SIMULATION

http://sim.sagepub.com

Variable Structure in DEVS Component-Based Modeling and Simulation

Xiaolin Hu, Xiaolin Hu, Bernard P. Zeigler and Saurabh Mittal

SIMULATION 2005; 81; 91
DOI: 10.1177/0037549705052227

The online version of this article can be found at:
http://sim.sagepub.com/cgi/content/abstract/81/2/91

Published by:

http://www.sagepublications.com

On behalf of:

Society for Modeling and Simulation International (SCS)

Additional services and information for *SIMULATION* can be found at:

Email Alerts: http://sim.sagepub.com/cgi/alerts

Subscriptions: http://sim.sagepub.com/subscriptions

Reprints: http://www.sagepub.com/journalsReprints.nav

Permissions: http://www.sagepub.com/journalsPermissions.nav

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

Variable Structure in DEVS Component-Based Modeling and Simulation

Xiaolin Hu
Computer Science Department
Georgia State University, Atlanta, GA 30303
xhu@cs.gsu.edu

Xiaolin Hu
Bernard P. Zeigler
Saurabh Mittal
Arizona Center for Integrative Modeling and Simulation
Electrical and Computer Engineering Department
University of Arizona, Tucson, AZ 85721

Variable structure refers to the ability of a system to dynamically change its structure according to different situations. It provides component-based modeling and simulation environments with powerful modeling capability and the flexibility to design and analyze complex systems. In this article, the authors discuss variable structure—specifically, the structure change and interface change capability—in DEVS-based modeling and simulation environments. The operations of structure change and interface change are discussed, and their respective operation boundaries are defined. Three examples are given to illustrate the role of variable structure and how it can be used to model and design adaptive complex systems. Principles for the implementation of variable structure are also presented and illustrated in the DEVSJAVA modeling and simulation environment.

Keywords: Variable structure, component-based modeling and simulation, DEVS, adaptive complex systems

1. Introduction

With the rapid advance of component-based technology in software engineering, component-based software has been widely used to develop highly modular simulation environments. The integration of component-based technology with modeling and simulation environments gives the latter powerful capability and greatly supports reusability of components and interoperability of simulation environments. The reuse of components, together with visual programming technology, makes it possible to drag and drop existing components during the modeling process, thus easing system modeling and significantly reducing development time. With component-based technologies such as the federate concept introduced by the High Level Architecture (HLA) [1-3], different simulation environments can interact through standard interfaces and work together. Thus, interoperability is achieved, and more powerful simulations can be conducted. Component-based technology also makes the modeling of a complex system easier to manage because a complex system can be divided into several manageable pieces, each referring to a component. It also promotes distributed simulation as component-based technology has a natural fit to distributed environments.

Motivated by these advantages, various component-based modeling and simulation environments have been developed. Furthermore, HLA was developed to enhance the interoperability of models and simulation environments. In HLA, component models are referred to as federates. Each federate provides an interface through which messages can be passed and received. Because these interfaces comply with the same HLA interface specification, federates developed by different developers can communicate with each other via the runtime infrastructure. Other works such as JSIM [4], SIMKIT [5], Silk [6], and VSE [7] focus on the implementation of component-based modeling and simulation environments. For example, the JSIM simulation environment uses Java and Java beans technology to support component-based modeling and simulation. A visual design interface is provided for users to develop and assemble components.

The Discrete Event System Specification (DEVS) [8] supports component-based modeling and simulation by emphasizing the theory of hierarchical modular modeling. In a DEVS-based environment such as DEVSJAVA [9], a component is a model with clear-defined interfaces called input and output ports. A model could be an atomic model or coupled model, which is composed from other DEVS models. By adding couplings between output/input ports

SIMULATION, Vol. 81, Issue 2, February 2005 91-102
©2005 The Society for Modeling and Simulation International
DOI: 10.1177/0037549705052227

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

of different components, messages can be passed from one component to another. Under the property of closure under coupling, a coupled model itself can be treated as a subcomponent of other models. This kind of hierarchical modular construction makes each DEVS model a self-contained component that can be easily reused. Because of this, the DEVS component-based modeling and simulation environment does not rely on the underlining implementation language. In fact, various DEVS environments such as DEVSC++, DEVSJAVA, DEVSCorba, and so forth [9-11] have been developed. The DEVS/HLA [12] was also developed to allow DEVS to work with HLA.

A component system is built by composition of individual components. Thus, in a component-based modeling and simulation environment, the modeling process is to build components and to assemble them to capture a system's structure and behavior. For some complex systems, the structures and behaviors could be very complex as the systems may continuously reconfigure themselves to adapt to different situations. For example, a distributed computing system may dynamically add or remove computing nodes according to the load of the system. Other examples include the ecological systems that typically evolve over time to adjust to the new environment. To model these complex systems, a variable structure modeling capability is needed. As variable structure greatly enhances the modeling capability, it also raises special design issues for component-based modeling and simulation environments.

In this article, we discuss the variable structure modeling capability in DEVS component-based modeling and simulation. While previous work [13-17] has established a theoretical background for the variable structure of DEVS, this article discusses it in the context of component-based technology and covers more aspects of it. The article first elaborates on the conceptual development of variable structure in component-based modeling and simulation. Then it discusses three examples to illustrate the role of it. After that, the implementation of variable structure in a DEVS modeling and simulation environment is presented. Finally, conclusions are drawn, and some open issues are discussed.

2. Conceptual Development for Variable Structure in DEVS

A component is "a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. It conforms to and provides the physical realization of a set of interfaces" [18]. A component system is built by composition of individual components and by establishing relationships among them. As each component holds a high degree of autonomy and has well-defined interfaces, variable structure of components can be achieved during runtime. For component-based modeling and simulation, variable structure provides several advantages: (1) it provides a natural and effective way to model those complex systems that exhibit structure and behavior changes to adapt to different situations. Examples of these systems include distributed computing systems, reconfiguration computer architectures [19, 20], fault-tolerance computers [21], and ecological systems [15]. Structure changing and component upgrading is an essential part of these systems. Without the variable structure capability, it is very hard, if not impossible, to model and simulate them. (2) From the design point of view, variable structure provides the additional flexibility to design and analyze a system under development. For example, as will be illustrated later, variable structure gives us the flexibility to design and simulate a distributed robotic system in which robots form relationships dynamically. (3) Variable structure makes it possible to load only a subset of a system's components for simulation. This is very useful to simulate very large systems with a tremendous number of components, as only the active components need to be loaded dynamically to conduct the simulation. Otherwise, the entire system has to be loaded before the simulation begins.

In general, there are six forms of reconfiguration of component-based systems [22]: addition of a component, removal of a component, addition of a connection between components, removal of a connection between components, update of a component, and migration of a component. The first four operations result in a structure change of the component-based system. In DEVS, they are usually referred to as variable structure modeling. The update of a component means a component is updated by a new version that might have a totally different behavior or interface from the old one. This can be accomplished either by replacing the old version with a new one or by directly upgrading a component to a new version. Replacing a component involves the process of adding the new component and removing the old one, as can be realized by the addition and removal operations. In this article, we are also interested in the upgrade of a component. Specifically, we discuss how a DEVS model (component) may change its interface by adding or removing its input and output ports in different stages. The migration of a component actually implies two involved entities: a component and the location (physical or soft) of the component. Since this is usually researched in mobile agent systems, it is not discussed in this article.

Figure 1 gives an example that shows a simple process of structure change. In this example, the initial system has two components, A and B. Then, component C and the connection from C to B are added. After that, component A is removed, resulting in a final system with two components, C and B. Note that removal of a component will automatically remove all the connections related to that component. In a modular DEVS environment, DEVS models are the components, and DEVS couplings are the connections. Thus, variable structure in DEVS means that DEVS models and couplings can be added or removed dynamically. Corresponding to the four operations of structure change, four methods are provided in a DEVS environment. They are *addModel()/removeModel()* to add/remove

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

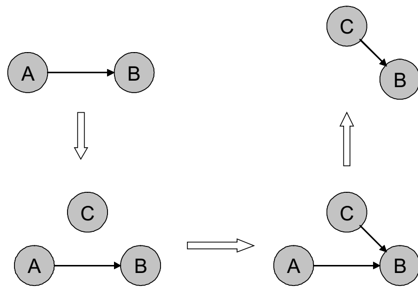


Figure 1. A variable structure process

DEVS models and *addCoupling()/removeCoupling()* to add/remove DEVS couplings. Note that the *addCoupling()* and *removeCoupling()* methods take four parameters: the source model, source model's output port, the destination model, and the destination model's input port. With these methods, the structure change process shown in Figure 1 can be realized as follows:

1. *addModel(C);*
2. *addCoupling(C, COutputPort, B, BInputPort);*
3. *removeModel(A);*

Natural questions for variable structure systems arise concerning the authorization and timing of the structure changes. Generally speaking, there is no specific restriction on which component cannot initiate a structure change. However, because a DEVS coupled model does not have its own behavior, an atomic model is needed to initiate a structure change. The initiation typically happens in the atomic model's internal or external transition functions. This is reasonable because a structure change is usually triggered by situation changes, which are captured as events in DEVS and are handled by the external or internal transition functions. In this sense, the atomic model acts as a supervisor to monitor the conditions of interest. For the system shown in Figure 1, component B could be the one to monitor the system's situations and initiate the structure change. For example, it may monitor the input from A. If this input is less than a predefined value, it adds component C and the coupling from C to B. Then it monitors the input from C, and if this input is greater than a predefined value, it removes A.

2.1 Operation Boundary

Another important question for variable structure systems is how to determine the particular components that can be affected by a structure change operation. To answer this question, we introduce the *operation boundary* concept and define it as the safe scope to conduct a meaningful operation. For example, in a distributed environment, a component can remove components on its local computer, but it is not allowed to remove components on remote computers. The latter violates the operation boundary of the remove operation in a distributed environment. To support the operations boundary in DEVS, models can maintain information on their locations in relation to the hierarchical structure of the overall coupled model. Components of the same coupled model, therefore belonging to the same parent, are called brothers. This approach is based on the structure knowledge maintenance concepts in Zeigler [23].

Thus, the structure change operations also need to work within this hierarchical structure and to maintain this structure. On the basis of this, we define the operation boundaries of the four structure change operations as follows:

- *addModel*(...): a model can only add components to its parent coupled model.
- *removeModel*(...): a model can only remove itself and its brothers.
- *addCoupling*(...): a model can only add couplings involving itself, its parent, and its brothers.
- *removeCoupling*(...): a model can only remove couplings involving itself, its parent, and its brothers.

These clearly defined operation boundaries make it easier for a user to check if an operation is legal or illegal. For example, it can easily be seen that a model can remove itself, but it cannot remove its parent. Our approach differs from that formalized by Barros [24], who uses a central network executive to initiate structure changes. We find that much greater flexibility, at minimal cost, is achieved by allowing any component in a coupled model (or network) to initiate changes within the operations boundary.

We note that operations boundaries are defined in terms of the model hierarchical structure independently of any distribution considerations. In distributed simulation, components reside on different computers, and it is up to the distributed environment to ensure that the correct structure changes are carried out as prescribed by the structure modification commands. The distributed coupling change capability is supported by the DEVSJAVA environment. That is, couplings can be added or removed between models on different computers. It is up to the DEVS simulators to determine whether the coupling change is local or involves other computers. However, remotely adding/removing models in DEVSJAVA is currently not supported.

2.2 Changing Port Interfaces

Besides structure change, another reconfiguration feature is provided in DEVS to allow an atomic model to add/remove input or output ports dynamically. For this purpose, the *addInport()* and *addOutport()* are provided for an atomic model to add new input and output ports, respectively; the *removeInport()* and *removeOutport()* are

1. Although it can be accomplished by sending a message to a remote simulator, which then conducts the adding/removing operation locally, we have not completed the design details.

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

provided for an atomic model to remove existing input and output ports, respectively. As input and output ports are the interfaces of DEVS models, changing ports of a model usually requires that the model's behavior also change accordingly. Thus, special attention has to be paid when adding/removing ports dynamically. The modeler has to ensure that if a model receives a new input (or output) port, the model has, or obtains, a corresponding way to handle the possible input received (or generated) on this port. We define the operation boundary of adding/removing ports as a model can only add/remove ports of itself and its brothers. Thus, atomic models inside a coupled model have the capability to modify the interfaces of their brothers, although the functionality to handle messages at those interfaces should be there or should be provided in the modified models. Particular ways of accommodating new ports are known. For example, one can make ports adhere to a labeling scheme such as *name* + *index*, which can be analyzed and interpreted. Detailed explanation will be given in section 3.3. As a new feature of DEVS variable structure, more research is needed to answer questions such as how to provide a general mechanism to update a model's external transition and output functions accordingly after the model's input and output ports are added/removed dynamically.

3. Examples of Variable Structure

To illustrate the role of variable structure in component-based modeling and simulation, we describe three examples in this section. The first example shows how a complex distributed robotic system can be designed and simulated using the variable structure capability. The second example illustrates the ability to employ variable structure to dynamically emulate the system entity structure (SES). The last one describes an advanced workflow model that dynamically reconfigures itself by adding/removing models and changing the interface of models.

3.1 Dynamic Team Formation of a Distributed Robotic System

Distributed robotic systems have been a very active research topic recently. In Zhang et al. [25] and Butler, Fitch, and Rus [26], a group of robots that can change their shape has been reported. As those robots change hardware components of their own, in this section, we describe a distributed multirobotic system that changes its software components. This system exhibits dynamic team formation in which independent robots form teams dynamically and then conduct a *Leader-Follower* march. We first describe a system that has been realized by two real mobile robots [27]. Then we discuss a more scalable system with an indefinite number of robots. The robot we use in the example is a car-type mobile robot with wireless communication capability [28].

For the system with two robots, the team formation process starts with both robots moving around and trying to find each other. Initially, there is no connection between these two robots, although they are connected to a software process, called a *Manager*, on a wireless laptop. When two robots find each other, the *Manager* establishes direct connections between them and asks them to organize into a *Leader-Follower* team. Then they begin to march: one follows the other with the same movement. During the march, if two robots lose each other, they will inform the *Manager* and then go back to the initial state to search for each other.

From the above description, we can recognize three basic components in this system: the *Manager*, which resides on a laptop (computer), and *robot1* and *robot2*, which reside on mobile robots. Figure 2a shows the model of this system, where the *Manager* is an atomic model, and each *robot* is a coupled model (Fig. 2b). A complete description of the *robot* model can be found in Hu and Zeigler [27]. The coupling of the system is as follows ($R1$ stands for *robot1*, $R2$ stands for *robot2*, *man* stands for *Manager*, and *distanceData*, *report*, *check*, etc. refer to the port names):

addCoupling(R1, "distanceData," man, "Robot1_distanceData");
addCoupling(R1, "report," man, "Robot1_report");
addCoupling(man, "Robot1_check," R1, "check");
addCoupling(R2, "distanceData," man, "Robot2_distanceData");
addCoupling(R2, "report," man, "Robot2_report");
addCoupling(man, "Robot2_check," R2, "check");

As we can see, there is no coupling between *robot1* and *robot2*. Each robot has output ports *distanceData* and *report*. These ports are coupled to the *Manager*'s corresponding input ports. Meanwhile, the *Manager* has output ports coupled to each robot's input port *check*, so that *Manager* can ask them to check if they are within the line of sight. The robots return the check result using the *report* port. Once the report messages returned from the robots are both positive, it means two robots are close and they see each other. In this case, the *Manager* will change the couplings of the system dynamically to establish a direct connection between the two robots. Specifically, in this example, the manager executes the following DEVSJAVA code:

removeCoupling(R1, "distanceData," man, "Robot1_distanceData");
removeCoupling (man, "Robot1_check," R1, "check");
removeCoupling (R2, "distanceData," man, "Robot2_distanceData");
removeCoupling (man, "Robot2_check," R2, "check");
addCoupling(R1, "readyOut," R2, "readyIn");
addCoupling(R2, "readyOut," R1, "readyIn");

Note that the *addCoupling* method is overloaded so it accepts strings to specify components in addition to object references. This feature makes it convenient for the modeler to keep track of models that have been added using

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

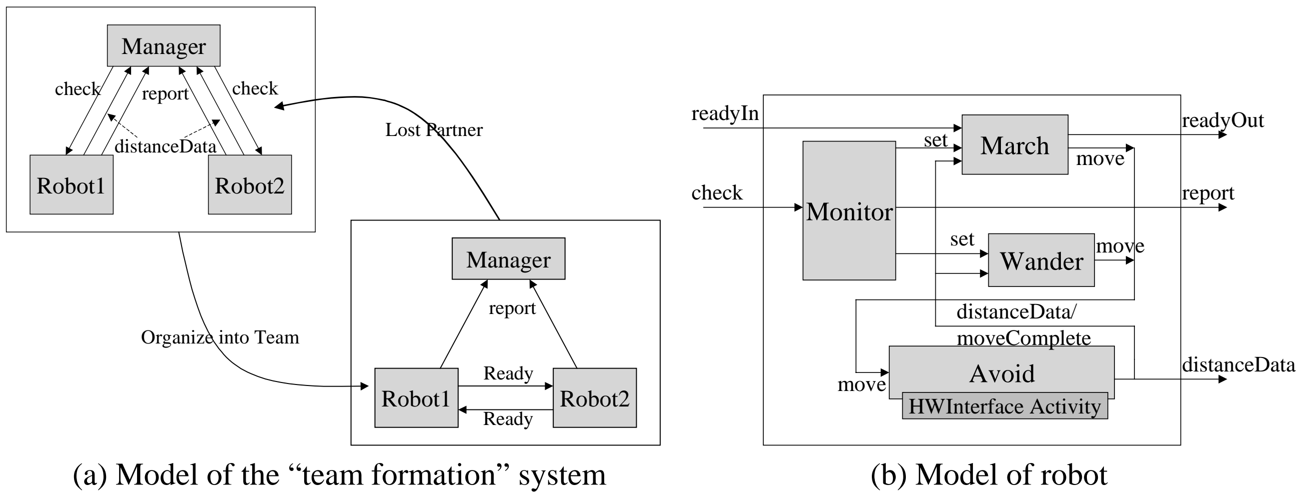


Figure 2. Model of dynamic team formation system

string names. Explicit references can also be obtained from the parent coupled model by supplying the string names. This requires that all models be given unique names. After executing the DEVSJAVA code, a bidirectional connection is established by coupling two robots' *Ready* port to each other, so they can communicate directly. The *distanceData* and *check* couplings between robots and *Manager* are removed because they are no longer needed during the process of the robot march. The *report* coupling remains so robots can still inform the *Manager* in case they lose each other. During the march, if two robots lose each other, they send the "*Lost Partner*" message to the *Manager* using the *report* port. This will trigger the *Manager* to add and remove couplings among the components. As a result, the system goes back to the initial situation, where two robots move independently and try to find each other.

As the above system only includes two robots, more scalable systems with an indefinite number of robots can be developed based on the same variable structure idea. Figure 3 shows an example with 10 independent robots searching for each other, forming groups dynamically, and finally organizing into one large *Leader-Follower* team. During this process, couplings between models are added and removed, resulting in a variable structure system.

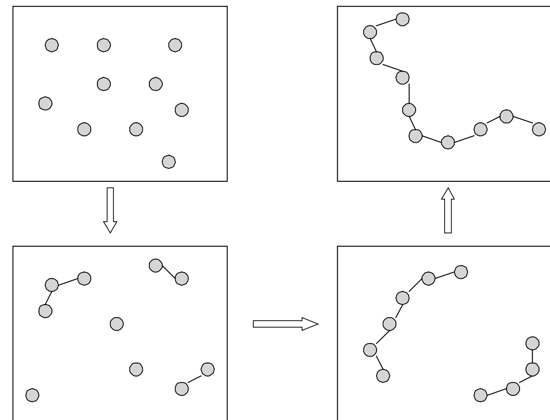


Figure 3. A scalable dynamic team formation example

3.2 Dynamically Emulate the System Entity Structure (SES)

The system entity structure (SES) provides a way for specifying system composition [29] with information about decomposition, coupling, and taxonomy. It also provides a formal framework for representing the family of possible structures. From the design point of view, SES represents the design space with various possible design configurations. Thus, the process of design/analysis is to prune SES—in other words, to search the best design configuration. For complex systems, the number of the combination of different configurations is very large. Thus, it is desirable to be able to emulate SES and automatically search the best design configuration. In this section, we show an example that demonstrates how this can be achieved by employing the variable structure capabilities.

This example system is *efpSES*, as shown in Figure 4a. It has two components: an experimental frame model *ef* and a processor model that has three specializations representing three design choices of the system. The specializations of the processor model include a single processor, *proc*; a divide-and-conquer processor, *DandC3*; and a pipeline processor, *pipeLine*. To automatically simulate all these alternatives of the processor model, *efpSES* employs an

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

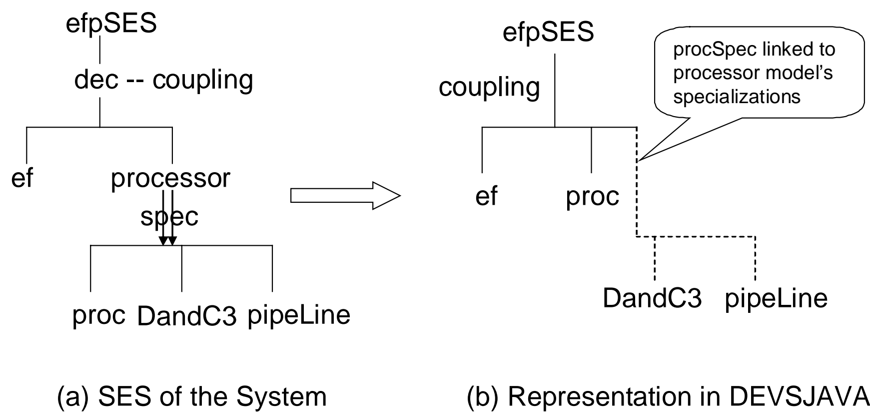


Figure 4. Dynamically emulate the system entity structure (SES)

instance of class *specEntity* to control the successive substitution of alternatives. *specEntity* is a specialized entity developed to emulate the SES of a system. In this example, the user defines *procSpec*, a subclass of *specEntity*, and provides it with the first and subsequent specializations: *proc*, *DandC3*, and *pipeLine*. Then, as shown in Figure 4b, the user adds *procSpec* to the coupled model and tells it which component to control (the dashed lines in Figure 4b show that *procSpec* is linked to the processor model's specializations). Based on this information, during simulation, the *procSpec* automatically replaces the processor model with different specializations until all of them are tested. Since the addition of local control components preserves the hierarchical, modular structure, the hierarchical properties of the SES are automatically obtained. Moreover, this variable structure capability provides a general way to emulate the SES and automatically test all the alternatives of a system's design space, as described in Couretas, Zeigler, and Patel [30].

While the SES involves only replacement of components by alternatives, the approach can be further extended to allow a restructuring executive to observe the simulation and make decisions regarding the alternatives to employ based on prevailing conditions. Such restructuring is discussed in the following example.

3.3 A Reconfigurable Workflow System

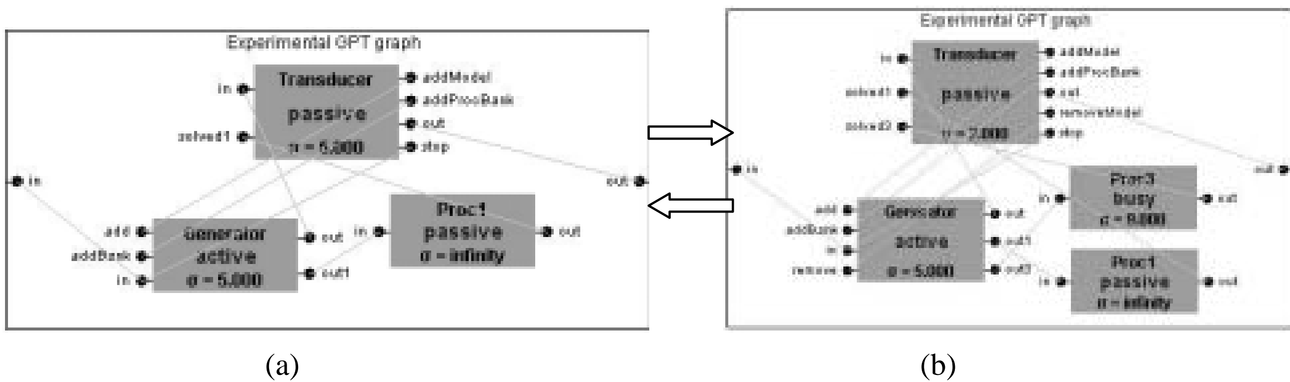
A simple workflow prototype is referred to as GPT. This is a coupled model that is composed of a *Generator*, a *Processor*, and a *Transducer*. It is the simplest self-contained model that simulates three basic components of any workflow system. The *Generator* generates jobs, the *Processor* processes them, and the *Transducer* keeps track of the system state as a whole computing performance indexes such as system throughput (jobs processed per second) and average job turnaround time. In this section, we describe a reconfigurable GPT system where the *Processor*(s) can be dynamically added or removed and the *Generator* and *Transducer* can change their interfaces accordingly.

As shown in Figure 5a, this system starts with the basic GPT components: *Generator*, *Proc1*, and *Transducer*. *Generator* generates jobs and sends them out through the *out1* port coupled to the *Proc1's in* port. *Proc1* executes the job and sends the solved job to the *Transducer* at the *solved1* port. Note that the *Generator* has input ports *add* and *addBank*, and the Transducer has output ports *addModel* and *addProcBank* coupled to the two *Generator* ports, respectively. This suggests that the system has the capability to add a processor and a processor bank.

In this example, the *Transducer* makes decisions of when to add or remove processor(s). The *Generator* executes the addition or removal operations. Thus, if the *Transducer* notices that *Proc1* cannot handle all the generated jobs, it sends out a message to the *Generator*, which then adds another processor, *Proc3*. As shown in Figure 5b, *Proc3* is in a similar position as *Proc1* in the system. Note that the interfaces of *Generator* and *Transducer* also change accordingly. Besides the *Generator*'s earlier output port *out1*, a new output port, *out3*, has been added explicitly for *Proc3*. Similarly, the *Transducer* has added input port *Solved3* to collect jobs processed by *Proc3*. Also, the *Generator* and *Transducer* are now outfitted with ports for removing the processor (*remove* and *removeModel* ports). This is a new functionality that has been added in this stage. The interface change of *Generator* and *Transducer* is a reflection of the system's structure change. Initially, there was no functionality to remove models, as there was no need of it. As new processors are added, so is the corresponding functionality to remove them. A typical set of commands that were executed by the *Generator* after receiving the addition message from the *Transducer* is as follows:

2. See gpt.java in the SimpArc package of DEVSJAVA.

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.



(a) (b)

Figure 5. Stages of the reconfigurable GPT system

mg = new modelProc("Proc" + index); // in this example, the value of index is 3
addModel(mg);
addOutport("Transducer," "removeModel");
addInport("Generator," "remove");
addOutport("Generator," "out" + index);
addInport("Transducer," "solved" + index);
addCoupling("Transducer," "removeModel," "Generator," "remove");
addCoupling(getName(), "out" + index, ("Proc" + index," "in");
addCoupling(("Proc" + index, "out," "Transducer," "solved" + index);

Notice that a labeling scheme is used as the *Generator* model adds output port *out + index* for the new processor. Similarly, the *Transducer* handles the jobs solved by the processor using input ports with name *solved + index*. This allows expressing the *Transducer*'s processing by parsing port names to obtain their role and index parts, independently of the number of processors. The *Transducer* retains its basic behavior independent of the structure change by providing the code in advance to handle the messages coming on new ports. More flexible approaches may be obtained by providing schemas that can be accessed at runtime to support desired interfaces, a subject for further research.

In this example, after *Proc3* is added, it can also be removed when the *Transducer* thinks *Proc1* alone is enough to process all the generated jobs. To achieve this, the *Transducer* sends out a *removal* message using the *removeModel* port to the *Generator*. The *Generator* then removes *Proc3*, and the system goes back to the initial stage. Similarly, a processor bank (a coupled model) that contains multiple processors can also be added and removed.

From the above description, we can see that the system is able to expand itself, modify the interfaces of its components according to the structure change, and then shrink back to the original system. It displays a complete cycle of growth, from a basic functional level to an expanded system capable of high throughput and coming back to the initial state when its job (maximizing throughput) is done.

4. Implementation of Variable Structure in DEVS

The implementation of variable structure is based on the earlier development of the DEVSJAVA modeling and simulation environment. So our discussion starts from a review of this environment, with emphasis on the hierarchical structure of DEVS models and simulators. Although a particular implementation environment is employed as a basis, the design is generic and can be employed in any hierarchical, modular DEVS environment.

4.1 Hierarchical Structure of DEVS Models and Their Simulators

In a DEVS modeling and simulation environment, there is a clear separation between models and their simulators. DEVS models are defined by the users to model the system under development. DEVS simulators are provided by the DEVS simulation environment to simulate or execute DEVS models. Corresponding to the hierarchical structure of a DEVS model, its simulators also form a hierarchical structure. Figure 6 gives an example that shows the relationship of a hierarchical coupled model and its corresponding simulators (the dashed lines show the hierarchical relationship between simulators). This model has three components: *Atomic3*, *Atomic4*, and *Coupled1*, which has two subcomponents: *Atomic1* and *Atomic2*. The simulators manage the information of the hierarchical coupled model in a hierarchical way. On the very top level, there is a *coordinator* assigned to the coupled model. This *coordinator* is the parent of all its subsimulators, which have a one-to-one relationship to the components of the coupled model. Following the hierarchical structure of the coupled model, there is a *coupledSimulator* assigned to each atomic model and a *coupledCoordinator* assigned to each coupled

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

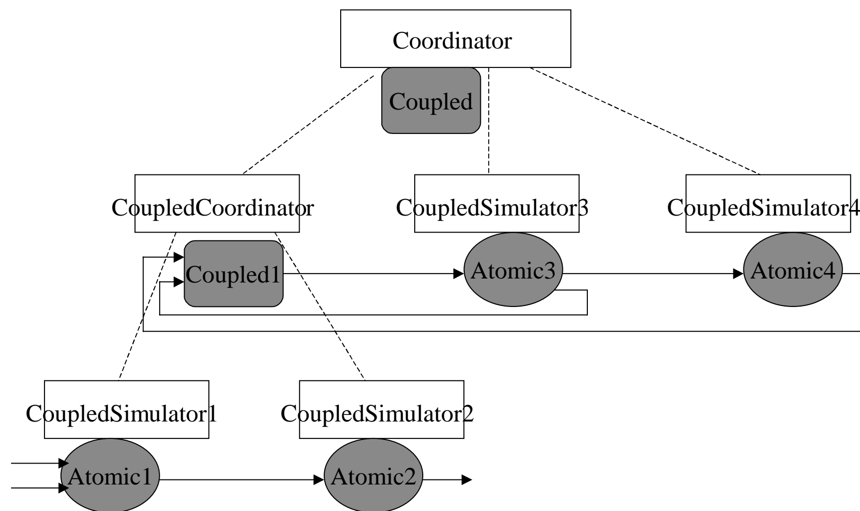


Figure 6. Relationship between models and their simulators (fast-mode simulation case)

model. A *coupledCoordinator* acts as both a *coordinator* and a *coupledSimulator*. This is because it needs to communicate not only with its children (like a *coordinator*) but also with its parent and brothers (like a *coupledSimulator*).

This hierarchical structure of models and simulators requires several data structures to keep information so that the system can be efficiently implemented. Figure 7 shows the related data structures managed by simulators and models. This figure also shows that the *atomic* class implements *variableStructureInterface*, which defines the methods for adding/removing DEVS models, couplings, and ports. For simplicity, Figure 7 only shows the information related to the implementation of variable structure.

First, let us see the data structures managed by DEVS coupled models, as shown by the *digraph* class in Figure 7 (atomic models do not need them). This is straightforward because coupled models need to keep track of their subcomponents and the couplings among them. Thus, each coupled model has two variables as defined as follows:

- *ComponentsInterface components;*
- *couprel cp;*

The data structure for simulators can be categorized into three categories to store three different types of information as shown as follows:

- Children simulator info: *ensembleSet simulators;*
- Model's coupling info: *couprel coupInfo, extCoupInfo;*
- Model-simulator mapping info: *Function modelToSim, internalModelTosim;*

The first variable, *simulators*, is used by a *coupledCoordinator* (*coupledSimulator* does not use it) to store its children simulators. For example, in Figure 6, the *simulators* variable for *coordinator* has three instances: *coupledCoordinator1*, *coupledSimulator3*, and *coupledSimulator4*. The *simulators* variable for *coupledCoordinator1* has two instances: *coupledSimulator1* and *coupledSimulator2*. The second group of variables, *coupInfo* and *extCoupInfo*, is used by simulators to store the models' coupling information. Specifically, *coupInfo* stores the couplings that start from a model and end with the model's brothers or parent. *extCoupInfo* is used by *coupledCoordinator* (*coupedSimulator* does not use it) to store the couplings that start from a model and end with the model's children models. Using *coupledCoordinator1* in Figure 6 as an example, the *coupInfo* has one coupling instance that starts from *Coupled1* and ends with *Atomic3*. The *extCoupInfo* has two coupling instances; both of them start from *Coupled1* and end with *Atomic1*. The third group of variables, *modelToSim* and *internalModelTosim*, is used by simulators to store the model-simulator mapping information. Again, using *coupledCoordinator1* in Figure 6 as an example, the *modelToSim* has three instances: *(Coupled1,coupledCoordinator1)*, *(Atomic3,coupledSimulator3)*, and *(Atomic4,coupledSimulator4)*. The *internalModelToSim* has two instances: *(Atomic1,coupledSimulator1)* and *(Atomic2, coupledSimulator2)*.

Note that in this implementation, each model and simulator manages its own copy of information. This approach relieves the central coordinator's involvement in its child simulators' local activities. For example, by keeping a local copy of the coupling information, a simulator can send its model's output messages directly to the destination simulators. More information about the advantages of this approach can be found in Cho, Hu, and Zeigler [31] and Cho [32].

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

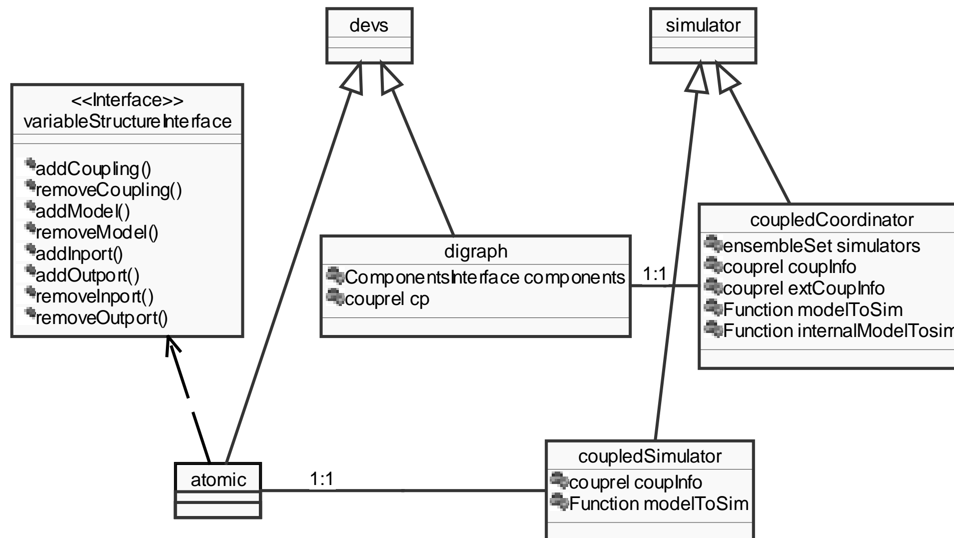


Figure 7. Methods and data structures used in variable structure implementation

4.2 Add/Remove Coupling Dynamically

Because DEVS models and simulators use coupling data structures to keep all the coupling information, the basic idea to implement this feature is to update those data structures. Below, we use *addCoupling()* to show how it works.

```
public void addCoupling(String src, String p1, String
dest, String p2){
    digraph P = (digraph)getParent();
    P.addPair(new Pair(src,p1),new Pair(dest,p2)); //
    update its parent model's coupling info
    coordinator PCoord = P.getCoordinator();
    PCoord.addCoupling(src,p1,dest,p2); // update the
    corresponding simulator's coupling info
}
```

The method first gets its parent, which is a coupled model. Then it calls its parent's *addPair()* method to update the parent's coupling information, the *cp* variable, as described in section 4.1. To update the coupling information of the affected simulators, the atomic model then calls the *coordinator's addCoupling()* method. This method uses the source model's name to find the corresponding simulator and then updates that simulator's coupling information, which is kept in the *coupInfo* or *extCoupInfo* variables. Note that for implementation convenience, the *getParent()* method is used. This method returns the parent model's reference that was established during the simulation's construction stage. As this method is not accessible to the modelers, it does not violate the hierarchical modular property of DEVS models.

4.3 Add/Remove Model Dynamically

Adding a model dynamically means not only that a new model is added but also that a new simulator needs to be created and added into the system. Furthermore, the new simulator needs to be initialized and synchronized with the ongoing simulation system. The *addModel()* method is shown as follows:

```
public void addModel(IODevs iod){
    digraph P = (digraph)getParent();
    P.add(iod);
    coordinator PCoord = P.getCoordinator();
    PCoord.setNewSimulator((IOBasicDevs)iod);
}
```

This method first adds the model as a new component to its parent by calling the *add()* method (update parent's *components* variable). Then it calls the *coordinator's setNewSimulator()* method. This method creates a new simulator for the added model and initializes that simulator. It is shown as follows:

```
public void setNewSimulator(IOBasicDevs iod){
    if(iod instanceof atomic){ //do a check on what model
    it is
        coupledSimulator s = new coupledSimu-
        lator(iod);
        ........ //update the corresponding data
        structures;
        s.initialize(getCurrentTime());
    }
    else if(iod instanceof digraph){
        coupledCoordinator s = new coupledCoordin-
        ator((Coupled) iod);
        ........ // same as when the model is atomic
    }
}
```

As can be seen, the method creates a new simulator based on the model type (atomic model or coupled model).

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

Then it updates the corresponding data structures such as *simulators*, *internalModelToSim*, and *modelToSim*. Finally, it initializes the created simulator. To synchronize with the current simulation time, the *initialize()* method takes the parameter of *getCurrentTime()*, which returns the current simulation time. After all these data structures are updated, the added model becomes eligible to participate in the subsequent simulation cycle, contributing to the determination of the global time of the next event and being able to receive inputs and generate outputs in the normal manner. Further details on the modification of the DEVS protocol needed for a well-defined variable structure are given in Zeigler [33].

Reverse to what adding a model means, removing a model dynamically means removing a model and its corresponding simulator(s) from the system. It also implies removing all the couplings related to that model from the system. Below is the *removeModel()* method. The method is basically the reverse of what *addModel()* does. It first removes the model from the parent model, and then it calls the *coordinator/coupledCoordinator's removeModel()* method to remove the simulator of that model. One extra step here is the *removeModelCoupling()* method, which removes all the couplings related to the model.

```
public void removeModel(String modelName){
    digraph P = (digraph)getParent();
    coordinator PCoord = P.getCoordinator();
    PCoord.removeModelCoupling(modelName); //
    remove the couplings of that model
    IODevs iod = P.withName(modelName);
    P.remove(iod); // remove the model
    PCoord.removeModel(iod); // remove the simulator
}
```

4.4 Add/Remove Coupling in Distributed Environment

Before we proceed to discuss how to implement the distributed coupling change capability, let's see how distributed simulation is implemented in DEVSJAVA. Figure 8 shows a distributed example with the same model as in Figure 6. In this example, the three components of the coupled model—*Coupled1*, *Atomic3*, and *Atomic4*—are distributed on three different computers. As can be seen, for each distributed component on a computer, there is a client simulator assigned to it (*CoupledSimulatorClient* for an atomic model; *CoordinatorClient* for a coupled model). These clients connect to a *CoordinatorServer*, which may reside on another computer (the dashed circles mean different parts of the system reside on different computers). During initialization, the *CoordinatorServer* waits for connections from clients. For each client, the *CoordinatorServer* creates a *SimulatorProxy* to communicate with it. After all the connections are received, the *CoordinatorServer* establishes the *modelToSim* and *coupInfo* and downloads them to *SimulatorProxies*. As *modelToSim* and *coupInfo* are kept in *SimulatorProxies* (not in the client simulators), all messages sent between clients will be first passed to *SimulatorProxies*. For example, in Figure 8, if *Atomic4* sends a message to *Coupled1*, the message will first be sent to *SimulatorProxy3*. Based on the *coupInfo* and *modelToSim*, *SimulatorProxy3* passes the message to *SimulatorProxy1*, which then sends the message to *CoordinatorClient1 (Coupled1)*.

As the coupling information of distributed models is kept in *SimulatorProxies*, the basic idea of implementing distributed coupling change is to update those *SimulatorProxies'* coupling information. To implement this, whenever an atomic model wants to add or remove a distributed coupling, the *CoupledSimulatorClient* for that atomic model generates a distributed coupling change request and sends it to the *SimulatorProxy* as shown as follows:

```
public void addDistributedCoupling(String src, String
p1, String dest, String p2){
    String dcc = Constants.addCouplingSymbol+":"
    +src+":"+p1+":"+dest+":"+p2;
    client.sendMessageToServer(dcc);
}
```

On the *SimulatorProxy*'s side, the *waitForMessageFromClient()* method is modified so that it can handle the distributed coupling change request. This method is shown as follows:

```
protected void waitForMessageFromClient() {
    String string = readMessageFromClient();
    //check to see if the message is a dynamic coupling
    change message
    if(string.startsWith(Constants.addCouplingSymbol)||
        string.startsWith(Constants.removeCoupling-
        Symbol))
            DynamicCouplingStrReceived(string);
    else{ // this is a regular DEVS message
            . . . . . . . . . . . // process the message
    }
}
```

The method checks to see if the received string starts with *addCouplingSymbol* or *removeCouplingSymbol*. If that is true, the received string is a distributed coupling change request, so the *DynamicCouplingStrReceived()* is called. Otherwise, the received string is a regular DEVS message, so the method processes it as usual. The *DynamicCouplingStrReceived()* method processes the string to get the source, the source's port, destination, and the destination's port of the coupling. Then it calls the *CoordinatorServer's addCoupling()* or *removeCoupling()* methods to update the coupling information of *SimulatorProxies*.

4.5 Add/Remove Ports

The operation of adding and removing ports dynamically is done by

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

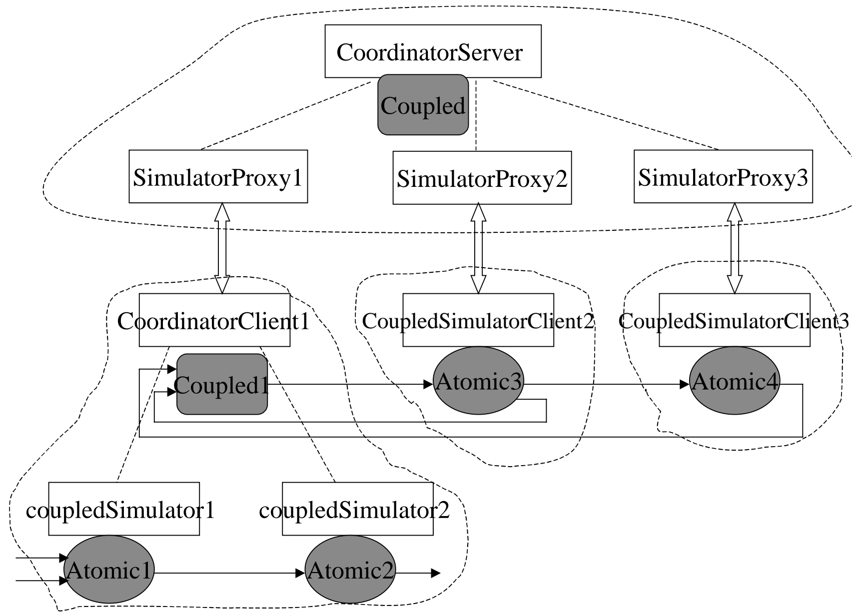


Figure 8. Models and their simulators in distributed simulation

- *addInport(String modelName, String portName)*
- *addOutport(String modelName, String port)*
- *removeInport(String modelName, String port)*
- *removeOutport(String modelName, String port)*

The functionality of modifying interfaces exists just at one horizontal level and is not present a level above (parent level) and a level below (brother's children). This restricts the ability of a model to alter the dynamics of the system to within its operations boundary. As mentioned above, the four forms of adding/removing inports/outports take the *modeName* as a parameter referring to the destination model to which the change is desired. The functioning of these methods can be seen in the *reconfigurable* GPT model. Internally, they are implemented as

```
public void addInport(String modelName, String port){
    digraph P = (digraph)getParent();
    IODevs iod = (IODevs)P.withName(modelName);
    if (P != null){
    if (iod instanceof atomic)
        iod.addInport(port);
    else
        ((digraph)iod).addInport(iod.getName(),port);
        }
}
```

The above function adds an input port to the model specified by the *modelName*. Inside the function, the model is accessed through the common parent (as they are brothers), and if it is an instance of an atomic model, then the port is added here directly; otherwise, the corresponding function in the digraph model is called, which adds the *port* to this brother digraph.

The mechanics of *addOutport()* is the same as that of *addInport()*. For the removal of ports, internally they are implemented in the same manner as the code described above, except that the line *iod.addInport(port)* is replaced by the line *iod.removeInport(port)*, where the variables have their usual meaning. The same situation happens in the case of *removeOutport()*, which is implemented on the same lines, with the change in the line mentioned above (*iod.removeOutport(port)*).

5. Conclusion

Variable structure capability provides a natural and effective way to model and simulate complex systems that exhibit structure, behavior, and interface changes to adapt to different situations. They also provide the additional flexibility to design and analyze a complex hierarchical system under development, as supported by the dynamic SES capability. In addition to the previously well-known structure operations, we introduced port (interface) alteration possibilities that greatly increase structure change flexibility. To maintain the hierarchical modular property of models, special attention has to be paid to the control of structure and interface changes. We introduced operation boundary constraints on structure change operations for this purpose. In general, as variable structure changes a component-based system during runtime, safety and security are a very important issue. More research on distributed reconfiguration and port-based structure transformation is needed to

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.

conduct safe and efficient dynamic change of component-based systems.

6. Acknowledgments

This research has been supported by NSF grant no. DMI-0122227, "Discrete Event System Specification (DEVS) as a Formal Modeling and Simulation Framework for Scaleable Enterprise Design."

7. References

[1] U.S. Department of Defense. 1998. High Level Architecture interface specification, version 1.3 (draft 9). Retrieved from http://hla.dmso.mil/hla/tech/ifspec/ifl-3d9b.doc

[2] U.S. Department of Defense. 1998. High Level Architecture object model template, version 1.3. Retrieved from http://hla.dmso.mil/hla/tech/omtspec/omtl-3d4.doc

[3] U.S. Department of Defense. 1998. High Level Architecture rules, version 1.3 (draft 2). Retrieved from http://hla.dmso.mil/hla/tech/rules/rulesl-3d2b.doc

[4] Miller, J. A., Y. Ge, and J. Tao. 1998. Component-based simulation environments: JSIM as a case study using Java beans. In *Proceedings of the Winter Simulation Conference*, vol. 1, pp. 373-81.

[5] Buss, A. 2002. Component based simulation modeling with SIMKIT. In *Proceedings of the Winter Simulation Conference*, vol. 1, pp. 243-9.

[6] Kilgore, R. A. 2000. Silk, Java and object-oriented simulation. In *Proceedings of the Winter Simulation Conference*, vol. 1, pp. 246-52.

[7] Balci, O., A. I. Bertelrud, C. M. Esterbrook, and R. E. Nance. Visual simulation environment. In *Proceedings of the Winter Simulation Conference*, vol. 1, pp. 279-87.

[8] Zeigler, B. P., T. G. Kim, and H. Praehofer. 2000. *Theory of modeling and simulation*. 2nd ed. New York: Academic Press.

[9] DEVS-Java Reference Guide. Retrieved from www.acims.arizona.edu

[10] Zeigler, Bernard P., Yoonkeon Moon, Doohwan Kim, and Jeong Geun Kim. 1996. DEVS-C++: A high performance modelling and simulation environment. *HICSS* 1:350-9.

[11] Kim, D., S. J. Buckley, and B. P. Zeigler. 1999. Distributed supply chain simulation in a DEVS/CORBA execution environment. In *Proceedings of WSC*, Phoenix, AZ.

[12] Zeigler, B. P., G. Ball, H. Cho, J. S. Lee, and H. Sarjoughian. 1999. Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions. In *Proceedings of the Simulation Interoperability Workshop*, Orlando, FL.

[13] Barros, F. J., and B. P. Zeigler. 1997. Adaptive queueing: A case study using dynamic structure DEVS. *International Transactions in Operational Research* 4 (2): 87-98.

[14] Barros, F. J., M. T. Mendes, and B. P. Zeigler. 1994. Variable DEVS-variable structure modeling formalism: An adaptive computer architecture application. In *Proceedings of te Fifth Annual Conference on Distributed Interactive Simulation Environments*, December, pp. 185-91.

[15] Uhrmacher, A. M. 1993. Variable structure models: Autonomy and control—Answers from two different modeling approaches. In *Proceedings on AI, Simulation, and Planning in High Autonomy Systems*, pp. 133-9.

[16] Uhrmacher, A. M. 2001. Dynamic structures in modeling and simulation: A reflective approach. *ACM Transactions on Modeling and Simulation* 11 (2): 206-32.

[17] Pawletta, T., and B. Lampe. 2002. *A DEVS-based approach for modeling and simulation of hybrid variable strucature systems*. Lecture Notes in Control and Information Sciences No. 279. New York: Springer.

[18] Brown, A. W., and K. C. Wallnau. 1998. The current state of CBSE. *IEEE Software* 15 (5): 37-46.

[19] Zeigler, B. P., and R. G. Reynolds. 1985. Towards a theory of adaptive computer architectures. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pp. 468-75.

[20] Zeigler, B. P., and A. Louri. 1993. A simulation environment for intelligent machine architecture. *Journal of Parallel and Distributed Computing* 18:77-88.

[21] Chean, M., and L. A. B. Fortes. 1990. A taxonomy of reconfigurable techniques for fault-tolerant processor arrays. *IEEE Computer* 23 (1): 55-69.

[22] Chen, X. 2002. Dependence management for dynamic reconfiguration of component-based distributed systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pp. 279-84.

[23] Zeigler, B. P. 1989. Concepts for distributed knowledge maintenance in variable structure models. In *Modelling and simulation methodology: Knowledge systems paradigm*, edited by B. Zeigler, M. Elzas, and T. Oeren, 45-54. Amsterdam: Elsevier.

[24] Barros, F. J. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation* 7 (4): 501-15.

[25] Zhang, Y., M. Fromherz, L. Crawford, and Y. Shang. 2002. A general constraint-based control framework with examples in modular self-reconfigurable robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland.

[26] Butler, Z., R. Fitch, and D. Rus. 2002. Distributed goal recognition algorithms for modular robots. In Proc. of IEEE ICRA, 2002.

[27] Hu, X., and B. P. Zeigler. 2004. Model continuity to support software development for distributed robotic systems: A team formation example. Submitted to *Journal of Intelligent & Robotic Systems, Theory & Application*, pp. 71-87, January.

[28] Peipelman, J., N. Alvarez, K. Galinet, and R. Olmos. 2002. 498 A & B technical report. Department of Electrical and Computer Engineering, University of Arizona.

[29] Zeigler, B. P. 1990. *Object-oriented simulation with hierarchical, modular models: Intelligent agents and endomorphic systems*. New York: Academic Press.

[30] Couretas, J., B. P. Zeigler, and U. Patel. 1999. Automatic generation of system entity structure alternatives: Application to initial manufacturing facility design. *Transactions of the SCS* 16 (4): 173-85.

[31] Cho, Y. K., X. Hu, and B. P. Zeigler. 2003. The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems. *SIMULATION* 79 (4).

[32] Cho, Y. K. 2001. RTDEVS/CORBA: A distributed object computing environment for simulation-based design of real-time discrete event systems. Ph.D. diss., University of Arizona, Tucson.

[33] Zeigler, B. P., H. Sarjoughian, and W. Au. 1997. Object-oriented DEVS. In *Proceedings of Enabling Technology for Simulation Science*, Orlando, FL.

Xiaolin Hu *is an assistant professor in the Computer Science Department at the Georgia State University, Atlanta, GA.*

Bernard P. Zeigler *is a professor in the Electrical and Computer Engineering Department at the Arizona Center for Integrative Modeling and Simulation, University of Arizona, Tucson.*

Saurabh Mittal *is a PhD candidate in the Electrical and Computer Engineering Department at the Arizona Center for Integrative Modeling and Simulation, University of Arizona, Tucson.*

Downloaded from http://sim.sagepub.com at Ebsco Electronic Journals Service (EJS) on April 10, 2008
© 2005 Simulation Councils Inc.. All rights reserved. Not for commercial use or unauthorized distribution.