIC = {(AB.in1, A.in) (AB.in2, B.in)} in Figure 3, means that input port in1 of AB is connected to input port in of A, and input port in2 of AB is connected to input port in of B. The period prefix the name of a component to the names of ports to uniquely identify them. This notation obviates having to give different names to all the ports.

![Figure 3. Model coupling scheme.](image-url)
External output coupling is the relation of the output ports of the coupled model to those of the component models. It represents how the output ports of the composite model are connected to the output ports of the component models. Thus EOC = \{ (B.out, AB.out) \} in Figure 3 means that the output port out of B is connected to the output port out of AB.

Internal coupling is the relation of the output ports of the components to the input ports of other components. It specifies 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. The specification IC = \{ (A.out, B.in1) (B.out1, A.in1) \} in Figure 3 means that the output port out of A is connected to the input port in1 of B, and the output port out1 of B is connected to the input port in1 of A. The list of components connected to a component M is called influences of M.

As described above, the internal coupling scheme of a coupled model specifies how the components of the coupled model are connected together. Two kinds of the internal coupling are possible: uniform and nonuniform. The uniform internal coupling means that once a influences pattern of a representative component in a coupled model is specified, the pattern is applied for all the components in the coupled model. On the other hand, a coupled model with the nonuniform internal coupling scheme should have a explicit coupling scheme specifying connections among all components in the coupled model. We now define two subclasses of coupled-models based on different internal coupling schemes: digraph-models for nonuniform internal coupling scheme in section 3.2 and kernel-models for uniform internal coupling scheme in section 3.3.

3.2. Class Digraph-models

Digraph-models is defined by a finite set of explicitly specified children and an explicit coupling scheme connecting them. Internal and external coupling relations specify how output ports of children couple to input ports of other children, and how input/output ports of coupled-models couple to input/output ports of its components, respectively. Methods, build-composition-tree, set-ext-out-coup, and set-ext-inp-coup are available for specifying an external coupling scheme. Set-inf-dig and set-int-coup are the methods for internal coupling specification. Since digraph-models is a subclass of coupled-models a coordinator is attached to a digraph model. Figure 4 shows the first version class hierarchy in DEVS-Scheme.

3.3. Class Kernel-models and Subclasses

As we mentioned earlier, the coupling scheme of the class digraph-models is nonuniform. However, there exist classes of models in which the influences pattern of components is uniform. For example, in a hypercube model, influences of a cell M1 consists of cells located nearest M1. We call the class of models having such uniform coupling scheme kernel-models. As an example, we define the class kernel-models in DEVS-Scheme as a subclass of the class coupled-models.

Since kernel-models is a subclass of coupled-models, an abstract simulator attached to the kernel-models is an object of the class coordinators. We now describe the role of polymorphism in defining new classes in DEVS-Scheme. To do so, we first will define a new class called hypercube-models as a subclass of kernel-models. We then will show the ability of the classes digraph-models and hypercube-models to respond to the same message, received from their respective coordinators, in different ways.

Hypercube-models is a specialization of kernel-models, an instance of which realizes the hypercube configuration representing a well-known multiprocessor computer architecture. In such a configuration, any n-dimensional hypercube configuration consists of two isomorphic (n - 1)-dimensional hypercube configurations.

In a hypercube model, a component communicates only with some of the closest neighborhoods in the hypercube, resulting in minimum communication paths among the components. To specify the number of influences of a component, an instance variable num-infl is provided. The method get-influences of hypercube-models first accesses the num-infl and returns the first num-infl number of the closest neighborhoods in the hypercube. Since the influences pattern for the hypercube-models is uniform, the internal coupling of a component and its influences in a hypercube model
can be computed by using the coupling scheme of the origin cell position and its influencees.

If the external coupling of the hypercube model is origin-only, the method checks whether one of the two is a member of its receivers. If one of them is a receiver, the given port name is returned. Otherwise, it looks up the out-in-coup table. Since the out-in-coup table has the cell positions of the influencees of the origin cell, the method computes a cell position of an influencee of the origin cell from the cell position of the given influencee before it looks up the table. The number of influencees of each cell, num-infl, in a hypercube model ranges from zero to the dimension of the hypercube. By definition of its influencees, the Hamming distance between positions of a cell and any of its influencees is 1. Thus, if num-infl = 3 in a 3-dimensional hypercube model, three influencees of a cell at (000) are cells at (100), (010), and (001), and those of a cell at (111) are cells at (011), (101), and (110), and so on. However, if num-infl = 2 in the 3-dimensional hypercube model, two influencees of the cell at (0, 0, 0) are cells located at (100) and (010). The two influencees are obtained by selecting the first two out of all three influencees.

As described in section 2, each model class in DEVS-Scheme has its associated abstract simulator class which interprets model’s dynamics in simulation. This implies that whenever we develop new subclasses of the class coupled-models, we have to develop the associated simulator classes. For the modeler, this seems to be a critical problem in developing new classes in DEVS-Scheme. However, such a problem does not arise within the DEVS-Scheme environment. The reason is that in DEVS-Scheme, a model created from any subclass of the class coupled-models may have an associated coordinator created from the class coordinators which is provided by DEVS-Scheme. By taking an advantage of polymorphism inherited from the underlying object-oriented language, DEVS-Scheme can proceed simulation based on such a pair of the model-coordinator. This is why the modeler can develop new model classes in DEVS-Scheme without changing the existing ones.

4. POLYMORPHISM IN SIMULATION

4.1. Message-Based Simulation

With the DEVS-Scheme environment, simulation proceeds by means of messages passed among the three specialized processors associated with models described earlier. The processors carry information concerning external events and internal scheduling, and others needed for synchronization. Types of messages to be transmitted and received are: x, *, y, and done. Each message bears information about message source, time, and content. The content of a message consists of port and value. While x-message and *-message are transmitted from parent processor to its child(ren), y-message and done-message are transmitted from child(ren) processor(s) to its parent.

Different classes of processors respond to a received message in different ways, the ability of which is referred to as polymorphism. Figure 5 summarizes how processors respond to different types of messages when they receive them. During message passing among processors, the processor that receives a message consult with the attached dev-component and get knowledge—such as receivers, influencees, interface map, and others—that is required to route the received message to their appropriate components. For example, if a processor is a coordinator, it consults with the attached coupled model. If consulted, the coupled model computes receivers, influencees, and interface map, using its methods get-receivers, get-influencees, and translate, respectively.

The term "polymorphism" was first introduced by Strachey (Strachey, 1967) to characterize functions that work on arguments of more than one type. In the context of object-oriented languages, polymorphism is the ability of different classes of objects to respond to the messages by associating generic names with objects’ behaviors (Stefik & Bobrow, 1986).

We now discuss polymorphism of digraph-models and hypercube-models to respond the same message received from their coordinators. Consider two models created from different subclasses of coupled-models: dig-M, an object of digraph-models, and hc-M, an object of hypercube-models. Assume that dig-M has three components with the coupling as shown in Figure 6, and that hc-M is a 3-dimensional hypercube as shown in Figure 7. The figures also show hierarchical simulator architectures for dig-M and hc-M, respectively. C:dg-M and C:hc-M are objects of the class coordinators that are attached to dig-M and hc-M, respectively. Recall that C:dg-M and C:hc-M consult with their respective models, dig-M and hc-M, to get information necessary to proceed simulation. The consultations are done by passing messages between the coordinators and associated models. Since C:dg-M and C:hc-M are objects created from the same class, they send the same message to their respective models, expecting that the responses to the message should be different.

Two types of messages, namely an external (x, t) message and an internal (y, t) message that coordinators receive are considered. When a coordinator receives the (x, t) message from its parent, it consults with its associated model that knows how to compute the destination of the external message. Similarly, when a coordinator receives the (y, t) message from its child, it consults with the associated model that knows how to compute the destination of the internal message.
For the external message, assume that both C: dig-M and C: hc-M receive an external event \((x, t)\) from outside. Since the two coordinators do not know the destination(s) of the external event message, they have to consult with their attached models to know the external input couplings. Each of dig-M and hc-M comprises the knowledge on coupling and has a means for inferring the knowledge to compute its receivers. The knowledge on the coupling is represented by the external coupling scheme and stored in an instance variable of each model. The inferencing means is realized by the method get-receivers. The point is that get-receivers of dig-M and get-receivers of hc-M employ different ways in computing their receivers. Figure 8 gives the method get-receivers of digraph-models and hypercube-models, respectively.

The Figure 6 and Figure 7 illustrate models’ responses to associated coordinators’ requests for receivers and influences during simulation. As shown in Figure 6, when dig-M receives the message get-receivers from C: dig-M, it computes its receivers and returns (DM1 DM2) as a result to C: dig-M. Similarly, when hc-M receives the message get-receivers from C: hc-M, it computes its receivers and returns (HM0) as a result to C: hc-M, as shown in Figure 7. When C: dig-M receives (DM1 DM2) from dig-M, it then routes the \((x, t)\) message to S: DM1 (simulator of DM1) and S: DM2 (simulator of DM2). Likewise, when C: hc-M re-
ceives (HM0) from hc-M, it then routes the (x, t) message to S:HM0.

For the internal message, assume that C:dig-M and C:hc-M receive the (y, t) messages from S:DM2 and S:HM5, respectively. To response the (y, t) message, C:dig-M and C:hc-M have to know the influences of S:DM2 and S:HM5, respectively. To do so, C:dig-M and C:hc-M consults with their models dig-M and hc-M. Each of dig-M and hc-M comprises the knowledge on its internal coupling and has a means for inferring the knowledge to compute influences for a given component. The knowledge on the coupling is represented by internal coupling scheme and stored in an instance variable of each model. The inferring means is realized by the method get-influences. Again, get-influences of dig-M and get-influences of hc-M employ different ways in computing influences. Figure 9 presents the method get-influences of the digraph-models and hypercube-models, respectively. Figure 6 and Figure 7 illustrate coordinators’ requests to associated models during simulation, where dig-M returns (DM3) to C:dig-M and hc-M returns (HM1 HM4 HM7) to C:hc-M.

Based on the two coupling relations described above, namely, the digraph coupling and the hypercube coupling, the second version of class hierarchy for DEVS-Scheme can be developed as shown in Figure 10. Note that even if we defined hypercube-models as a subclass of kernel-models, no abstract simulator class for hypercube-models is defined. Polymorphism makes it possible to develop new subclasses in DEVS-Scheme in such an incremental manner.

As another example, consider the class ring-models as a subclass of kernel-models, which can be used to model the popular ring connection in a computer network. An object of ring-models consists of a set of components which are connected in a circular manner. Such a ring model has a uniform internal coupling scheme; influences of a component M in the ring model is only one component next to M in the ring. Consider a ring model with ten components, that is RM0, RM1, . . . , RM9. As with the C:hc-M, when C:r-M receives an external event message (x, t), C:r-M sends a message get-receivers to r-M. R-M then returns (RM1) to C:r-M. Similarly, when C:r-M receives a (y, t) message from RM2, it consults with r-M to know influences of RM2 by sending the message get-influences to r-M. R-M then return (RM3) to C:r-M.

Yet another subclass of kernel-models called cellular-models was developed and details of the class can be found in (Kim, 1988). Figure 11 shows the third version class hierarchy in DEVS-Scheme. The class kernel-models and its subclasses shown in Figure 11 are developed mainly for studying parallel computer systems through simulation modelling.

5. CONCLUDING REMARKS

The DEVS-Scheme environment allows us to develop, in an incremental manner, a variety of subclasses based

![Diagram](image-url)

**FIGURE 7.** Hypercube model (left) and its processor (right).

![Diagram](image-url)

**FIGURE 8.** Method get-receivers (a: Digraph-models; b: Hypercube-models).
method get-influences (m)
    return successors of m in influence-digraph

(a)

method get-influences (m)
    compute cell positions of closest neighborhoods of m in hypercube
    return list of components located at the computed cell positions

(b)


on the exiting ones, which is not possible using conventional discrete event simulation languages. Each such subclass can be developed specific to our application domain. For example, the class controlled-models was developed as a subclass of the kernel-models for simulation study on robotics and intelligent control (Zeigler, 1990). To develop a new subclass of the class coupled-models the modeler needs to define its coupling scheme and associated methods to inference such a coupling scheme. A coupling scheme specifies the internal and external coupling of the new subclass. As recognized in (Freundlich, 1990), such a coupling scheme is not data but knowledge in that a coupled model knows how it computes a connection from one component to another using the coupling scheme. More specifically, the model does not store the connection in a way that one component is connected to another component. Instead, the model has knowledge of the coupling scheme to compute the connection between models using an inference mechanism. In fact, the methods get-receivers and get-influences associated with a coupled model, which are explained in section 4, can be viewed as inference engines to compute receivers and influences using the coupling scheme, respectively. Since each subclass of the class coupled-models has its own get-receivers and get-influences, it responds to its coordinator’s messages requesting destination model(s) in its own way. It is worth to note that polymorphism does not simply mean that different models created from different subclasses return different receivers and different influences. It actually means that different models created from different subclasses have different ways to compute their receivers and influences using their own coupling schemes.

Concepts of a model–simulator pair and their implementation of DEVS-Scheme in an object-oriented environment makes it possible for the modeler to develop subclasses of coupled-models in an incremental manner. The development of model subclasses does not require to define new simulator classes associated with model subclasses. This is because a model created from any such subclass can be associated with a coordinator created from the class coordinators that is provided by DEVS-Scheme. Once associated, a simulator asks questions on couplings to its model by means of message passing during simulation. When asked, a model applies its methods to inference coupling knowledge represented by coupling scheme within the model to answer such questions. Since the same class of processor is associated with different classes of models, polymorphism inherited from the underlying object-oriented language is exploited in the answering process.

The approach to developing new model classes described in this paper is a powerful means for application-oriented development of model classes, not possible by using conventional simulation languages or environments. It should be noted that the approach was possible by employing concepts of the model–simulator pairing associated with the DEV5 formalism and implementing the concepts in the object-oriented paradigm. Combination of DEVS-Scheme and ESP-Scheme, a model base management system for DEV5 models developed in DEV5-Scheme, provides a pow-
erful framework for a knowledge-based simulation and system design environment (Kim et al., 1990).

REFERENCES


