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

Tag 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 model and 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 simulations 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 which 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 giving rise to hierarchical construction.

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

$$ M = \langle x, s, y, \delta_{in}, \delta_{ext}, i, t_a \rangle $$

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
- $\delta_{in}$ is a function, the internal transition specification
- $\delta_{ext}$ is a function, the external transition specification
- $i$ is a function, the output function
- $t_a$ 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) $\delta_{in}$ is a mapping from $s$ to $S$:

$$ \delta_{in}(s) \rightarrow S $$

(c) $\delta_{ext}$ is a function:

$$ \delta_{ext}: Q \times X \rightarrow S $$

(d) $i$ is a mapping from $S$ to the non-negative reals with infinity:

$$ \tau: S \rightarrow R, \text{ and} $$

(e) $\tau$ is a mapping from $Q$ to $Y$:

$$ \tau: Q \rightarrow Y $$

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]:

$$ DN = \langle D, \{ M_i \}, \{ I_i \}, \{ Z_{ij} \}, \text{SELECT} \rangle $$

where

- $D$ is a set, the component names;
- for each $i$ in $D$,
and for each
with the following constraints:

\[ M_1 = \langle X_i, Y_i, S_i, \eta_i, \tau_i \rangle \]
\[ I_i = \text{a set, the influences of } i \]
\[ Z_{ij} \text{ is a function, the i-to-j output translation} \]

and

\[ \text{SELECT is a function, the tie-breaking selector} \]

with the following constraints:

\[ I_i = \text{a subset of } D, i \text{ is not in } I_i \]
\[ Z_{ij} : Y_i \rightarrow X_j \]
\[ \text{SELECT: subsets of } D \rightarrow D \]
\[ \text{such that for any non-empty subset } E, \text{SELECT}(E) \text{ is in } E. \]

2.2 Abstract Simulator

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], some of which were implemented by multiprocessor computer systems [Concepcion 1985].

The abstract simulator for an atomic DEVS has five variables that realize the dynamics of the atomic DEVS. The correctness of the simulator has been proved [Zeigler 1988], and the algorithm for the simulator that is divided by two parts — "when receive \((s,t)\)" part and "when receive \((s,t)\)" part — was given in [Zeigler 1984]. The function of the "when receive \((s,t)\)" part for the simulator is to schedule internal events and execute transitions due to such events. The function of the "when receive \((s,t)\)" part is to execute transitions caused by external events.

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 by two phrases of "when receive ..." parts. The correctness of abstract simulator for the coupled DEVS has been also proved [Concepcion and Zeigler 1984].

3. 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 subclasses of a universal class called entities. The class entities provides tools — such as constructor and destructor — for manipulating objects for not only the class itself but two subclasses defined above.

3.1 Class Models and Subclasses

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 components of structure in the formalism. Four instance variables of the atomic-models, named 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 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, devs-component, and models have an instance variable, processor. A root-coordinator manages the overall simulation and is linked to a coordinator of the outermost 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 children, y-message and done-message are transmitted from children 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 consult with the attached devs-component and get 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 \((M1.pl, M2.p2)\) means that the output port \(p1\) of model \(M1\) is connected to the input port \(p2\) of model \(M2\).

In this specification, \(M1.pl\) is connected to \(M2.p2\) means that information flows only from \(M1.pl\) 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. For example, external input coupling, \(EIC = \{(AB.in1, A.in) \}, (AB.in2, B.in)\}\) in Fig. 2, 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 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 component 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 coupled model 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 DEVS-Scheme Environment

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

Figure 1. Processors' Responses to Messages
COOR: Coordinator
SIM: Simulator
ROCO: Root-coordinator

specification IC = \{(A.out, B.in), (A.in, B.out)\} 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.

External Input Coupling = \{(A.out1, B.in), (A.in2, B.out)\}
External Output Coupling = \{(B.out1, A.in), (B.out2, A.out)\}
Internal Coupling = \{(A.out, B.in), (B.out1, A.in)\}

Figure 2. Model Coupling Scheme

4. CLASS EVOLUTION IN DEVS-SCHEME

One way to taxonomize the class coupled-models in DEVS-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 have 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-in-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 DEVS-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 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 influencees of a component, an instance variable num-infl is provided. The method get-influencees of hypercube-models first accesses the num-infl and returns the first num-infl number of the closest neighborhoods in the hypercube. Since the influencees pattern for the hypercube-models is uniform, the internal coupling of a component and its influencees 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 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, in a 3-dimensional hypercube model, three influencees 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 influencees of the model is two, the influencees 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

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. 4, 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 (HMO) 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 (HMO) from hc-M, it then routes its (x, t) message to S:HMO.

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 response 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. As with the C:hc-M, when C:rm receives an external event message (x, t), C:rm consults with r-M 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.
ing the message get-influences to r-M, R-M then return (RM3) to Cr-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 a 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


