ISOMORPHIC REPLICATION OF HIERARCHICAL MODELS IN DEVS-SCHEME

Tag Gon Kim
Department of Electrical and Computer Engineering
University of Kansas, Lawrence, KS 66045

Bernard P. Zeigler
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721

ABSTRACT

DEVS-Scheme is a realization of the DEVS formalism in a LISP-based object-oriented framework. It enables a modeler to develop discrete-event models in a hierarchical, modular manner. In developing complex, hierarchial models in DEVS-Scheme, isomorphic replication of models is needed to conveniently construct such models with varying numbers of similar components (each of which may itself be a complex structure). This paper describes how DEVS-Scheme takes advantage of SCOOPS, the object-oriented superset of Scheme, to develop methods for creating isomorphic copies of complex, hierarchical models and also the associated methods for checking isomorphism between models.

1. INTRODUCTION

The Discrete Event System Specification (DEVS) formalism introduced by (Zeigler 1976) 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 (Concepcion 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 1988; Kim and Zeigler 1990). DEVS-Scheme supports building models in a hierarchical, modular manner, a systems oriented approach not possible in conventional languages.

Models developed using DEVS-Scheme may have complex, hierarchical structures. Often, such complex structures can be conveniently constructed by using identical copies of existing models. Thus, special attention has to be paid to replicating the structures. For example, all members (or children) of a kernel model in DEVS-Scheme are identical copies of a model called *init-cell* within the kernel model. The *init-cell* can be an instance of any sub-classes of the *models* class in DEVS-Scheme.

This paper describes how DEVS-Scheme takes advantage of SCOOPS, the object-oriented superset of Scheme, to develop methods for creating isomorphic copies of complex, hierarchical models and also the associated methods for checking isomorphism between models. Section 2 describes manipulation of complex, hierarchical structures in DEVS-Scheme. Section 3 describes the method *make-new* in DEVS-Scheme. Section 4 presents definition of *isomorphism* between models in general, based on which methods for checking isomorphism between two DEVS models are developed in section 5.

2. MANIPULATION OF COMPLEX STRUCTURES

Basic model classes in DEVS-Scheme are *atomic-models* and *coupled-models*. The object-oriented nature of DEVS-Scheme allows the modeler to freely develop new model classes and add them to DEVS-Scheme as subclasses of existing model classes. Extensions of atomic-models include classes for rule-based and table-based modelling. Likewise, *coupled-models* has been specialized into *digraph-models* (for explicit coupling) and *kernel-models* (for models with unbounded numbers of components). *Kernel-models* has further specialized classes, such as *broadcast-models, kypercube-models, cellular-models,* etc., based on coupling schemes of components in a kernel model (see (Kim 1988) for details on the class hierarchy in DEVS-Scheme).

As we have seen above, the development of complex, hierarchical structures using DEVS-Scheme usually requires isomorphic copies of existing models. DEVS-Scheme provides two main alternatives for creating *isomorphic* copies of complex, hierarchical models. The first method, *make-new*, when sent to a model, creates an isomorphic copy of the original which is an instance of the same class as the original. The primary classes (*atomic-models, digraph-models*, and the specializations of *kernel-models*) require their own versions of the *make-new* method since each has features that are unique to itself. Note that sub-classes of these primary classes, if they do not add additional structure, can inherit the *make-new* method from the primary class.

Since *coupled-models* instances are hierarchical in structure, the *make-new* method must be recursive in the sense that components at each level must replicate themselves with their own *make-new* methods. The second method, *make-class*, when sent to a model, creates a class definition with the original model as template. Instances created in such a class will be isomorphic to the original. However, in contrast to the effect of *make-new*, such instances are mem-

bers of a different class from the original. Description of *make-class* is not the scope of this paper (see (Kim 1988) for details).

To explain construction of complex, hierarchical models using the method *make-new*, consider creation of an instance of *hypercube-models*, a subclass of *kernel-models*. The hypercube model would have a number of components connected together in hypercube coupling, each of which may be a complex, hierarchical model created from some class in DEVS-Scheme. A facility *make-hypercube* and a method *make-members* in DEVS-Scheme are used to create such a hypercube model:

(make-hypercube PCS)
(send hc-PCS make-members 'ibm-pc 3)

where, PCS is an existing class, ibm-pc is the basic name of each member, and 3 is dimension of the hypercube.

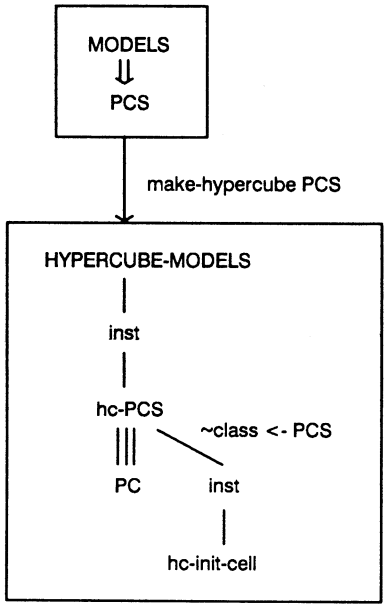

Fig. 1. Facility make-hypercube

As shown in Fig. 1, the first command makes a hypercube model hc-PCS with *kernel-class* PCS and hc-init-cell, an instance of PCS. Note that PCS can be any subclass in DEVS-Scheme including any subclass of *kernel-models* class itself. The second command causes the sequence:

(send hc-init-cell make-new 'ibm-pc1)
(send hc-init-cell make-new 'ibm-pc2)
.....
(send hc-init-cell make-new 'ibm-pc8).

The sequence creates above objects ibm-pc0, ibm-pc1,...., and ibm-pc8 each *isomorphic* to hc-init-cell (hence to each other) and belonging to the same class as hc-init-cell, namely the *kernel-class* PCS. Recall that *atomic-models, digraph-models,* and the specialization of *kernel-models* have their own versions of *make-new* method. For example, if hc-init-cell is a digraph model, hc-init-cell sends a message for *make-new* to each component. Each component then recursively applies its own version of *make-new*. Applying *make-new* method for a coupled model in such a recursive way is an advantage of object-oriented nature of DEVS-Scheme; the coupled model (and its components) do not have to know how the method *make-new* of each component (and components of each component) works. All that a coupled model needs to do, when it receives a message for the *make-new*, is to recursively forward the message to its components.

Methods for testing model *isomorphism* provides the key requirement in designing replication methods such as the *make-new* above. Isomorphism, meaning structure preservation, embodies a set of criteria which are largely determined by the structure of the class to which it is applied. However, such criteria are not necessarily uniquely determined by the underlying structure and there is some liberty remaining to the designer. To make sense, however, the chosen criteria must render isomorphism as an equivalence relation, i.e., reflexive, symmetric, and transitive.

3. METHOD *make-new* in DEVS-Scheme

The method, *make-new*, in DEVS-Scheme provides a means of creating *isomorphic* copies of a model. All models classes have the method *make-new*, but its operation for one model class is fundamentally different from that for the others.

An instance of *atomic-models* creates an *isomorphic* model by creating a new model from the class whereby the instance was created. Since the two models are created from the same class, their instance variables have the same structure and their methods are the same. In addition, the original model makes copies of values of its own instance variables to those of the new model.

The *make-new* for *coupled-models* is different from that of *atomic-models*. The fundamental difference is that *isomorphic* copies of a coupled model require copies of both its *components* and its *coupling scheme*. The *make-new* of a digraph model is shown in Fig. 2, where d1 is created from d. Two instances, d and d1, are created from the same class, DS. Each component of d applies method *make-new* to create an *isomorphic* copy of the corresponding component of d1. The coupling scheme in the digraph model consists of external and internal coupling schemes each of which is represented by a list of pairs of ports. Each such port contains a pairs of names of models and names of ports.

Make-new of a kernel model is different from that of a digraph model. *Init-cell* of the kernel model, an instance variable of the kernel model, is used to create its components, all of which are *isomorphic* to its *init-cell*. In regard to copying the coupling schemes, each specialization of *kernel-models* is slightly different. Internal coupling

schemes for all sub-classes of the *kernel-models* class are maintained by a table called *out-in-coup* table. For external coupling schemes, a hypercube model and a cellular model have some variables to be copied, while a broadcast model does not have such variables.

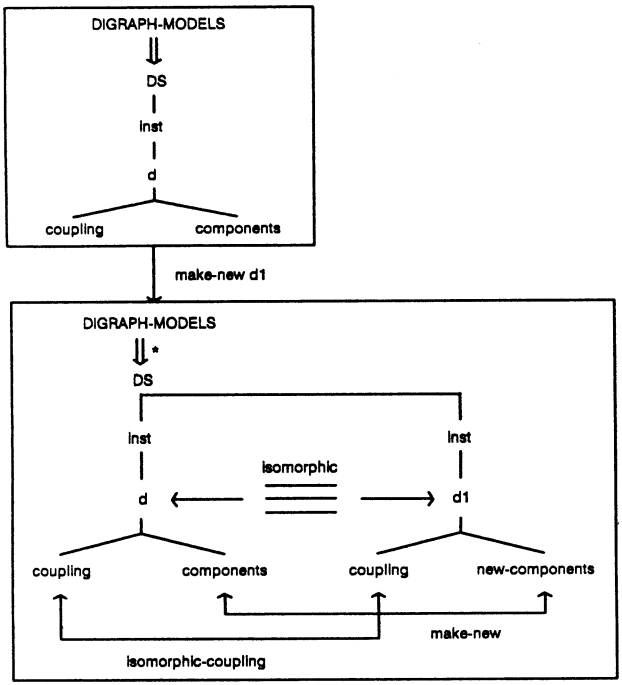

Fig. 2. Make-new for digraph-models

4. ISOMORPHISM BETWEEN MODELS

An *isomorphism* is a structure preserving mapping from one system to another, that constitutes a one-to-one correspondence between two systems. Such an isomorphism asserts that the systems have the same structural features recognizable at the level at which the morphism holds. Thus, the isomorphism provides an approach to checking isomorphic replication of complex, hierarchical models in DEVS-Scheme.

Two atomic models (in normal form) are said to be *isomorphic* if the components of structures of two models are such that

(1) set of *state variables* have identical names;
(2) *internal transition functions* are identical;
(3) *external transition functions* are identical, and
(4) *output functions* are identical.

Condition (1) establishes a one-one correspondence between the state sets of the models while the remaining conditions, establish the preservation of the defining functions of the models as dictated by the DEVS formalism (Zeigler 1984).

Following the isomorphism definition for DEVS multi-component models (Zeigler 1984), two coupled models would be *isomorphic* if there exists one-to-one correspondence between components of the coupled models such that

(1) coupling schemes of two coupled models are isomorphic, and
(2) corresponding component models are isomorphic.

The definition is recursive in that components of the coupled models can themselves be coupled models. The definition of the isomorphism of components recursively refers to the definition itself until the components are either atomic models or their specialized models.

The coupling schemes (sets of pairs of ports) of two coupled models are said to be isomorphic if there is one-to-one correspondence between two sets such that

(1) the coupled models have input and output ports with the same names;
(2) corresponding components have corresponding ports, and
(3) the connectivity relations of corresponding ports are preserved.

The definition above is employed for *digraph-models*. However, for *kernel-models* a looser one was implemented. Recall that members of a kernel model are created from *init-cell* such that all members are intended to be *isomorphic* to it. Thus, to check isomorphism between two kernel models, we check isomorphism between their *init-cells*. We do not, however, require that the models have the same number of components. Since the coupling schemes for *kernel-models* are given by finite tables, these can be checked appropriately without regard to the existing components. By ignoring the number of components currently in existence, only the generative capabilities of two kernel models need be the same for them to be considered isomorphic.

Each primary class of models in DEVS-Scheme has its own method, *isomorphic?*, to check isomorphism at its level. For example, (send a1 isomorphic? a) checks whether two atomic models, a1 and a, are isomorphic each other. Isomorphism between two digraph models is based on a correspondence between their components. Finding such a correspondence in general is computationally intractable. However, we do not develop a general algorithm to compute such correspondence. Instead, we develop an algorithm to check the isomorphism between coupled models in which correspondence of their components can be computed by listing them in an appropriate order.

In DEVS-Scheme, we know the correspondence of the components of two models if they were created from a common ancestor using method *make-new*. For example, *make-new* of a digraph model M creates a new digraph model M' in such a way that M visits each component and creates corresponding new components recursively. Therefore, the order in which M visits its components is identical

to the order in which new components of M' are created. Hence, the list of components obtained by visiting children of M' corresponds to the list of those obtained by visiting children of M in the same order as M'. In DEVS-Scheme, method *make-new* visits components of M in preorder and creates corresponding components in such order. Using this preorder to establish the correspondence makes isomorphism checking feasible.

5. ISOMORPHISM ALGORITHM

Based on the definition above on isomorphism between two atomic models, we easily can develop an algorithm ,or a method, to check isomorphism between the two. Since state variables, internal transition function, external transition function, and output function are instance variables of *atomic-models* class, the method *isomorphic?* for atomic-models is a sequence of comparisons for pairs of instance variables of two models (Fig. 3 (a)).

With the definition of isomorphism between two coupled models and the assumption of known correspondence of their components, we now can develop algorithms, or methods, to check isomorphism. To take an advantage of *inheritance* mechanism provided by object-oriented nature of DEVS-Scheme, we develop different methods for each specialized models of coupled models. We can implement the algorithms such that common algorithms between a class and its subclasses are to be shared. The following realizes the idea.

To check the isomorphism of coupled models, we need to check both the components isomorphism and the coupling isomorphism, which are common to all specialized classes of *coupled-models*. Thus a method *isomorphic?* (Fig. 3 (b)) is designed for *coupled-models* so that *digraph-models* and *kernel-models* may use the method. The method applies two methods: *isomorphic-components?* and *isomorphic-*

```
Method isomorphic? (m)
 (and
   (= (state-var-sets this-model) (state-var-sets m))
   (= (int-trans-fn this-model) (int-trans-fn m))
   (= (ext-trans-fn this-model) (ext-trans-fn m))
   (= (output-fn this-model) (output-fn m)))
End
```

Fig. 3. (a) Method isomorphic? for atomic-models

```
Method isomorphic? (m)
 return (and (isomorphic-components? this-model m)
             (isomorphic-coupling? this-model m))
End
```

Fig. 3. (b) Method isomorphic? for coupled-models

```
Method isomorphic-components? (m)
 establish a cor-table comprising list of pairs (mi, mi')
   such that mi is a component of this-model and
   mi' is a corresponding component of m
 for each pair (mi, mi') in the cor-table
   apply isomorphic? (mi mi')
 if all pairs are isomorphic
   return true
 else return false
End
```

Fig. 3 (c) Method isomorphic-components? for digraph-models

```
Method isomorphic-components? (m)
 isomorphic? (init-cell-of-this-model init-cell-of-m)
End
```

Fig. 3 (d) Method isomorphic-components? for kernel-models

```
Method isomorphic-coupling? (m)
  establish a cor-table for this-model and m
  get coup1, coupling scheme of this-model
  get coup2, coupling scheme of m
  if length(coup1) < >length(coup2)
    return false
  for each pair in coup1
    find corresponding element in coup2 by substituting
    model name in the pair to corresponding model
    name in the cor-table and remove it from coup2
  if coup2 is empty
    return true
  else return false
End
```

Fig. 3 (e) Method isomorphic-coupling? for digraph-models

```
Method isomorphic-coupling? (m)
 get out-in-coup1, a table for internal coupling scheme
   for this-model
 get out-in-coup2, a table for internal coupling scheme
   for m
 get ext-coup1, a set of instance variables for external
   coupling scheme for this-model
 get ext-coup2, a set of instance variables for external
   coupling scheme for m
 if (and (ext-coup1 is the same as ext-coup2)
         (out-in-coup1 is equivalent to out-in-coup2))
   return true
 else return false
End
```

Fig. 3 (f) Method isomorphic-coupling? for kernel-models

coupling?. The method *isomorphic-components?* of *digraph-models* is different from that of *kernel-models*.

The method *isomorphic-components?* of digraph-models checks the isomorphism between corresponding components of two digraph models. However, in *kernel-models*, we can check isomorphism a much simpler way. Members of a kernel model are created by *init-cell* by use of method *make-new*, which ensures that all members are isomorphic to *init-cell*. Thus, checking the isomorphism between corresponding components of two kernel models is equivalent to checking the isomorphism between init-cells of the two models. Therefore, the methods *isomorphic-components?* for *digraph-models* and *isomorphic-components?* for *kernel-models* are designed separately as shown in Fig. 3 (c) and (d), respectively.

Since different specialized models of coupled models have different coupling schemes, the method *isomorphic-coupling?* should be different for all specialized models. Four different methods of *isomorphic-coupling?* have been developed for each such specialized model. Fig. 3 (e) and (f) show the method *isomorphic-coupling?* for *digraph-models* and *kernel-models*, respectively.

6. CONCLUSIONS

Isomorphic replication of a complex, hierarchical model is a powerful means to develop modular, hierarchical discrete-event models using the DEVS-Scheme environment. However, the environment should provides a means to check such isomorphic copies before using them. Methods *make-new* and *isomorphic?* in DEVS-Scheme are tools to make isomorphic copies and to check isomorphism, respectively. The Object-oriented nature of DEVS-Scheme makes it easier to develop such methods for different classes and their sub-classes using inheritance and polymorphism.

REFERENCES

Concepcion, A I and Zeigler, B P. 1988. "DEVS formalism: A framework for hierarchical model development." *IEEE Trans. on Software Engineering* Vol 14 No 2 (Feb) pp 228-241.

Kim, Tag Gon and Zeigler, B P. 1987. "The DEVS Formalism: Hierarchical, Modular System Specification in an Object Oriented Framework." *Proc in 1987 Winter Computer Simulation Conference*, Atlanta, GA, (Dec.) pp. 559-566.

Kim, Tag Gon. 1988. "A knowledge-Based Environment for Hierarchical Modelling and Simulation." Ph.D. Thesis, Dept. of ECE, University of Arizona, Tucson, AZ, (May).

Kim, Tag Gon and Zeigler, B P. 1990. "The DEVS-Scheme Simulation and Modelling Environment." in *Knowledge Based Simulation: Methodology and Application*, Paul A. Fishwick and Richard B. Modjeski, eds. Springer Verlag Inc.

Zeigler, B P. 1976. *Theory of Modelling and Simulation*. John Wiley, NY (Reissued by Krieger Pub. Co., Malabar, FL. 1985).

Zeigler, B P. 1984. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, Orlando, FL.