THE ROLE OF POLYMORPHISM IN CLASS EVOLUTION IN THE DEVS-SCHEME ENVIRONMENT

Tug Gon Kim
Department of Electrical and Computer Engineering
1013 Learned Hall
University of Kansas
Lawrence, Kansas 66045

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. This paper describes how polymorphism can be exploited in the development of new model classes within the DEVS-Scheme environment. The development of subclasses of the class coupled-models in DEVS-Scheme, which are suited for simulation modeling for parallel computer systems, is exemplified to show the role of polymorphism.

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, and outputs, and functions for determining next states and outputs and time advance, given current states and inputs [Conception and 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 closely paralleling the DEVS formalism [Kim and Zeigler 1987; Kim and Zeigler 1990]. DEVS-Scheme supports building models in a hierarchical, modular manner, a systems oriented approach 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 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 simulation the abstract simulator consults with its model to know various information necessary to manage simulation such as 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 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 of the class coupled-models in DEVS-Scheme suited for modeling parallel computer systems is exemplified to show importance of polymorphism. Section 2 reviews the DEVS formalism. Section 3 describes DEVS-Scheme and its simulation management. Section 4 shows development of new model classes in DEVS-Scheme based on polymorphism. Conclusions follow in section 5.

2. THE DEVS FORMALISM AND ABSTRACT SIMULATOR

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

2.1 DEVS Formalism

The DEVS (Discrete Event System Specification) formalism, developed by Zeigler [Zeigler 1976, 1984], specifies discrete-event models in hierarchical, modular form. Within the formalism, one must specify 1) 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 the dynamics of the model. The second form of 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 given rise to hierarchical construction.

Formally, an atomic DEVS is defined by a structure [Zeigler 1984]:

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

where
\( X \) is a set, the external input event types
\( S \) is a set, the sequential states
\( Y \) is a set, the external output event types
\( \delta_{\text{int}} \) is a function, the internal transition specification
\( \delta_{\text{ext}} \) is a function, the external transition specification
\( \lambda \) is a function, the output function
\( \tau \) is a function, the time-advance function

with the following constraints:
(a) The total state set of the system specified by \( M \) is \( Q = \{ (s,e) | s \in S, 0 \leq e \leq \tau(s) \} \);
(b) \( \delta_{\text{int}} \) is a mapping from \( S \) to \( S \);
(c) \( \delta_{\text{ext}} \) is a function:
\( \delta_{\text{ext}} : Q \times X \rightarrow S \);
(d) \( \tau \) is a mapping from \( S \) to the non-negative reals with infinity:
\( \tau : S \rightarrow \mathbb{R} \), and
(e) \( \lambda \) is a mapping from \( Q \) to \( Y \):
\( \lambda : Q \rightarrow Y \).

An interpretation of the DEVS and a full explication of the semantics of the DEVS are in [Zeigler 1984].

Closed under composition, the DEVS formalism defines a coupled DEVS in modular form as a structure [Zeigler 1984]:

\[ DN = \langle D, \{ M_i \}, \{ h_i \}, \{ Z_{ij} \}, \text{SELECT} > \]

where
\( D \) is a set, the component names;
for each \( i \) in \( D \),
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 and Zeigler 1990].

3.2 Class Processors and Subclasses

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, dev-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.

Simulation proceeds by means of messages passed among the above three specialized processors, which 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 last of which, in turn, consists of port and value. While $x$-message and $y$-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 processors respond to a message in different ways. Likewise, a processor responds to different messages in different ways. Fig. 1 summarizes how processors respond to different types of messages when they receive them. During message passing among processors, the processor that receives a message consults with the attached dev-component and gets knowledge—such as receivers, influences, 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, influences, and interface map, using its methods get-receivers, get-influences, and translate, respectively, as requested.

3.3 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 $(M_1.p_1, M_2.p_2)$ means that the output port $p_1$ of model $M_1(M_1.p_1)$ is connected to the input port $p_2$ of model $M_2(M_2.p_2)$. In this specification, “$M_1.p_1$ is connected to $M_2.p_2$” means that information flows only from $M_1.p_1$ to $M_2.p_2$. 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. For example, external input coupling $EIC = \{(AB.in1, A.in2) (AB.in2, B.in)\}$ in Fig. 2, means that input port in1 of AB is connected to input port in A, and input port in2 of AB is connected to input port in B. The period prefixes the name of a component to names of ports to uniquely identify them. This notation obviates having to give different names to all the ports.

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 Fig. 2 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
The Role of Polymorphism in Class Evolution in the DEV-S-Scheme Environment

<table>
<thead>
<tr>
<th>Messages Types</th>
<th>Types of Source</th>
<th>Type of Destination</th>
<th>Destination Processor's Response</th>
<th>Devs-components Methods Applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>COOR</td>
<td>COOR</td>
<td>send ( x ) to its receivers</td>
<td>get-receivers</td>
</tr>
<tr>
<td></td>
<td>COOR</td>
<td>SIM</td>
<td>compute its external transition</td>
<td>ext-transition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COOR or ROCO</td>
<td>COOR</td>
<td>send ( y ) to its imminent child</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIM</td>
<td>compute output ( y ) if possible</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>send ( y ) to its parent</td>
<td>output? get-parent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>compute internal transition</td>
<td>int-transition</td>
</tr>
<tr>
<td>y</td>
<td>SIM or COOR</td>
<td>COOR</td>
<td>translate ( y ) to ( x )</td>
<td>translate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>send ( x ) to parent</td>
<td>get-parent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>send ( x ) to source's influences</td>
<td>get-influences</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>wait until done from</td>
<td>get-wait-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>receivers if done</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>responds to ( x )-message</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>SIM or COOR</td>
<td>COOR</td>
<td>wait until done from</td>
<td>get-wait-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>all influences if done</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>responds to ( y )-message</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>compute minimum of next event time</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Processors' Responses to Messages

COOR: Coordinator
SIM: Simulator
ROCO: Root-coordinator

specification \( IC = \{ (A.out, B.in1) \ (B.out1, A.in1) \} \) in Fig. 2 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 \( M \).

![Figure 2. Model Coupling Scheme](image)

External Input Coupling = \{ (A.in1, B.in) \ (A.in2, B.in) \}
External Output Coupling = \{ (B.out, A.out) \}
Internal Coupling = \{ (A.out, B.in) \ (B.out1, A.in1) \}

4. CLASS EVOLUTION IN DEV-S-SCHME

One way to taxonomize the class coupled-models in DEV-S-Scheme is based on the coupling scheme of coupled-models. As we defined earlier, the internal coupling scheme of a coupled model specifies how components of the coupled model connected together. Two kinds of the internal coupling are possible: uniform and non-uniform. For a coupled model with the uniform internal coupling, influences of each component in the coupled model has a uniform pattern. On the other hand, a coupled model with the non-uniform internal coupling scheme has no such a uniform pattern of influences of components.

We now define a subclass of coupled-models called digraph-models that has non-uniform coupling scheme in the above sense.

4.1 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 methods for internal coupling specification. Since digraph-models is a subclass of coupled-models a coordinator is attached to a digraph model. Fig. 3 shows the first version class hierarchy in DEV-S-Scheme.

4.2 Class Kernel-models and Subclasses

As we mentioned earlier, the coupling scheme of the class digraph-models is non-uniform. 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 show the role of polymorphism in defining new classes in DEVS-Scheme. To do so, we first define a new class called hypercube-models as a subclass of kernel-models. We then 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 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 neighborhoods 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 influences.

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 influences of the origin cell, the method computes a cell position of an influence of the origin cell from the cell position of the given influence before it looks up the table.

The number of influences of each cell, num-infl, in a hypercube model ranges from zero to the dimension of the hypercube. By definition of its influences, the Hamming distance between positions of a cell and any of its influences is 1. Thus, in a 3-dimensional hypercube model, three influences of a cell at (0 0 0) are cells at (1 0 0), (0 1 0), and (0 0 1), and those of a cell at (1 1 1) are cells at (0 1 1), (1 0 1), and (1 1 0), and so on. However, if the number of influences of the model is two, the influences of the cell at (0 0 0) are cells at (1 0 0) and (0 1 0) in the 3-dimensional hypercube.

4.3 Polymorphism

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 and Bobrow 1986].

We now show polymorphism of digraph-models and hypercube-models to respond the same message received from their
The Role of Polymorphism in Class Evolution in the DEVS-Scheme Environment

![Diagram of DEVS-Scheme class hierarchy]

Figure 6. Second Version Class Hierarchy of DEVS-Scheme

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 Fig. 4, and that hc-M is a 3-dimensional hypercube as shown in Fig. 5. The figures also show hierarchical simulator architectures for dig-M and hc-M, respectively. C:dig-M and C:hc-M are objects of the class coordinators that are attached to dig-M and hc-M, respectively. Recall that C:dig-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:dig-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 (done, t) message that coordinators receive are considered. When a coordinator receives the (x, t) message, it consults with its associated model to know external input coupling of the model. When a coordinator receives the (done, t) message, it consults with its associated model to know internal coupling of the model.

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. As shown in Fig. 1, the coordinators C:dig-M and C:hc-M send the same message get-receivers to their respective models dig-M and hc-M. However, since dig-M and hc-M are created from different classes, their responses to the message get-receivers are different. That is, dig-M responds to the message by returning (DM1 DM2 DM3) to C:dig-M while hc-M does by returning (HM0) to C:hc-M. When C:dig-M receives (DM1 DM2) from dig-M, it then routes the (x, t) message to S:DM1 (simulator of DM1), S:DM2 (simulator of DM2), and S:DM3 (simulator of DM3). Likewise, when C:hc-M receives (HM0) from hc-M, it then routes its (x, t) message to S:HM0.

For the internal message, assume that C:dig-M and C:hc-M receive the (done, t) messages from DM2 and HM5, respectively. As shown in Fig. 1, to respond to the (done, t) message, C:dig-M and C:hc-M have to know the influences of DM2 and HM5, respectively. To know the influences (or internal coupling scheme), C:dig-M and C:hc-M send the message get-influences to dig-M and hc-M, respectively. Again, the responses from the two DEVS models to the same message are different. Dig-M returns (DM3) and hc-M returns (HM1 HM2 HM7).

The second version of class hierarchy for DEVS-Scheme is shown in Fig. 6. 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 a incremental manner.

As another example, consider the class ring-models as a subclass of kernel-models. 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, i.e., RM0, RM1,..., RM9. As with the C:hc-M, when C:rm-M receives an external event message (x, t), C:rm-M sends a message get-receivers to r-M. R-M then returns (RM1) to C:rm-M. Similarly, when C:rm-M receives a (done, t) message from RM2, it consults with r-M to know influences of RM2 by send-

![Diagram of DEVS-Scheme class hierarchy]

Figure 7. Third Version Class Hierarchy of DEVS-Scheme
ing the message get-influences to r-M, R-M then return (RM3) to C: r-M.

Similarly, we can develop new subclasses of kernel-models in DEVS-Scheme. Whenever we develop such new classes, we have to define methods such as get-receivers, get-influences that are specific to the new classes to reply questions asked by coordinators. Another subclass of kernel-models called cellular-models was defined in [Kim 1988]. Fig. 7 shows the third version class hierarchy in DEVS-Scheme. The class kernel-models and its subclasses shown in Fig. 7 has been defined for simulation modeling of parallel computer systems.

5. CONCLUSIONS

We have described the development of classes in the DEVS-Scheme environment in an incremental manner. Polymorphism inherited from the underlying object-oriented language of DEVS-Scheme made it possible to do so. The coupling scheme of a hierarchical, modular model was used as a basis for developing such classes. Although we demonstrated the development of the classes suited for modeling of parallel computer systems, namely kernel-models and its subclasses, classes specific to application domains may be developed.

REFERENCES


