# Dynamic Reconfiguration in DEVS Component-based Modeling and Simulation[1]

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

## Abstract

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

## Keywords

Dynamic reconfiguration, Component-based modeling and simulation, variable structure modeling, 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 environment 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 technology, different simulation environments can interact through standard interfaces and work together in architecture such as HLA [1,2,3]. Thus interoperability is achieved and more powerful simulation can be conducted. Component-based technology also makes the modeling of a complex system more easily managed, since a complex system can be divided to several manageable pieces, each of them referring to as a component. It also promotes distributed simulation as component-based technology has a natural fit to distributed environment.

---

1 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"

Motivated by these advantages, various component-based modeling and simulation environments have been developed. Furthermore, the HLA architecture 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. As these interfaces comply with the same HLA interface specification, federates developed by different developers can communication with each other via the Run-Time Infrastructure. As HLA promotes component-based modeling and simulation from the architecture point of view, 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 utilizes Java and Java beans technology to support component-based modeling and simulation. A visual design interfaces is provided for users to develop and to assemble components.

DEVS (Discrete Event System Specification) [8] supports component-based modeling and simulation not only by providing the environments to enable component-based modeling and simulation, but also by emphasizing the theory of hierarchical modular modeling. In DEVS, 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 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 sub-component 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, DEVS component-based modeling and simulation environment does not rely on the underline implementation language. In fact, various DEVS environments such as DEVSC++, DEVSJAVA, DEVSCorba, etc [9,10,11]. 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 a complex system, the structure and behavior could be very complex as the system may continuously reconfigure itself 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 which typically actively evolve over time to adjust to the new environment. To model these complex systems, a dynamic reconfiguration modeling capability is needed. As dynamic reconfiguration greatly enhances the modeling capability, it also raises special design issues for component-based modeling and simulation environments.

In this paper, we discuss dynamic reconfiguration, specifically variable structure and interface changing capability, in DEVS component-based modeling and simulation. As previous work [13,14,15,16] has established a theoretical background for variable structure of DEVS, this paper discusses it in the context of

component-based technology and covers more aspects of DEVS dynamic reconfiguration. The paper first elaborates on the role of dynamic reconfiguration in component-based modeling and simulation. This role is then illustrated by three examples that illustrate how variable structure is support by DEVSJAVA 3.0 (www.acims.arizona.edu). After that, the paper discusses how dynamic reconfiguration can be implemented in a DEVS modeling and simulation environment. Finally, the conclusions are drawn and some open issues are discussed.

## 2. Dynamic Reconfiguration in DEVS Environment

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 [17]. A component system is built by composition of individual components and establishing relationship among them. As each component holds a high degree of autonomy and has well defined interfaces, dynamic reconfiguration of components can be achieved during runtime. For component-based modeling and simulation, dynamic reconfiguration provides several advantages. First, it provides a natural and effective way to model those complex systems which exhibit structure and behavior changes to adapt to different situations. Examples of these systems include distributed computing systems, reconfiguration computer architectures [18,19], fault-tolerance computers [20] and ecological systems [15]. For these systems, structure changing and component upgrading is an essential part of the systems. Without the dynamic reconfiguration capability, it's very hard, if not impossible, to model and simulate these systems. Secondly, from the design point of view, dynamic reconfiguration provides the additional flexibility to design and analyze a system under development. For example, as will be illustrated later, dynamic reconfiguration gives us the flexibility to design and simulate a distributed robotic system in which robots form relationship dynamically. Thirdly, dynamic reconfiguration makes it possible to load only a sub-set of system's components for simulation. This is very useful to simulate very large systems with tremendous number of components as only the active components can be loaded dynamically to conduct simulation. Otherwise, the entire system has to be loaded before simulation begins.

In general, there are six forms of reconfiguration of component-based systems [21]: 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 which might have 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 paper, we are

also interested in 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. As it is usually researched in mobile agent systems, it is not discussed in this paper.

Variable structure models are the models which can dynamically change their model structure such as the inner components of the model and the relationship between those components. Figure 1 gives an example which 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 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 DEVS models; *addCoupling()/removeCoupling()* to add/remove DEVS couplings. Note that the *addCoupoling()* and *removeCoupling()* methods take four parameters: the source model, source model's output port, the destination model, destination model's input port. With these methods, the structure change process showed in Figure 1 can be realized as below:

*(1) addModel(C);*

*(2) addCoupling(C, COutputPort, B, BInputPort);*

*(3) removeModel(A);*

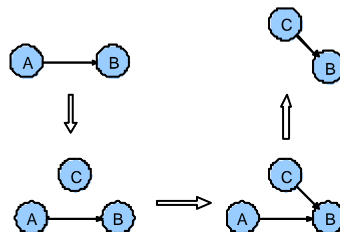


Figure 1. A variable structure process

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, so 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 interested conditions. For the system showed in Figure 1, component *B* could be the one to monitor 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 $A$.

**Operation Boundaries**

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 distributed environment. To support operations boundaries in DEVS, models can maintain information on their locations in relation to the hierarchical structure of the overall coupled model. A component of a coupled model has knowledge of its parent. 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 [22].

Thus the structure change operations also need to work within this hierarchical structure and to maintain this structure. Based on 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 be easily seen that a model can remove itself, but it cannot remove its parent. Our approach differs from that formalized by Barros [23] who restricts the ability to initiate change to a central network executive. 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. Since a network executive can be emulated by designating an atomic model to be the only source of structure change, the class of models defined by Barros is included in our formulation. Further, this inclusion allows us to benefit from his formal characterization of variable structure DEVS.

We note that operations boundaries are defined in terms of 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's 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].

**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 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 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 In order not to violate the autonomy property of a component, 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, though 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 treat ports adhere to a labeling scheme such as *name+index* which can be analyzed and interpreted. .Detailed explanation will be given in the section 3.3. As a new feature of DEVS dynamic reconfiguration, more research is needed to answer questions such as how to provide a general mechanism to update a model's external transition function and output function accordingly after the model's input and output ports are added/removed dynamically.

## 3. Examples of Dynamic Reconfiguration

To illustrate the role of dynamic reconfiguration 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 which dynamic 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 [24,25], a group of robots which can change their shape have been reported. As those robots change the hardware components of themselves, in this section we describe a distributed multi-robotic system which changes its software components. This system exhibits dynamic team formation in which independent robots form teams

[2] 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.

dynamically and then conduct *Leader-Follower* march. We first describe a system which has been realized by two real mobile robots [26]. Then we discuss a more scalable system with indefinite number of robots. The robot we use in the example is a car type mobile robot with wireless communication capability [27].

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 each other.

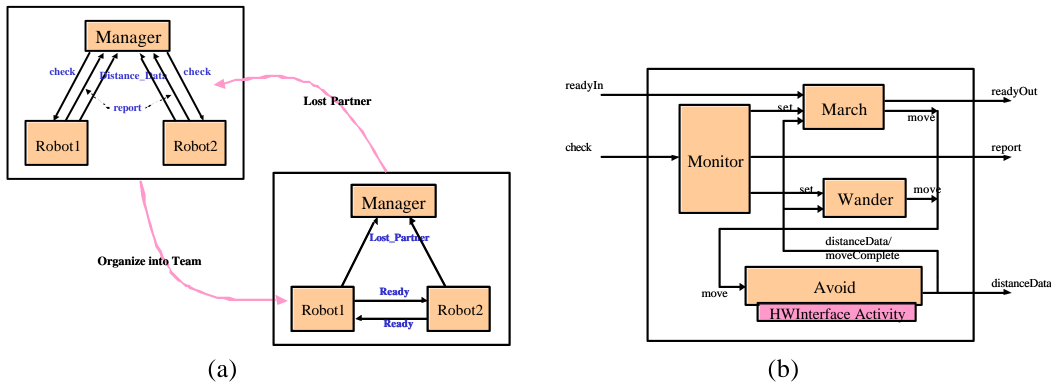



Figure 2: Model of Dynamic Team Formation System

From above description, we can recognize three basic components in this system: the *Manager* which resides on a laptop (computer), *robot1* and *robot2* which reside on mobile robots. Figure 2(a) shows the model of this system. In this system, the *Manager* is an atomic model and each *robot* is a coupled model as showed in Figure 2(b). More description about the *robot* model can be found in [26]. The coupling of the system is as follows: (*R1* stands for *robot1*; *R2* stands for *robot2* and *man* stands for *Manager)*:

*addCoupling(R1, "distanceData", man, "Robot1Data");*

*addCoupling(R1, "report", man, "Robot1Report");*

*addCoupling(man, "Robot1Check", R1, "Check");*

*addCoupling(R2, "distanceData", man, "Robot2Data");*

*addCoupling(R2, "report", man, "Robot2Report");*

*addCoupling(man, "Robot2Check", R2, "Check");*

As we can see there is no coupling between *robot1* and *robot2*. Each robot has output ports d*istanceData* and *report*. These ports are coupled to *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-line-of-sight. The robots return the check result using the *report* port. Once the report messages returned from the robots are both positive, this means two robots are close and they see each other. In this case, the *Manager* will change the couplings of the system dynamically in order to establish a direct connection between the two

robots. Specifically in this example, the manager executes the following DEVSJAVA code:

```
removeCoupling("Robot1", "distanceData", " Manager" , "Robot1Data");
removeCoupling (" Manager" , "Robot1Check", "Robot1", "Check");
removeCoupling ("Robot2", "distanceData", " Manager" , "Robot2Data");
removeCoupling (" Manager" , "Robot2Check", "Robot2", "Check");
addCoupling("Robot1", "readyOut", "Robot2", "readyIn");
addCoupling("Robot2", "readyOut", "Robot1", "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 string names. Explicit references can also be obtained from the parent coupled model by supplying the string names. This requires that all models are given unique names. After executing the DEVSJAVA code, bi-directional connection is established by coupling two robots' *Ready* port to each other, so two robots can communicate directly. The *distanceData* and *Check* couplings between robots and *Manager* are removed because they are no longer needed during the process of 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 *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 situation as it is initially started, where two robots move independently and try to find each other.

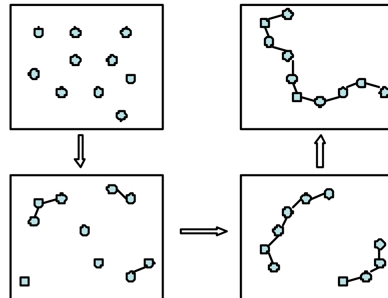


Figure 3: A Scalable Dynamic Team Formation Example

As the above system only includes two robots, more scalable systems with indefinite number of robots can be developed based on the same dynamic reconfiguration idea. Figure 3 shows an example with ten 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.

### 3.2 Dynamically Emulate the System Entity Structure (SES)

The System Entity Structure (SES) provides a way for specifying system composition[28] 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 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 which demonstrates how this can be achieved by employing the variable structure capabilities.

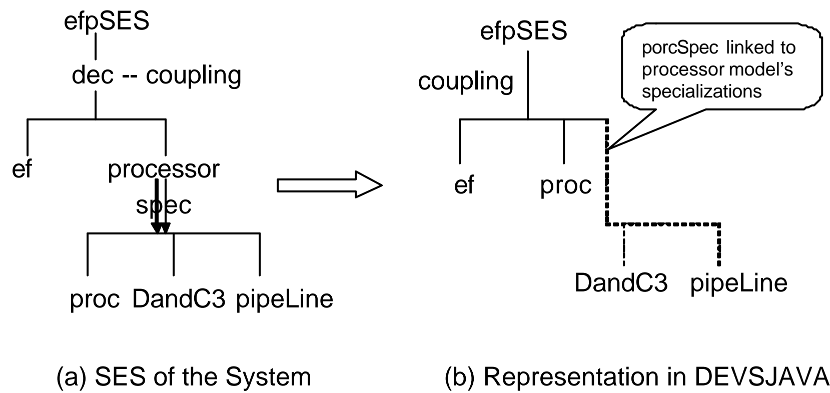



Figure 4. Dynamically Emulate the System Entity Structure (SES)

This example system is *efpSES* as shown in Figure 4(a). It has two components: an experimental frame model *ef* and a processor model which 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 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 4(b), the user adds *procSpec* to the coupled model and tells it which component to control. 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 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 [29].

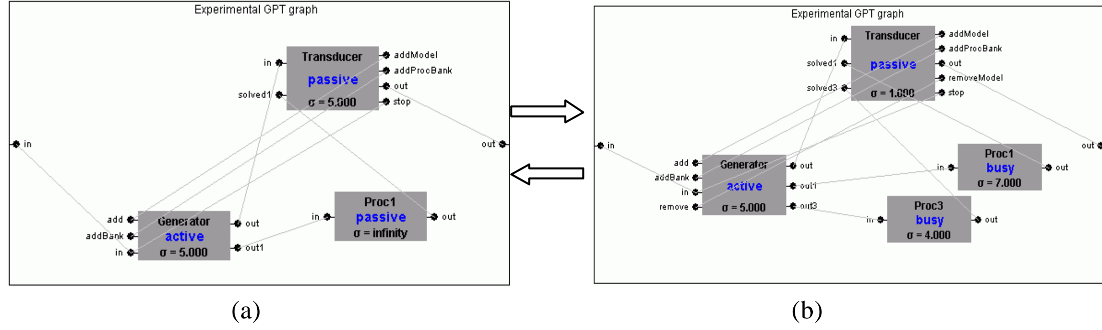
While the SES involves only replacement of components by alternatives, the approach can be further extended to allow a restructuring executive to observe the simulation and make decisions regarding the alternatives to employed based on prevailing conditions. Such restructuring is discussed in the following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

### 3.3 A Reconfigurable Workflow System

A simple workflow prototype is referred to as GPT.[3] This is a coupled model that is composed of a *Generator*, a *Processor*, and a *Transducer*. It is the simplest

[3] See gpt.java in the SimpArc package of DEVSJAVA.

self-contained model that simulates three basic components of any workflow system. *Generator* generates jobs. *Processor* processes them and *Transducer* keeps track of the system state as a whole and gives computes 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 *Processor*(s) can be dynamically added or removed and *Generator* and *Transducer* change their interfaces accordingly.




(a) (b)

Figure 5: Stages of the Reconfigurable GPT System

As shown in Figure 5(a), this system starts with the basic GPT components: *Generator*, *Proc1* and *Transducer*. *Generator* generates jobs and sends them out through *out1* port coupled to the *Proc1's in* port. *Proc1* executes the job and sends the solved job to *Transducer* at *solved1* port. Notice 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 *Proc1* can't handle all the generated jobs, it sends out a message to the *Generator*, which then adds another processor *Proc3*. As shown in Figure 5(b). *Proc3* is in a similar position as *Proc1* in the system. Notice that the interfaces of *Generator* and *Transducer* also change accordingly. Besides *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 processor (*remove* and *removeModel* port). 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 wasn't any functionality to remove any model as there was no need of it. As new processors are added so is the corresponding functionality to remove them is also added at the same time. Typical set of commands that were executed by the *Generator* after receiving the addition message from the *Transducer* are:

```
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 run time 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) which 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 dynamic reconfiguration is based on earlier development 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 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 refer to the software components which are defined by the users to capture the behavior and structure aspects of the system under development. DEVS simulators are provided by the DEVS simulation environment to simulate or execute DEVS models.

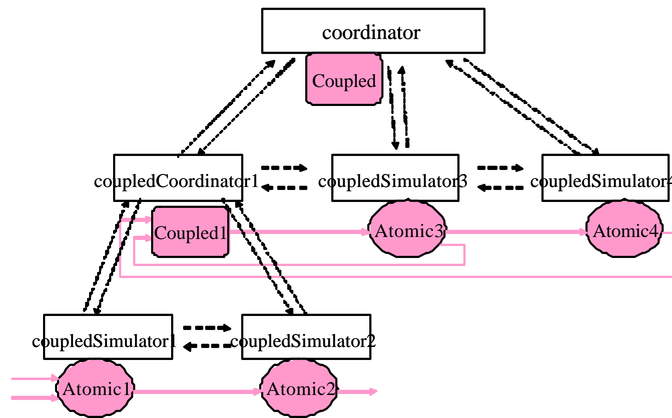



Figure 6. Relationship between models and their simulators (fast-mode simulation case)

Corresponding to the hierarchical structure of a DEVS model, its simulators also form a hierarchical structure. Figure 6 gives an example which shows the relationship of a hierarchical coupled model and its corresponding simulators. This model has three components: *Atomic3*, *Atomic4*, and *Coupled1* which has two sub-components: *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 sub-simulators, which have a one to one relationship to the components of the coupled model. Another important role for the *coordinator* is to set up the system such as to establish data structures, create and initialize all sub-simulators and to control the overall simulation cycle. Follow the hierarchical structure of a coupled model, there is a *coupledSimulator* assigned to each atomic model; a *coupledCoordinator* assigned to each coupled 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. In the hierarchical structure, as each model and simulator manages its information locally, there is no need for the system to build global data structures In fact, in the implementation, each model and simulator has its own data structures to store its local information.

First let's see the data structure managed by DEVS coupled models (atomic models don't need them). This is straightforward because coupled models need to keep track of their sub-components and the couplings among them. Thus, each coupled model has two variables as defined below:

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

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

- 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* and *coordinator* (*coupledSimulator* doesn't 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* are used by a simulator to store its model's coupling information. Specifically, *coupInfo* stores the couplings which start from its model and ends to the model's brothers or parent. *extCoupInfo* is used by a *coordinator* or *coupledCoordinator* (*coupedSimulator* doesn't use it) to store the couplings which start from its model and ends to the model's children models. Use *coupledCoordinator1* in Figure 6 as an example, the *coupInfo* has one coupling instance which starts from *Coupled1* and ends to *Atomic3*; the *extCoupInfo* has 2 coupling instances. Both of them start from *Coupled1* and end to *Atomic1*. The third group of variables *modelToSim* and *internalModelTosim* are used by simulators to store the model-simulator mapping information. Again, use *coupledCoordinator1* in Figure 6 as an example, the *modelToSim* has three instances: *(Coupled1,coupledCoordinator1)*, *(Atomic3,coupledSimulator3)*, and *(Atomic4,coupledSimulator4)*; the *internalModelToSim* has two instances: *(Atomic3,coupledSimulator1)*, and *(Atomic2, coupledSimulator2).*

These data structures form a mini-database which stores all the local information about the structure and relationship among models and simulators. To efficiently implement the variable structure capability, which allows couplings and models (and their simulators) to be added or removed dynamically, it's very important to well understand these data structures.

### 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 *cddCoupling()* 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 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* or *coupledCoordinator's addCoupling()* method. This method uses the source model's name to find the corresponding simulator and then update that simulator's coupling information, the *coupInfo* or *extCoupInfo* variable.

### 4.3 Add/Remove Model Dynamically

Different from add couplings dynamically, adding a model dynamically means not only a new model is added, but also a new simulator needs to be created and added into the system. Furthermore, the new simulator has to be initialized and synchronized with the ongoing simulation system. The *addModel()* method is shown below:

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

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

```
public void setNewSimulator(IOBasicDevs iod){
     if(iod instanceof atomic){    //do a check on what model it is
          coupledSimulator s = new coupledSimulator(iod);
            ………//update the corresponding data structures;
            s.initialize(getCurrentTime());
      }
     else if(iod instanceof digraph){
            coupledCoordinator s = new coupledCoordinator((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). Then it updates the corresponding data structures such as *simulators*, *internalModelToSim*, and *modelToSim*. Finally it initializes the created simulator. In order to synchronize with the current simulation time, the *initialize()* method takes the parameter of *getCurrentTime()* which returns the current simulation time. Since all simulator and coordinator updating is done immediately, a model can be added safely during the internal or external transition functions of an existing brother. It becomes eligible to participate in the subsequent simulation cycle, contributing to the determination of the global time of next event and able to receive inputs and generate outputs in the normal manner. Further details on modification of the DEVS protocol needed for well-defined variable structure are given in [30].

Reverse to what adding a model means, removing a model dynamically means to remove a model and its corresponding simulator(s) from the system. Furthermore, removing a model also implies that 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, then it calls *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. Figure 7 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 atomic model; *CoordinatorClient* for coupled model). These clients connect to an *CoordinatorServer*, which may reside on another computer. During initialization, the *CoordinatorServer* waits for connections from clients. For each client, the *CoordinatorServer* creates a *SimulatorProxy* to communication with it. After all the connections are received, the *CoordinatorServer* establishes the *modelToSim* and *coupInfo* and download them to *SimulatorProxies*. As *modelToSim* and *coupInfo* are kept in *SimulatorProxies* (not in the client simulators), all messages sent between clients will be firstly passed to *SimulatorProxies*. For example in Figure 7, 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 *CoordinatorClient (Coupled1).*

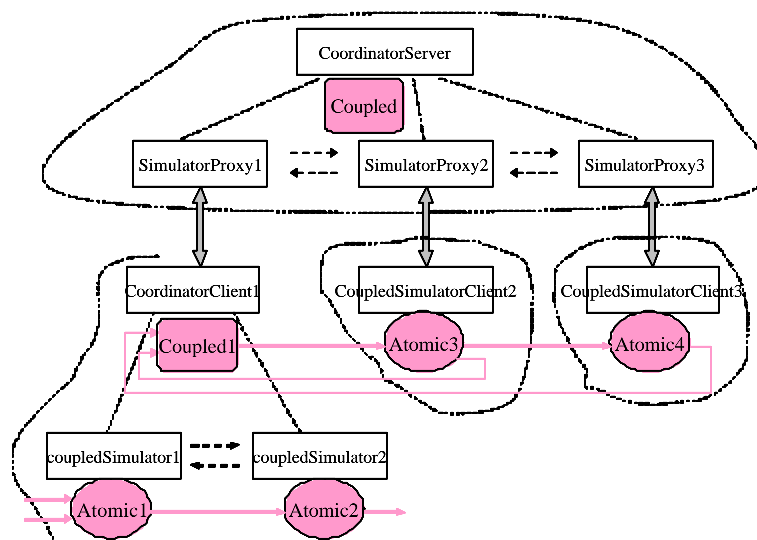



Figure 7. Models and their simulators in distributed simulation

As the coupling information of distributed models are kept in *SimulatorProxies*, so 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 below:

```
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 below:

```
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.removeCouplingSymbol))
            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 s 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, source port, destination, and destination port of the coupling. Then it call *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:

- *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 (brothers 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, each functions takes 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 models is accessed through the common parent (as they are brothers) and if its an instance of atomic, 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 exactly 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. Same thing 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 which 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 previously well-known structure operations we introduced port (interface) alteration possibilities that greatly increase structure change flexibility. In order not to violate the autonomy property of a component-based system, 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 dynamic reconfiguration change a component-based system during runtime, safety and security is a very important issue. More research on distributed reconfiguration and port-based structure transformation is needed to conduct safe and efficient dynamic change of component-based systems.

## Reference

1. U.S. Department of Defense, "High Level Architecture Interface Specification", Version 1.3 (Draft 9), February 1998. http://hla.dmso.mil/hla/tech/ifspec/ifl-3d9b.doc

2. U.S. Department of Defense, "High Level Architecture Object Model Template", Version 1.3, February 1998. http://hla.dmso.mil/hla/tech/omtspec/omtl-3d4.doc
3. U.S. Department of Defense, "High Level Architecture Rules", Version 1.3 (Draft 2), February 1998. http://hla.dmso.mil/hla/tech/rules/rulesl-3d2b.doc
4. Miller, J.A.; Y. Ge; J. Tao, "Component-based simulation environments: JSIM as a case study using Java Beans", *Simulation Conference Proceedings, 1998*. Winter , Volume: 1 , 13-16 Dec 1998 Page(s): 373 -381 vol.1
5. Buss, A., "Component based simulation modeling with simkit", *Winter Simulation Conference, 2002. Proceedings of the* , Volume: 1 , 2002 Page(s): 243 -249
6. Kilgore, R.A., "Silk, Java and object-oriented simulation", *Simulation Conference Proceedings, 2000*. Winter , Volume: 1 , 2000 Page(s): 246 -252 vol.1
7. Balci, O, et al. "Visual Simulation Environment", *Simulation Conference Proceedings, 1998*. Winter , Volume: 1 , 13-16 Dec 1998 Page(s): 279 -287 vol.1
8. Zeigler, B.P., T.G. Kim, and H. Praehofer.: *Theory of Modeling and Simulation*. 2 ed. 2000, New York, NY: Academic Press
9. B. P. Zeigler, et al., "DEVS-C++: A High Performance Modeling and Simulation Environment", *29th Hawaii International Conference on System Sciences* (HICSS'96) Volume 1: Software Technology and Architecture, January 03 - 06, 1996
10. DEVS-Java Reference Guide, www.acims.arizona.edu
11. D. Kim, S.J. Buckley, and B.P. Zeigler. " Distributed Supply Chain Simulation in a DEVS/CORBA Execution Environment," in *Proceeding of WSC*. Phoenix, Arizona, 1999
12. B.P. Zeigler, *et. al*., " Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions," in *Proceedings of Simulation Interoperability Workshop,* Orlando, Florida 1999.
13. Barros, F.J. and B.P. Zeigler, "Adaptive Queueing: A Case Study Using Dynamic Structure DEVS". *International Trans. in Oper. Res.*, 1997. Vol. 4, No. 2, pp 87-98
14. Barros, F.J.; Mendes, M.T.; Zeigler, B.P., "Variable DEVS-variable structure modeling formalism: an adaptive computer architecture application". *'Distributed Interactive Simulation Environments'., Proceedings of the Fifth Annual Conference on* , 7-9 Dec 1994 Page(s): 185 -191
15. Uhrmacher, A.M. " Variable Structure Models: Autonomy and Control – Answers from Two Different Modeling Approaches" .*Proc. AI, Simulation, and Planning in High Autonomy Systems*. IEEE Computer Society Press, 1993, 133-139
16. Uhrmacher, A.M., "Dynamic Structures in Modeling and Simulation - A Reflective Approach". *ACM Transactions on Modeling and Simulation, Vol.11. No.2* , p. 206-232, April 2001.
17. Brown, A.W.; Wallnau, K.C.;"The current state of CBSE", *IEEE Software* , Volume: 15 Issue: 5 , Sep/Oct 1998 Page(s): 37 -46
18. Zeigler, B.P. and R.G Reynolds. " Towards a Theory of Adaptive Computer Architectures" . Proc. *5th International Conference on Distributed Computing Systems*. Computer Society Press, 1985, 468-475
19. Zeigler, B.P. and A. Louri. " A Simulation Environment for Intelligent Machine Architecture". *Journal of Parallel and Distributed Computing* 18: 1993, 77-88
20. Chean, M. and L.A.B. Fortes. " A Taxonomy of Reconfigurable Techniques for Fault-Tolerant Processor Arrays". *IEEE Computer* 23, no. 1 (1990, Dec.): 55-69
21. Chen, X., "Dependence management for dynamic reconfiguration of component-based distributed systems", *Automated Software Engineering*, 2002. Proceedings. ASE 2002. 17th IEEE

International Conference on , 2002 Page(s): 279 -284

22. "Concepts for Distributed Knowledge Maintenance in Variable Structure Models," in: *Modelling and Simulation Methodology: Knowledge System Paradigms*, (Eds. M.S. Elzas, B.P. Zeigler and T.I. Oren), North Holland Pub. Co., Amsterdam, pp. 45-54, 1989.
23. Barros. F.J. "Modeling Formalisms for Dynamic Structure Systems". *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 4, 501-515, 1997
24. Zhang, Y., et al., "A General Constraint-Based Control Framework with Examples in Modular Self-Reconfigurable Robots", *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, Lausanne, Switzerland, Oct. 2002
25. Z. Butler, R. Fitch and D. Rus, "Distributed Goal Recognition Algorithms for Modular Robots" *ICRA* 2002
26. Hu, X., B.P. Zeigler, "Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example", submitted to *Journal of Intelligent & Robotic Systems, Theory & Application*
27. Peipelman. J., et al.: "498 A & B Technical Report". *Department of Electrical and Computer Engineering, University of Arizona*, 2002
28. Zeigler, B.P. 1990. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. Academic Press.
29. Couretas, J., B. P. Zeigler, U. Patel, "Automatic Generation of System Entity Structure Alternatives: Application to Initial Manufacturing Facility Design." *Transactions of the SCS*, 1999,16(4), pp. 173-185.
30. Zeigler, B.P., H. Sarjoughian, and W. Au. "Object-Oriented DEVS", *Proc. Enabling Technology for Simulation Science*, SPIE AeoroSense 97. 1997. Orlando, FL.