Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

The DEVS Formalism: Hierarchical, Modular Systems Specification in an Object Oriented Framework

Tag Gon Kim, Bernard P. Zeigler

Department of Electrical and Computer Engineering
The University of Arizona
Tucson, Arizona 85721

ABSTRACT

This paper describes a realization of DEVS (Discrete Event System Specification) formalism in an object-oriented programming environment, SCOOPS of PC-Scheme. The realization, DEVS-Scheme, is a powerful environment combining simulation modelling with AI techniques, which supports hierarchical, modular specification of discrete event models. We will describe the taxonomical organization of classes in DEVS-Scheme in which kernel-models classes will be emphasized. As an example, we will illustrate specification of kernel models including broadcast models, hypercube models, and cellular models. Finally, implementation of isomorphism checks between pairs of models in DEVS-Scheme will be discussed. Thus, DEVS-Scheme represents a significant step toward implementing system theoretic based formalisms and operations.

1. Introduction

The compatibility between object-oriented programming paradigms and discrete event world view formalisms has been well noted (O'Keefe, 1986; Zeigler, 1987). In such programming systems, an object, an instance of class, is a package of data structures and associated operations, usually representing a real world counterpart. Naturally these programming paradigms incorporate AI knowledge representation schemes within simulation models, resulting in so-called knowledge based simulation systems (Reddy et. al. 1986; Klar, 1986; Kerkoffs et. al. 1986). Such schemes can be used not only to organize information about the nature of the objects involved in the simulation, but also to model intelligent agents within components themselves(Davis, 1986; Robertson, 1986; Zeigler, 1987).

DEVS-Scheme is an environment for specification of hierarchical, modular discrete event models and simulation in LISP-based object-oriented framework (Zeigler, 1986a; Kim, 1987). In contrast to existing knowledge based simulation systems, DEVS-Scheme is based on the DEVS formalism, a theoretically well grounded means of expressing hierarchical, modular discrete event models(Zeigler, 1976, 1984, 1986b; Concepcion & Zeigler, 1987).

This paper first describes the taxonomical organganization of classes in DEVS-Scheme with an emphasis on kernel-models, a generalized class whose sub classes provide powerful means of defining uniform networks of isomorphic models. Next we discuss specification of both atomic and coupled DEVS models based on such classes. This facilitates the development of hierarchical, modular discrete event systems. Finally, creation of derived classes and isomorphism(Oren, 1984) in generation of new models in DEVS-Scheme are presented.

2. Organization of Classes in DEVS-SCHEME

DEVS-Scheme, an implementation of DEVS formalism, is principally coded in SCOOPS, the object-oriented superset of PC Scheme. The taxonomical hierarchy of classes in DEVS-Scheme shown in Figure 1 represents general classes and their specialized classes with variables attached. In what follows after a brief review, we shall focus on some of the main class variables and methods of kernel-models.

Entities is the universal class which provides tools for manipulating objects not only in these classes but also through inheritance,in any of their subclasses. Details of the such general facilities including mk-ent, show-class, name->entity are in (Zeigler, 1986a). Models classes and processors classes are the main subclasses of entities.

2.1 Class Models

As shown in Figure 1, models class is further specialized into the major classes atomic-models and coupled-models, which in turn is specialized into digraph-models and kernel-models. Kernel-models is also specialized into more specific classes, broadcast-models, hypercube-models, and cellular-models. Objects of such classes realize either atomic DEVS or coupled DEVS specified by our formalism.

2.1.1 Class Atomic-models

Atomic-models realizes the atomic level of the DEVS formalism by use of its variables and methods which correspond to components of 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, respectively when they are evaluated. We shall describe specification of atomic model in section 3.1. Methods of atomic-models and their examples are described in detail in (Zeigler, 1986a, 1987b).

2.2.2 Class Coupled-models

Coupled-models is the major class which embodies the hierarchical model composition of the DEVS formalism. Digraph-models and kernel-models are specializations which enable specialization of coupled models in specific ways. In DEVS formalism, we define coupled-models by specifying its component models (also called children) and desired communication links among the children. Instance variables corresponding to children and coupling relations, and methods which manipulate the variables realize the formalism. Methods, get-children, get-

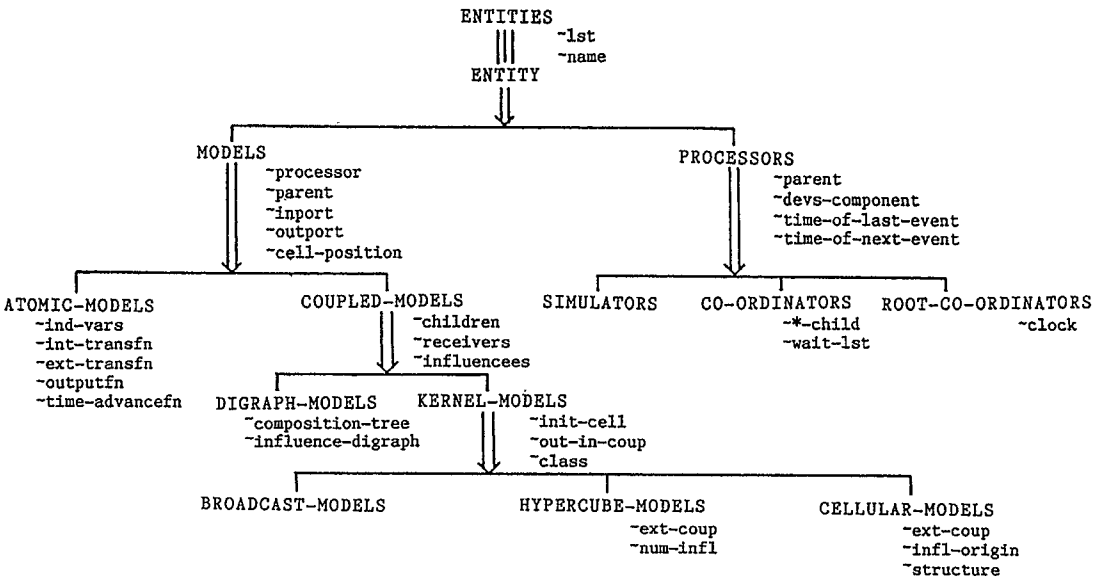


Figure 1. Class Inheritance Structure of DEVS-Scheme.
Capital letters : Classes
Low case : instance/class variables

influencees, get-receivers, and translate are available for any specializations of coupled-models (Zeigler, 1986a).

2.2.3 Class Digraph-models

Digraph-models, a specialized class of coupled-models, is composed of 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.

2.3.4 Class Kernel-models

In contrast to digraph-models, instances of kernel-models are coupled models whose components (children) are isomorphic models (we shall define isomorphism in detail in section 4.). Method make-members creates the isomorphic children using an instance variable of kernel-models called init-cell. The children are all members of the same class called the kernel class of the kernel-models. The modeller can specify the coupling scheme of a kernel model either explicitly or implicitly depending on the specialization of kernel-models. Whether internal coupling scheme is implicit or explicit, it is uniform for all children of a kernel model.

An instance variable, out-in-coup of kernel-models tells how to translate output ports to input ports of the internally coupled children. Different specialized classes of kernel-models realize different internal and external coupling scheme. We have defined three coupling schemes: broadcast coupling, hypercube coupling and cellular coupling realized by the sub classes broadcast-models, hypercube-models, and cellular-models, respectively. Three facilities, make-broadcast, make-hypercube, and make-cellular create a broadcast model, a hypercube model and a cellular model, respectively.

2.3.5 Class Broadcast-models

Broadcast-models is a simple but important subclass of kernel-models. All members (children) of a broadcast coupled model communicate directly with each other and with the outside world. Thus methods, get-children, get-influencees and get-receivers uniquely determine children, influencees, and receivers of the coupled model, respectively. For example, influencees of any child of a broadcast model are all children of the broadcast model except itself. The receivers of a broadcast model are all the children of the broadcast model. The only additional information on the coupling scheme of broadcast-models is how to translate output ports to input ports. Method add-port-pair inherited from kernel-models enables the modeller to specify pairs of ports for internal coupling by inserting the pairs in out-in-coup table which is an instance variable of the class.

2.3.6 Class Hypercube-models

Hypercube-models is a specialization of kernel-models whose children are 2**n instances of the kernel class, where n is the dimension of the hypercube. To specify positions of all the children in the hypercube, each instance has cell-position as an instance variable. Any n-dimensional hypercube configuration consists of 2 isomorphic (n-1) dimensional hypercube configurations, representing a well known mutiprocessor architecture.

In a hypercube model the modeller explicitly

specifies both internal and external coupling schemes. The external coupling can be either broadcast or origin-only. In broadcast external coupling, inputs and outputs of all the children in a hypercube coupled model are coupled to input and output of the hypercube coupled model, respectively. On the other hand, only the cell at origin is coupled to the hypercube coupled model in the origin-only external coupling scheme. Method set-ext-coup selects one of the coupling schemes. Referring to internal coupling, the modeller should specify the number of influencees for each child, and corresponding pairs of ports. Like broadcast-models, out-in-coup table holds the pairs of ports.

The range of the number of influencees is from zero to dimension of a hypercube. The Hamming distance between cell positions of a model M and any influencees of M is one. For example, in a 3-dimensional hypercube model, 3 influencees of a cell at (0 0 0) are (1 0 0), (0 1 0), and (0 0 1), and those of a cell at (1 1 1) are (0 1 1), (1 0 1), and (1 1 0) and so on.

2.3.7 Class Cellular-models

A specialization of kernel-models, cellular-models provides for coupling of a finite or infinite set of geometrically located cells, each of which is connected to other cells in a uniform way. The cellular-models described here realizes the formalism for discrete event cell space models (Zeigler, 1976, 1984).

Influencees of a cell in a cellular model can be computed from the influencees pattern of the origin cell, infl-origin, for both fixed structure and variable structure cellular-models. For a fixed structure cellular model, the influencees of a cell C is set of existing cells in cell space whose cell position is the member of influencees pattern of C. Therefore a cell may have no influencees if no existing cells in cell space have the cell positions of the influencees. However, for a variable structure cellular model, non existing cells in the cell space at the cell positions of the influencees will be created in simulation time. Therefore, all cells have the same number of influencees. An instance variable, structure is used to specify one of these structure types.

In a 3-dimensional fixed structure cellular model, for example, if the influencees pattern of origin is {(1 1 1) (2 2 2)}, then influencees of the origin cell are all existing cells in the cell space whose cell position are either at (1 1 1) or at (2 2 2). Uniformity of influencees pattern then requires that influencees of a cell at (0 1 1) in the cellular model are a cell at (1 2 2), and a cell at (2 3 3) if they exist. Both internal and external coupling scheme of cellular-models can be specified in similar way to those of hypercube-models. Details are in (Kim, 1987).

2.2 Class Processors

Simulation of DEVS models are based on the abstract simulator principles developed as the part of theory (Zeigler, 1984; Concepcion, 1984). The principles are implemented by three specialized classes of processors namely simulators, co-ordinators, and root-co-ordinators as shown in Figure 1 . A root-co-ordinator manages the overall simulation and is linked to co-ordinator of the outmost coupled model. On the other hand, simulators and co-ordinators handle atomic-models and coupled-models, respectively. A facility make-pair creates both an atomic model(a coupled model) as well as a simulator(a co-ordinator) assigned to it. Simulation proceeds by means of messages passed among above three specialized processors which carry information concerning internal events (*-message) and external events (x-message), as well as data needed for synchronization. Details of simulation process for DEVS models can be found in (Zeigler, 1986a)

3. Model Specification in DEVS-Scheme

3.1 Atomic Model Specification in DEVS-Scheme

Discrete event models specification is based on set-theoretic formalism and has been implemented by DEVS-Scheme in LISP-based programming environment (Zeigler, 1986a; Kim, 1987). The environment implements components of the structure of an atomic DEVS in the formalism. To specify atomic DEVS, the modeller are to specify the components accordingly in term of Scheme functions. Details of atomic model specification in DEVS-Scheme along with examples are in (Zeigler, 1986a, 1987b; Kim, 1987).

3.2 Coupled Model Specification in DEVS-Scheme

A Coupled model can be characterized by specifying its component models as well as its coupling scheme consisting of internal and external coupling schemes.

3.2.1 Digraph Model Specification in DEVS-Scheme

Specification of a digraph model starts with specifications of components followed by coupling scheme. Recursively, components can be specified by their components specification and coupling scheme. Recursive specification of coupled models in DEVS-Scheme is based on a closure property of formalism under coupling(Zeigler, 1984). Examples of specification of digraph models are in (Zeigler, 1986a, 1987b).

3.2.2 Kernel Model Specification in DEVS-Scheme

As indicated before, make-broadcast, make-hypercube, and make-cellular create instances of the respective classes. When created, each of above coupled models has a init-cell from which its children will be created. After creating any of the above coupled models, the modeller may specify coupling schemes and other information needed using various methods provided by kernel-models and the specialization class.

The details are illustrated in the following examples.

1. (make-broadcast MS) creats br-MS, a coupled model of class broadcast-models, and also creates init-cell, an instance of class MS. The init-cell is an instance variable of br-MS and will be used to create members of br-MS. At the same time class MS is attached to br-MS as an instance variables. Note that class MS can be any specialized class of models including broadcast-models itself. Figure 2 shows this facility.

2. (send br-MS make-members 'M 3) creates three children M0, M1, and M2 of br-MS. (Figure 3)

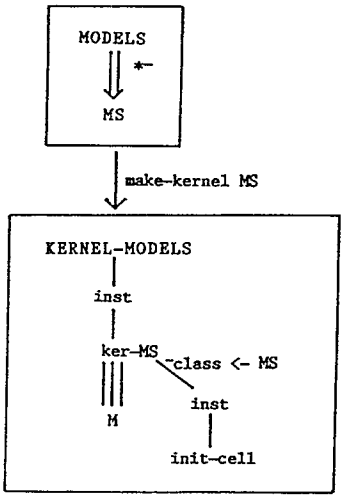


Figure 2. Facility Make-kernel Models.
"-kernel" can be one of
{"-broadcast" "-hypercube" "-cellular"}

*- : MS is any sub classes of MODELS
except MODELS
<- : MS is assigned to instvar class

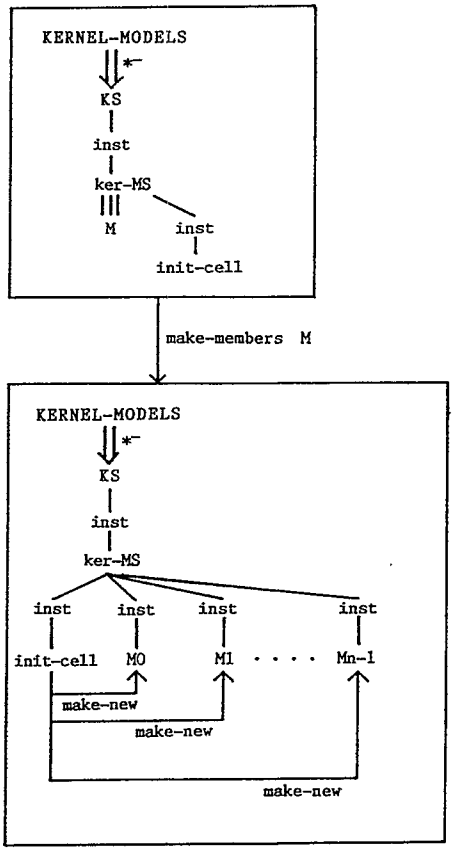


Figure 3. Creation of Components in Kernel Models.

3. (send br-MS add-port-pair 'out 'in) adds the port pair ('out 'in) to out-in-coup ,an instance variable of br-MS. Henceforth, when a component places an output on port 'out, it will be sent to all the 'in ports of its brothers.

4. (make-hypercube MS) is similar to (make-broadcast MS) and creates hc-MS and init-cell.

5. (send hc-MS make-members 'M 3) creates 4 members in 3-dimensional hypercube, namely M0 at (0 0 0), M1 at (0 0 1), M2 at (0 1 0), M3 at (0 1 1), M4 at (1 0 0), M5 at (1 0 1), M6 at (1 1 0), M7 at (1 1 1).

6. (send hc-MS set-ext-coup 'broadcast) couples hc-MS to each of its components.

7. (send hc-MS set-num-infl 3) set influencees of each members. In this case, each component has three influencees which are the three closest neighbors in the hypercube.

8. (send hc-MS add-port-pair 'out 'in) is analogous to step 3. Step 4 to 8 result in Figure 4.

9. (make-cellular MS) creates ce-MS as in step 4.

10. (send ce-MS make-init-active-cells 'M '((-1 1) (0 1))) creates 6 components in 2-dimensional cell space, namely M0 at (-1 0), M1 (-1 1), M2 at (0 0), M3 at (0 1), M4 at (1 0), and M5 at (1 1).

11. (send ce-MS set-structure 'variable) sets the structure type of ce-MS to variable where new cells can be created in simulation time as needed.

12. (send cd-MS set-infl-origin '((2 2) (3 5))) sets influencees pattern of origin cell to cell-position of (2 2) and (3 5). Influencees of a cell at (a b) are the cells located at (a+2 b+2) and (a+3 b+5) if they exist.

4. Creation of New Classes in DEVS-Scheme

A class from which instances are created may provide a kernel class for more than one instance of kernel models. Such classes cannot be identical but must be isomorphic copies of each other. To facilitate this copying, a method make-class provides sub classes of a class and copies the class structure of the parent class as shown in Figure 5.

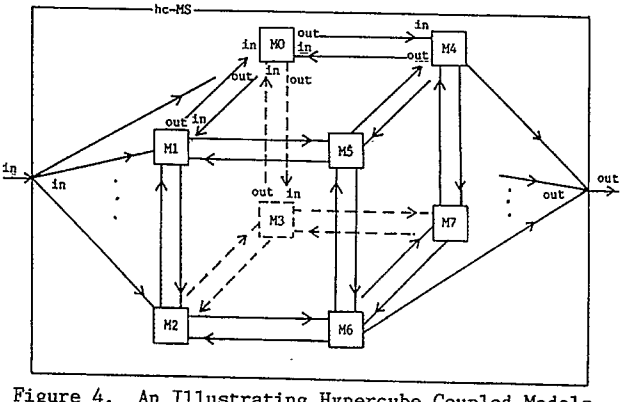
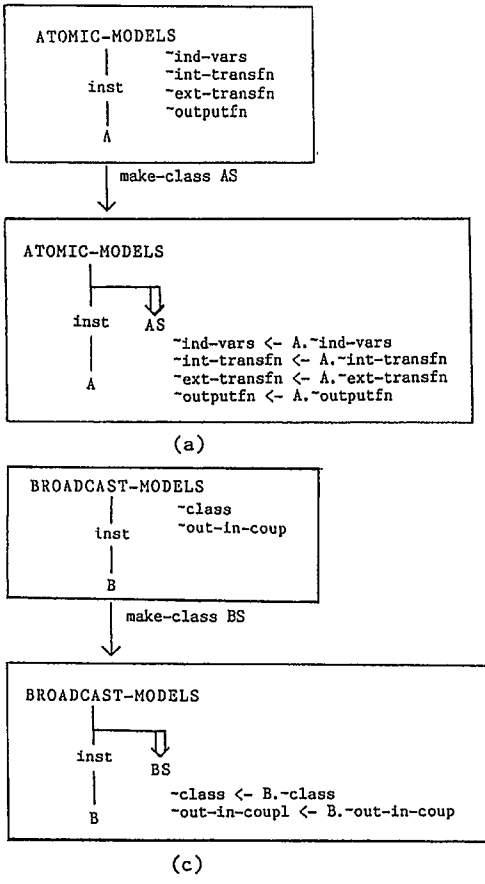


Figure 4. An Illustrating Hypercube Coupled Models.
Class MS can be Any Class Under Models
Class in Figure 1 Including Hypercube



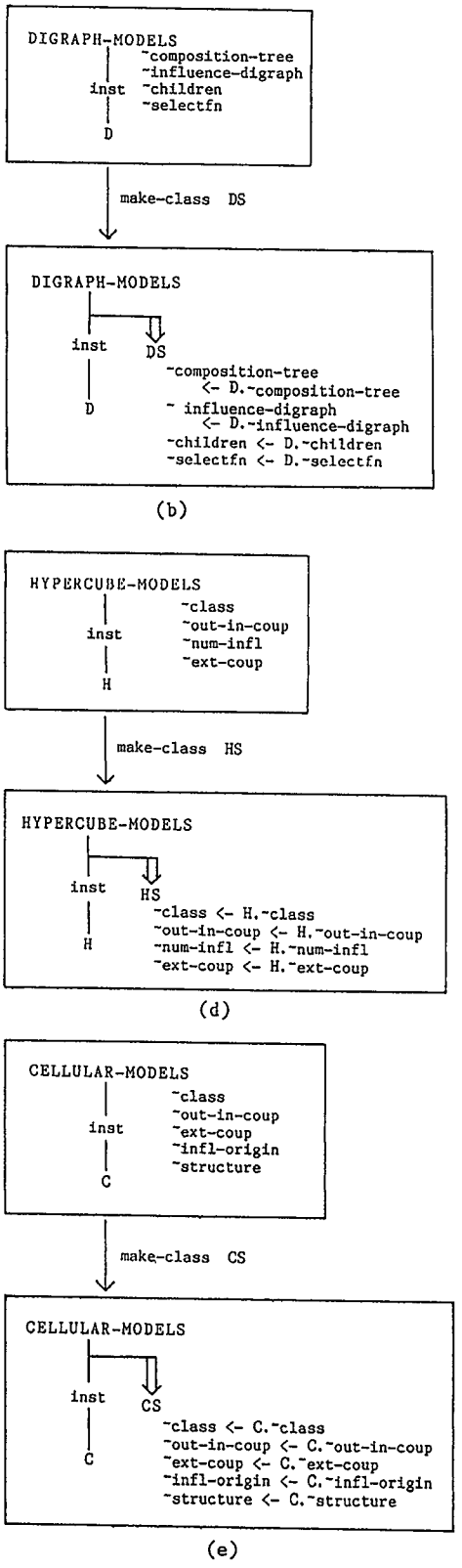


Figure 5. Creation of New Classes. (a) Atomic Models (b) Digraph Models (c) Broadcast Models (d) Hypercube Models (e) Cellular Models

Method make-class of a atomic model A creates a class AS which is a subclass of atomic-models. As shown in Figure 5 (a), the method copies structure of AS from that of atomic-models by copying four instance variables ind-vars, int-transfn, ext-transfn, and outputfn from atomic models. Similarly, Figure 5 (b) shows make-class of a digraph model where four instance variables of digraph-models, composition-tree, influence-digraph, children, and selectfn are copied from D into its subclass DS. Figure 5 (c), (d) and (e) show make-class of a broadcast model, a hypercube model, and a cellular model, respectively. Again, make-class of each class in the figure copies instance variables of their sub classes from their parent class. Note that an instance of a class C can create any number of sub classes of C such that structure of each sub classes inherits the class structure from its parent C. Each such sub class so created can be a kernel class of instances of a kernel model.

5. Creation of New Models and Isomorphism

A method, make-class, provides a means of creating an isomorphic copy of a model for any specialized model of models class.

Two atomic models are said to be isomorphic if four components of structures of two models are such that
(1) sets of state variables are identical
(2) internal transition functions are identical
(3) external transition functions are identical
(4) output functions are identical.

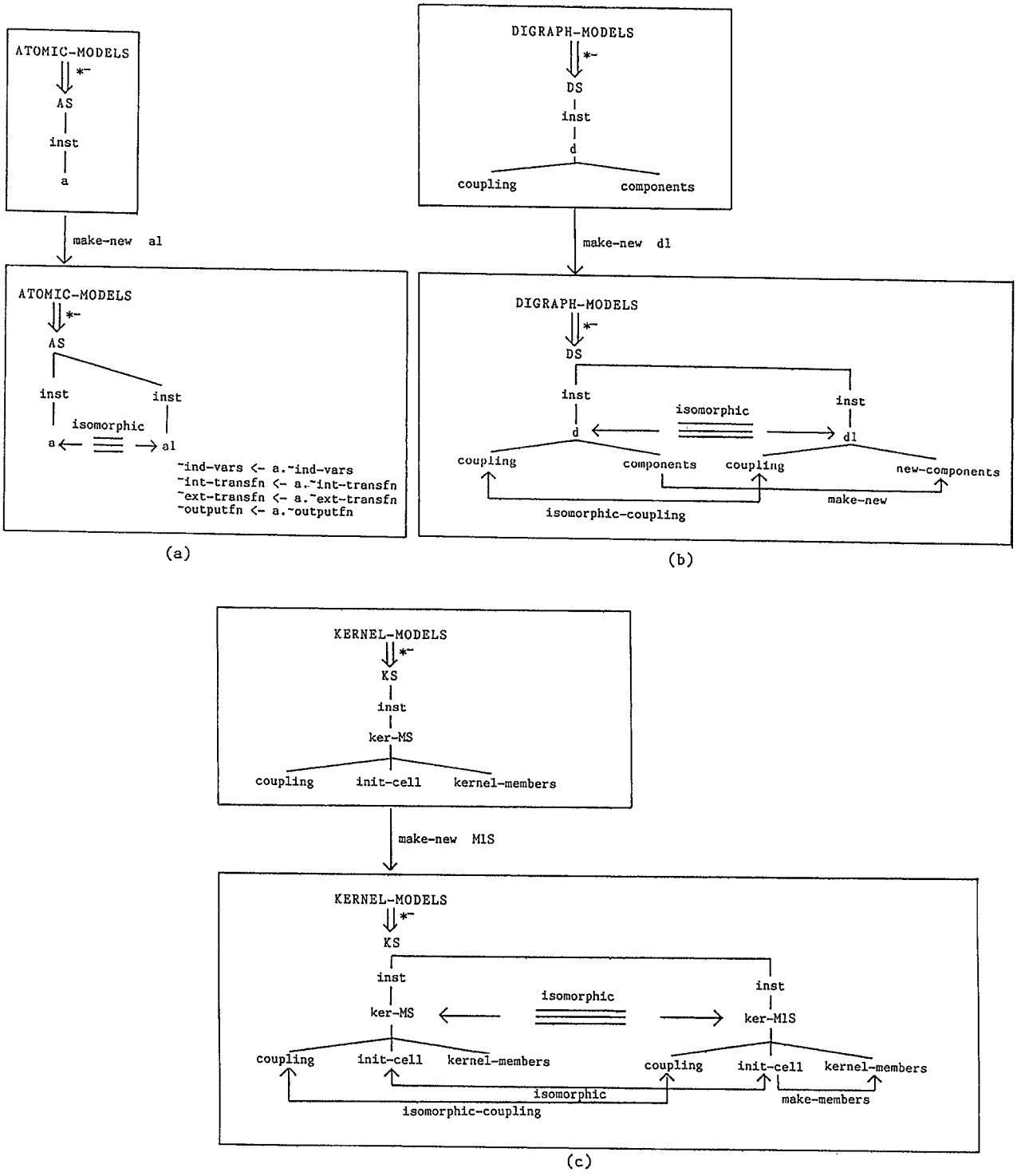


Figure 6. Creation of New Models. (a) Atomic Models
(b) Digraph Models (c) Kernel Models

Methods are provided to check isomorphism between two models. Each specialized model has its own methods to check isomorphism in its level. For the simplest case, method isomorphic? of a atomic-model checks isomorphism between two atomic models, say a and a1 by sending a message (send a isomorphic? a1) to a or a message (send a1 isomorphic? a) to a1 as shown in Figure 6 (a).

We can define isomorphism between two coupled models as follows:

Two coupled models are said to 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
(2) corresponding component models are isomorphic

The coupling schemes(sets of pairs of ports) of two coupled models with correspondence between their components are said to be isomorphic if there is one-to-one correspondence between two sets such that
(1) the coupled models have identical ports
(2) corresponding components have corresponding ports
(3) relations of corresponding ports are identical

More specifically, a method, isomorphic-coupling? of digraph-models contains following Scheme code:

```
;;
;; method isomorphic-coupling?
;;    for digraph-models m1 and m2
;;
;;  coupl : list representing coupling scheme of m1
;;  coup2 : list representing coupling scheme of m2
;;  each element of the above lists is a list of :
;;     name of source model (src)
;;     name of destination model (dest)
;;     list of pairs of ports (ports-pairs)
;;
;;  cor-name-tab is a table containing pairs of
;;    names of corresponding models
;;       where src-> corresponds to src
;;
(let rep (
          (coupl (send m1  get-coupling))
          (coup2 (send m2  get-coupling))
         )
  (cond
    ((not (equal? (length coupl) (length coup2)))
       #!false)
    ((and (null? coupl) (null? coupl))
       #!true)
    (else
      (let* (
             (next-coupl (car coupl))
             (next-coup2 ())
             (src (car next-coupl))
             (dest (cadr next-coupl))
             (ports-pairs (caddr next-coupl))
             (src->  ;;  corresponding src
                (table-look-up cor-name-tab src))
             (dest-> ;;  corresponding dest
                (table-look-up cor-name-tab dest))
            )
        (set! next-coup2
          (list src-> dest-> ports-pairs))
        (rep (cdr coupl) (remove next-coup2 coup2))
      ))
  )
)
;;
```

To check isomorphism between two coupled models requires to know correspondence between their components. 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.

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, to check isomorphism between two kernel models, we check isomorphism between init-cells of the models and check the coupling schemes of the two. Figure 6 (c) show method make-new for kernel-models.

6. Summary

We have described realization of Discrete Event System Specification(DEVS) formalism in a LISP-based programming environment. DEVS-Scheme supports hierarchical, modular models specification of discrete event models. Class organization of DEVS-Scheme has been presented and kernel-models classes has been discussed in detail. We have presented creation of new classes in DEVS-Scheme, and isomorphism checking between pairs of models in DEVS-Scheme. These operations demonstrate the power of object-oriented paradigms such as SCOOPS to realize powerful concepts of systems theory as envisioned by (Oren, 1984). Conversely, they suggest new structuring concepts requiring systems theoretic formalization.

Reference

Concepcion, A. I. (1984), "Distributed Simulation on Multiprocessors: Specification, Design and Architecture", Doctoral Dissertation, Wayne State University, Detroit.

Concepcion, A. I. and Zeigler, B. P. (1987), "DEVS Formalism: A Framework for Hierarchical Model Development", IEEE Trans. Soft. Engrg., (accepted for publication).

Davis, P. K. (1986), "Applying Artificial Intelligence Techniques to Strategic-level Gaming and Simulation", In: Modelling and Simulation Methodology in the Artificial Intelligence Era, Elzas, M.S., Oren, T. I., Zeigler, B. P. (Eds), North Holland, Amsterdam.

Kerckhoffs, E. J. H. and Vansteenkiste G. C. (1986), "The impact of Advanced Information Processing on Simulation - an Illustrative Review", Simulation, 46:1, 17-26.

Kim, Tag Gon (1987), "A Knowledge-Based Environment for Hierarchical Modelling and Simulation", Doctoral Dissertation, University of Arizona, Tucson.