Hierarchical System Concepts for Simulation of High Autonomy Systems

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

ABSTRACT

A high autonomy system is an intelligent system with the self-determination power. Such a system must perform a variety of functions such as reasoning, planning, sensing, control, and other activities necessary to achieve predefined system objectives over an extended period of time under uncertainties in its environment. Simulation modelling can be a powerful tool for design of such complex autonomy systems. This paper describes the hierarchical, modular system concepts for simulation of high autonomy systems with an example of an intelligent control system called the AIDECS (AI-based, Distributed Environmental Control System). The paper emphasizes on development of methods, in an object-oriented manner, for isomorphic replications of complex hierarchical structures as a means of constructing hierarchical models. We show how polymorphism and inheritance in the object-oriented programming can be exploited to develop such methods in DEVS-Scheme, a realization of the DEVS formalism in a LISP-based, object-oriented framework.

Key Words: High Autonomy System, Hierarchical System, Object-oriented Programming, DEVS-Scheme, Polymorphism, Models Isomorphism.
1. INTRODUCTION

A high autonomy system is an intelligent system with the self-determination power. Such a system must perform a variety of functions such as reasoning, planning, sensing, control, and other activities necessary to achieve predefined system objectives over an extended period of time under uncertainties in its environment. The architecture of such a system may employ at least three control layers, namely, an execution layer for control and sensing, a management layer for high-level decision-making to achieve overall goals, and a coordination layer for linking the two layers (Saridis, 1983).

At the intersection of artificial intelligence, automatic control, and operations research, design of autonomy systems clearly requires the tools of artificial intelligence and simulation to successfully integrate decision making and physical layers (Zeigler and Rozenblit, 1990). Since an autonomy system operates as an independent unit with layers structure, hierarchical structuring with modular models may be a good approach to simulation modelling of such a system. The main advantage of such a hierarchical, modular approach in simulation modelling is that hierarchical construction provides a convenient means for connecting components in a structured manner while modularity promotes modification and inter-substitution of components at the various levels of the hierarchy (Zeigler, 1990).

As high autonomy systems have multi-components, each having its own subcomponents, simulation models for such systems become complex, hierarchical structures. Often, such complex structures can be conveniently constructed by using isomorphic copies of existing models. Thus, special attention should be paid to check isomorphism between models before using them as component models. To create isomorphic copies
of various class of models in a unified manner, each class must have its own method to create such copies. Thus, object-oriented programming is well suited to developing methods for such a copying process.

Object-oriented programming paradigm is well noted to be compatible to modelling real world systems (O’Keefe, 1986; Zeigler, 1987a). Indeed, DEVS-Scheme realized Zeigler’s DEVS (Discrete Event System Specification) formalism in a LISP-based, object-oriented environment. The DEVS-Scheme environment supports specification of discrete-event models in modular, hierarchical fashion (Zeigler, 1987b; Kim and Zeigler, 1990b), a systems oriented approach not possible in conventional simulation languages.

This paper describes the concepts of hierarchical system for simulation of high autonomy systems and development of complex hierarchical models based on the concepts. Specifically, it develops methods for creating isomorphic copies of complex, hierarchical models in DEVS-Scheme as a means of constructing yet complex hierarchical simulation models. Section 2 gives an introductory example of a high autonomy system to motivate simulation based design of high autonomy systems. In section 3, we introduce the concepts of modular, hierarchical system for simulation modelling in such design. Sections 4 and 5 briefly describe the object-oriented programming paradigm, and the simulation concepts and class evolution in the DEVS-Scheme environment, respectively. Section 6 develops methods for isomorphic replications of complex, hierarchical models and the associated methods for checking isomorphism between models in the DEVS-Scheme environment. Conclusions are given in section 7.
2. INTRODUCTORY EXAMPLE: AIDECs (AI-Based, Distributed Environmental Control System)

This section briefly introduces an example of a high autonomy system for intelligent control, called the AIDECs (AI-Based, Distributed Environmental Control System). The AIDECs is under design for a Bioregenerative, Closed Ecological Life Support System (BCELSS), a complete closed life support system capable of sustaining itself for a long period. We shall use the AIDECs to illustrate hierarchical system concepts for simulation of high autonomy systems throughout this paper.

An intelligent environmental control system such as the AIDECs in the BCELSS needs to provide real-time control of the environmental variables such as temperature, humidity, and others. The functional requirements of such a control system include scheduling agricultural activities—such as seeding, irrigation, harvesting, pest control, and others—and collecting extensive data. Detail design issues including functional requirements can be found in (Kim and Zeigler, 1990a).

The AIDECs will be capable of supporting, in an integrated fashion, all activities ranging from planning of agricultural activities at the highest level to real-time control of environmental conditions at the lowest level. Here, we briefly describe the system architecture of the AIDECs under design. Details of the subsystems in the AIDECs are available in (Kim, 1990; Kim and Zeigler, 1990a).

The AIDECs, as shown in Fig. 1, consists of the Consultation Expert Systems (CES), the User Interface Systems (UIS), the Constraints Checker (CC), the Scheduling System (SS), the Control Data Base (CDB), the Schedule Decision Expert System (SDES), and the Real-Time CTRL & DAQ System (RTCDAS).
We now briefly describe subsystems in the AIDECS. The CES consists of a set of expert systems for identifying and controlling pest such as aphids in the BCELSS. The UIS enables the user to specify a time-based schedule to the environmental control system in a uniformed manner. The CC decides whether the schedule is acceptable or not by comparing it with a set of constraints contained within it, and sends it to the SS. The SS consists of the Schedule Manager (SM) and the Schedule Executor (SE). The SM generates the specification of the schedule, translates the specification
into a schedule object, and sends the object to the SE. The SE transforms the object into a set of activities, each of which has slots for a pair of condition and action, and others. The SE continually evaluates each activity in the set; if the condition of an activity is satisfied, the associated action in the activity is fired, which sends a micro-level control signal(s) to the RTCDAS. This signal has information on set point(s) such as temperature, and location within the BCELSS.

3. HIERARCHICAL, MODULAR SYSTEM CONCEPTS

Hierarchical constructions of modular models play an important role in modelling and simulation for design of complex, real world systems. This section presents the concepts of hierarchical, modular modelling of high autonomy systems with an example of the AIDECs.

3.1 Modularity and Hierarchy

Modularity and hierarchy are important properties in the software system design. In such a system, a module is a program text that can function as a self-contained and independent unit addressing only a single, logically coherent task. Such a module unit should be totally independent of the source of input, the destination of output, and the history of activation of the module. The important reason for keeping modules independent has to do with modification and testing of the software system. Interaction of such a module with others can occur only through predefined input/output ports. In the modular software system discussed here, we can treat the module unit much like a circuit board; we can remove one module and plug in a different one without affecting the remainder of the system, as long as the input and output specifications for the two modules are compatible.
Such a module is well characterized by a system that can be defined by an input/output interface, which characterizes the module’s interaction with other systems, and a state object, which represents the memory of the system. The system so defined naturally has an information hiding feature in that only those aspects of the system state that can be inferred from its input/output behavior are visible to, and can be manipulated by, the outside world.

Systems may be coupled to build composition systems, which may themselves be employed as components to be coupled with other systems to form higher level systems. A hierarchical construction is a finite recursion of such couplings. Thus a module is a programming construct that may be formally characterized as the system defined above. A software system specified by interfacing such modules is then usually characterized as a coupling of such systems. These concepts apply to the discrete-event world in which the DEVS formalism is a system-theoretic characterization of the programming constructs employed in discrete-event simulation languages.

3.2 Modular Models and Model Base

Model base, an organized library, has a set of models that are either atomic (models with no components) or coupled. The models in the set represent behavioral knowledge; how models behave when they receive stimuli. New models can be saved in, and saved models can be retrieved from, the model base. Models so retrieved may be used to create isomorphic models (we shall define models isomorphism later) that can be either atomic or coupled. Model behavior so retrieved will be attached to corresponding model structures to comprise a complete model.

Fig. 2 shows the fundamental concepts of modularity and model base. To explain the concepts, consider simulation modelling of the RTCDAS in the AIDCERS. Suppose
resulting model RTCS (Fig. 2 (b)) is called a coupled model, which is once again in modular form. As can be seen, the term modularity, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. Once placed into the model base in Fig. 2 (c), RTCS can itself be employed to construct yet larger models in the same manner used with its component models. This property, called closure under coupling, enables hierarchical construction of modular models (Zeigler, 1984).

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.p1, M2.p2) means that the output port p1 of model M1 (M1.p1) is connected to the input port p2 of model M2 (M2.p2). In this specification, “M1.p1 is connected to M2.p2” means that information flows only from M1.p1 to M2.p2. Thus, the coupling scheme of any model can be represented by the collection of three relations, namely, CS = (EIC, EOC, IC).

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 = \{(RTCS.current-in, SEN.in) (RTCS.desired-in, CTRL.desired-in)\} in Fig. 2 (b), means that input port current-in of RTCS is connected to input port in of SEN, and input port desired-in of RTCS is connected to input port desired-in of CTRL. 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.
resulting model RTCS (Fig. 2 (b)) is called a coupled model, which is once again in modular form. As can be seen, the term modularity, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. Once placed into the model base in Fig. 2 (c), RTCS can itself be employed to construct yet larger models in the same manner used with its component models. This property, called closure under coupling, enables hierarchical construction of modular models (Zeigler, 1984).

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 \((M1.p1, M2.p2)\) means that the output port \(p1\) of model \(M1\) (\(M1.p1\)) is connected to the input port \(p2\) of model \(M2\) (\(M2.p2\)). In this specification, \(M1.p1\) is connected to \(M2.p2\)” means that information flows only from \(M1.p1\) to \(M2.p2\). Thus, the coupling scheme of any model can be represented by the collection of three relations, namely, \(CS = (EIC, EOC, IC)\).

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, \(EIC = \{(RTCS.current-in, SEN.in) (RTCS.desired-in, CTRL.desired-in)\}\) in Fig. 2 (b), means that input port current-in of RTCS is connected to input port in of SEN, and input port desired-in of RTCS is connected to input port desired-in of CTRL. 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 \( \text{EOC} = \{(\text{SEN.out, RTCS.data-out}) , (\text{ACT.out, RTCS.control-out})\} \) in Fig. 2 (b) means that the output port \textit{out} of SEN is connected to the output port \textit{data-out} of RTCS, and the output port \textit{out} of ACT is connected to the output port \textit{control-out} of RTCS.

Internal coupling is the relation of the output ports of the components to the input ports of other components. It specifies how the components inside the coupled model are interconnected by indicating how the output ports of some components are connected to input ports of other components. The specification \( \text{IC} = \{(\text{SEN.out, CTRL.measured-in}) , (\text{CTRL.out, ACT.in})\} \) in Fig. 2 (b) means that the output port \textit{out} of SEN is connected to the input port \textit{measured-in} of CTRL, and the output port \textit{out} of CTRL is connected to the input port \textit{in} of ACT.

### 3.4 Hierarchical Construction of Modular Models

Since the RTCS model has been constructed, we continue constructing a yet larger hierarchical model for the AIDECs using the following definition.

A hierarchical model can be defined recursively in the following manner:

1) An \textit{atomic model} by itself is a hierarchical model.

2) Suppose \( C \) is a \textit{coupled model} and \( C_1, C_2, \ldots, C_n \) are hierarchical models. We can then construct a new hierarchical model by adding \( C_1, C_2, \ldots, C_n \) to the components of \( C \) and specifying associated coupling with them.

The general pattern by which hierarchical models can be constructed is shown in Fig. 3. As shown in the top of Fig. 3, a component can be either an atomic model or a coupled model (we use a double vertical arrows for representation). In the latter
case, it is built from one or more components (we use a triple vertical lines for representation) and an associated coupling specification. Since the term component appears twice, the diagram unfolds to an arbitrary depth. For example, if we stop the recursion after one round from the top of Fig. 3, we get the diagram shown in the bottom of Fig. 3.

We can construct an actual hierarchical model for the RTCDAS in the AIDECs from such a pattern, as shown in Fig. 4 (a). We start at the top, the root of the tree, and choose the coupled model, which we label RTCDAS. This choice necessitates specifying a coupling scheme and a set of components that can be atomic models or coupled models. We choose two atomic models and one coupled model. Selecting two atomic
models CENV and DAS, and one coupled model RTCS which we already constructed completes the hierarchical construction of the RTCDAS model. Note that associated coupling schemes attached to the coupled model represent connections among the
hierarchical models. Fig. 4 (b) shows the hierarchical model for the RTCDAS constructed in Fig. 4 (a).

Once the hierarchical model RTCDAS is constructed, a complete hierarchical model for the AIDECs can be constructed using the same manner as used in RTCDAS construction.

4. OBJECT-ORIENTED PROGRAMMING PARADIGM

The object-oriented programming (OOP) paradigm is a technique in which a software system is decomposed into subsystems based on objects. In such a paradigm, computation is done by objects sending messages among themselves. The object-oriented programming technique enhances software maintainability, extensibility, and reusability (Cox, 1986). This section briefly reviews the object-oriented programming paradigm.

4.1 Basic Concepts of OOP

Conventional software systems, implemented by languages such as C or PASCAL, tend to consist of collections of subprograms based on functional decomposition techniques that concentrate on algorithmic abstractions. Such systems has both a main program and subprograms that are called by the main program to work at appropriate times. Thus, the decision-making power of such systems is concentrated in the main program. The subprograms play a supporting role and come alive only when the flow of control passes through them.

In contrast to conventional programming techniques, object-oriented techniques implemented by object-oriented languages such as Smalltalk (Goldberg and Robson, 1989), C++ (Stroustrup, 1986), SCHEME-SCOOPS (TI, 1985) encourage a much more decentralized style of decision-making by creating objects that may exist
throughout the life of the program. Such objects may act as experts in their own task assignments by incorporating the appropriate knowledge within their own definitions. This distribution and localization of knowledge simplify the main routine and relieve it its decision-making power.

4.2 Objects and Message Passing

The basic units of the object-oriented program are objects. Each object has its own variables that represents its information, and procedures called method that describes the manipulation of its variables. Only the methods owned by the object can access and change the values of the object’s variables. Values originally assigned to the variables of an object will persist indefinitely throughout its lifetime unless they are changed by some method. Thus, the variables collectively constitute the state of the object. Only the methods of an object can change its state by subsequently changing the values of its variables.

A message is a specification that tells how the variables of an object are manipulated. Message passing is a process by which objects can communicate with each other, and with higher levels of control, to cause changes in their states. The object that receives a message is called receiver of the message. Each receiver has its own protocol, a set of messages that it can respond to. General form of a message is:

(send O1 m1 a1 a2 ..... an)

where, O1: receiver, name of object, m1: name of method to be applied, and a1, a2, ..... , an: list of argument values

This represent a message sent to object O1 telling it to apply its method m1 with the argument values a1, a2, ..... , an. Carrying out the orders of this message may change the state of the object and/or produce an output in response to the message.
One of the most useful concepts afforded by the object-oriented paradigm is that different objects can have variables or methods having the same names. However, different objects have a different set of values of the variables. A set of such variables exclusively owned by an object can be private. Such methods may exhibit a basic similarity in purpose despite a difference in the detailed manner in which this purpose is achieved.

4.3 Object Classes and Inheritance

In object-oriented programming systems, objects having similarities among them are not defined individually but by means of a class definition. Such a definition provides a template for creating any number of instances, or objects, each one an identical copy of a basic prototype. The class definition specifies information on name of class, class variables, instance variables, methods, constructor, destructor, and inheritance.

Objects are organized into a family of mutually distinct homogeneous classes to implicitly share certain, common properties using a technique called inheritance (Daniel and Patrick, 1987). The object-oriented paradigm provides a natural way to exploit such a technique to afford economy of class definition and extensibility. Actually, already in 1965 the discrete-event simulation language SIMULA (Franta, 1977) introduced concepts of class and class inheritance. In the simplest case of single inheritance, a class only inherits from at most one parent class (or superclass), and the classes form a tree structure, a specialization hierarchy under a root class.

The root class, a class having no parent class, is the most general class. A new class can be defined based on the older classes including the root class from which all features including class variables, instance variables and methods are inherited. Such a
new class is more specialized in that it may have additional variables and methods of its own, based on differences between the class and its parent. Its children, if created, may be even more specialized, inheriting from the parents and hence from the grandparents. General definitions for methods provided by the root class can be used to interface with higher-level procedures. The specialized definitions for methods provided by the more specialized classes can override such general definitions if both definitions have the same names for methods. The system may evolve by adding more and more specialized classes without changing the higher level procedures as long as the newly introduced specialized methods are as compatible as the general ones. Thus extensibility and program evolution are inherent in the object-oriented programming approach.

4.4 Object-Oriented Modelling and Simulation

In a real world system, it is natural to think that components are distributed throughout the system and interact with one another. Accordingly, a simulated model of such a system would have distributed components, each of which may have decision-making power of its own. It follows that the conventional programming technique is not well suited to modelling and simulation of such a system. Thus object-oriented programming provides a natural means of representing knowledge in a form suited for simulation modelling.

In object-oriented modelling simulation, a simulation model consists of objects, called components, which are connected to form a network. We consider that each real component has a set of states that changes over time when they interact with each other. Object models of these components can also have state variables represented by instant variables whose values are changed when messages are communicated and methods are applied. This sets up a correspondence between real components, their
states, and interaction on the one hand, and object counterparts, state variables, and message passing communication on the other. Such a correspondence is a natural one provided that the real system interactions can be conveniently represented by message communication.

If simulation implements a form of discrete-event simulation, state changes are brought about as events scheduled on a time base. Since classes for models and classes for simulators are defined separately, instances of a model class must be attached to instances of the corresponding simulator class to simulate the model. Indeed, this idea has been implemented in the DEVS-Scheme environment (Zeigler, 1987b; Kim and Zeigler, 1990b) which realizes Zeigler’s DEVS (Discrete Event System Specification) formalism which supports specification of discrete event models in hierarchical, modular manner.

5. DEVS-Scheme and CLASS EVOLUTION

The DEVS-Scheme environment enables the modeler to specify models in a manner closely paralleling the DEVS formalism (Zeigler, 1976; Zeigler, 1984; Concepcion and Zeigler, 1988). DEVS-Scheme supports building models in a hierarchical, modular manner, a systems oriented approach not possible in conventional simulation languages. Here, we describe simulation concepts and class evolution in DEVS-Scheme through polymorphism.

5.1 Simulation Concepts in DEVS-Scheme

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.

Since an abstract simulator does not know any information inside its associated model, the simulator consults with its model during the simulation time 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 model-processor pair. More specifically, the processor that receives a message consult with the attached DEVS model and get knowledge—such as receivers, influencees, interface map and others—that is required to route the received message to their appropriate components.

5.2 Classes in DEVS-Scheme

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 andy methods that correspond to components of structure in the formalism. The class coupled-models realizes the coupled DEVS which embodies the hierarchical model composition of the DEVS formalism. Subclasses of the coupled-models include digraph-models and kernel-models, which, in turn, has its own subclasses. Fig. 5 shows model classes in DEVS-Scheme. (See (Kim, 1988) for details).

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.
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 and details of how processors respond to different types of received messages can be found in (Kim, 1988).

5.3 Polymorphism and Class Evolution

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.

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 show polymorphism of *digraph-models* and *hypercube-models* in DEVS-Scheme to respond the same message received from their coordinators. Consider two models created from different subclasses of *coupled-models*: dig-M, an object of *digraph-models*, and hc-M, an object of *hypercube-models*. Assume that dig-M has three components with the coupling as shown in Fig. 6 (a), and that hc-M is a 3-dimensional hypercube as shown in Fig. 6 (b). 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

![Diagram](image.png)

**Fig. 6.** Models (left) and Associated Processors (right). (a) Digraph Model. (b) Hypercube Model.
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.

Consider that coordinators receive two types of messages, namely an external (x, t) message and an internal (done, t) message. 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. To do so, 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. To response the (done, t) message, C:dig-
M and C:hc-M have to know the influencees of DM2 and HM5, respectively. To know the influencees (or internal coupling scheme), C:dig-M and C:hc-M send the message get-influencees 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).

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-influencees that are specific to the new classes to reply questions asked by coordinators.

Polymorphism explained in this section along with inheritance described in section 4.3 plays an important role in developing methods for isomorphic replications of hierarchical models in DEVS-Scheme, as we shall see in the following section.

6. ISOMORPHIC REPLICATION OF HIERARCHICAL MODELS

Simulation models for high autonomy systems such as the AIDECs may have complex, hierarchical structures. Often, such complex structures can be conveniently constructed by using identical copies of existing models. This section describes how such complex hierarchical structures are isomorphically replicated in the DEVS-Scheme environment.

6.1 AIDECs Revisited

The AIDECs introduced in section 2 has a RTCDAS that interacts with a controlled environment. In a real world, the controlled environment may be divided into a collection of areas which are distributed in different locations. Moreover, more than one variable needs to be controlled in a designated area. With this in mind, we modify our AIDECs design by adding a multiple of RTCDAS’s to the previous one. The
RTCDAS's may be appropriately distributed in personal computers linked to the network if the AIDEC is implemented in a PC-based computer network.

Recall that the RTCS in the RTCDAS has three components: a sensor, an actuator, and a controller. However, we may decompose the component models into their subcomponents as requires, resulting in a more complex hierarchical model for the RTCS. Such decomposition may be continued until the modeler thinks comfortable to develop atomic models in a modular form. Likewise, other components of the RTCS, i.e., CENV and DAS, may need such decomposition. As a result, the RTCDAS has a yet complex hierarchical structure, consisting of RTCS, CENV, and DAS, each of which has its subcomponents, each of which, in turn, has its own sub-subcomponents and so on.

6.2 Manipulation of Hierarchical Structures

For constructing a number of hierarchical models for the RTCDAS described above, it is very convenient to replicate isomorphic copies of a hierarchical model for the RTCDAS. Thus, special attention has to be paid to isomorphically replicating the hierarchical structures. In fact, the DEVS-Scheme environment has facilities to replicate isomorphic copies of a hierarchical model. To be more specific, each model class in DEVS-Scheme has its own method to create isomorphic models and check isomorphism between a pair of models. We now describe how DEVS-Scheme exploits polymorphism and inheritance provided by SCOOPS, the object-oriented superset of Scheme (TI, 1985), to develop methods for creating isomorphic copies of complex, hierarchical models and also the associated methods for checking isomorphism between 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 members 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 how to create isomorphic copies of complex, hierarchical models in DEVS-Scheme, assume that the RTCDAS’s in the AIDECS are distributed in personal computers linked to a network with a ring topology. For simulation modelling of such RTCDAS’s, we need to create a number of hierarchical models of the RTCDAS connected to a ring. For this, we first need to create an instance of the class *ring-models*, a subclass of *kernel-models* in DEVS-Scheme, which will play as a coordinator for all RTCDAS models. We then create isomorphic copies of a RTCDAS model, all of which will be members (children) of the coordinator.
A facility `make-ring` and a method `make-members` in DEVS-Scheme are used to create such a ring model:

```
(make-ring RTCDASS)
(snd ri-RTCDASS make-members 'RTCDAS 5)
```

where, RTCDASS is an existing class, RTCDAS is the basic name of each member, and 5 is number of members in the ring.

As shown in Fig. 7, the first command makes a ring ri-RTCDASS with `kernel-class RTCDASS` and ri-init-cell, an instance of RTCDASS. Note that RTCDASS, which in this example is `digraph-models`, can be any subclass in DEVS-Scheme including any subclass of `kernel-models` class itself. The second command causes the sequence:

```
(snd ri-init-cell make-new 'RTCDAS1)
(snd ri-init-cell make-new 'RTCDAS2)

..................................................
(snd ri-init-cell make-new 'RTCDAS5).
```

![Diagram](image)  
**Fig. 7.** Facility make-ring.
The sequence creates above objects RTCDAS1, RTCDAS2,..., and RTCDAS5 each isomorphic to ri-init-cell (hence to each other) and belonging to the same class as ri-init-cell, namely the kernel-class RTCDASS. Recall that atomic-models, digraph-models, and the specialization of kernel-models have their own versions of make-new method. For example, since ri-init-cell is a digraph model, ri-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, each of which appropriately responds to the message with its own method.

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.

6.3 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*. Fig. 8 shows the *make-new* of the digraph model of RTCDAS, where RTCDAS1 is created from RTCDAS. Two instances, RTCDAS and RTCDAS1, are created from the same class, RTCDASS. Each component of RTCDAS applies method *make-new* to create an *isomorphic* copy of the corresponding component of RTCDAS1. The coupling scheme in the digraph model con-
sists 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.

### 6.4 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 (Zeigler, 1976). 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 the formal checking for 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 multicomponent 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 pre-order and creates corresponding components in such order. Using this pre-order to establish the correspondence makes isomorphism checking feasible.

6.5 Isomorphic 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. 9 (a)).
Method isomorphic? (m)
(and
  (= (state-var-set this-model) (state-var-set 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. 9(a). Method Isomorphic? for Atomic-models.

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. 9 (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-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
Method isomorphic? (m)
    ( 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.

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. 9 (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. 9 (e) and (f) show the method isomorphic-coupling? for
digraph-models and kernel-models, respectively.
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. 9(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 of this-model
get ext-coup2, a set of instance variables for
    external coupling scheme for m
if both external and internal coupling schemes are same
    return true
else
    return false
End

Fig. 9(f). Method Isomorphic-coupling? for Kernel-models.

6.6 Analysis of Isomorphism Algorithm

We analyze the running time of the algorithm for isomorphism between two models. We first briefly discuss the running time for determining whether two trees are isomorphic. Next, we analyze the algorithm for isomorphism between two models in the general case. This is followed by analysis in a special case where the components correspondence of two models is known.
It is well known that the running time to determine whether two n-vertex (labeled) trees are isomorphic is $O(n)$ time. As shown in (Aho, Hopcroft, and Ullman), the algorithm for trees isomorphism first assigns tuples of integers to vertices at each level for each tree in an inductive manner. This is done by assigning zero’s to all leaves. Then, it compares two lists of the integer tuples at each level from leaf to root. Such comparison at level i takes $O(m)$ time, where m is number of vertices at level (i-1), by using the lexicographic sort algorithm (Aho, Hopcroft, and Ullman, 1974).

The procedure for checking isomorphism between two hierarchical models may be very similar to that for two trees. However, there is a significant difference between vertices of a tree and components of a model in establishing the correspondence. The difference is that a vertex does not have information within it, while a component model contains information within it to represent a real world counter-part. The correspondence between components of two models cannot be established without knowing components’ internal structures. In addition, each class of models has its own method for comparing such internal structures. Fig. 9(a) shows comparison of the internal structures for a pair of atomic models.

As discussed above, we check isomorphism between a pair of models by checking isomorphism between a pair of components for each level from leaves. The work of determining isomorphism between a pair of m atomic models at leaves is proportional to $m!$. Thus, summing over all levels for a pair of n-components models results in $O(n!)$ time.

Usually, the correspondence of the components for two models is not known. However, our special case described in section 6.4 makes it possible to have such cor-
respondence. In the special case, determining the isomorphism between a pair of m-components at level i takes $O(m)$ time. Thus, summing over all levels results in $O(n)$ time.

7. CONCLUSIONS

Simulation modelling may be the only way to verify and/or evaluate design of complex high autonomy systems before actual systems are implemented. We have described the hierarchical, modular approach to modelling such systems which allows the designer to freely consider design alternatives at various levels of design abstractions. As shown in this paper, isomorphic replications of complex, hierarchical models are a powerful means to develop modular, hierarchical models as long as methods for checking such isomorphism are provided. We have developed such methods for checking isomorphism between models as well as for creating isomorphic copies of existing models in the DEVS-Scheme environment. It has been shown that the object-oriented nature of DEVS-Scheme makes it easier to develop such methods for different classes and their sub-classes by using inheritance and polymorphism.

REFERENCES


