Model Continuity in the Design of Dynamic Distributed Real-Time Systems

Xiaolin Hu, *Member, IEEE*, and Bernard P. Zeigler, *Fellow, IEEE*

***Abstract*—Model continuity refers to the ability to transition as much as possible a model specification through the stages of a development process. In this paper, the authors show how a modeling and simulation environment, based on the discrete event system specification formalism, can support model continuity in the design of dynamic distributed real-time systems. In designing such systems, the authors restrict such continuity to the models that implement the system's real-time control and dynamic reconfiguration. The proposed methodology supports systematic modeling of dynamic systems and adopts simulation-based tests for distributed real-time software. Model continuity is emphasized during the entire process of software development—the control models of a dynamic distributed real-time system can be designed, analyzed, and tested by simulation methods, and then smoothly transitioned from simulation to distributed execution. A dynamic team formation distributed robotic system is presented as an example to show how model continuity methodology effectively manages the complexity of developing and testing the control software for this system.**

***Index Terms*—Discrete event system specification (DEVS), distributed real-time systems, dynamic reconfiguration, model continuity, modeling and simulation, robotic team formation.**

I. Introduction

DISTRIBUTED real-time systems continuously and autonomously control, and react to, the external environments. As they usually operate in dynamic environments, they tend to exhibit dynamic reconfiguration by changing their structures and operation modes in response to different situations. This is exemplified by many manufacturing control systems, distributed robotic systems, and intelligent sensor networks with mobile devices and computing nodes. The software to control such dynamic distributed real-time systems is extremely hard to design and difficult to verify due to several factors such as their multiple task synchronization as well as dynamic reconfiguration.

To handle the complexity of developing real-time software systems, modeling and simulation methods are frequently used. While simulation methods help to analyze and design the systems under development, they face a common deficiency—that the simulation models are discarded as unusable by the implementation stage. In fact, in most cases, the implementation artifacts are not derived in any direct way from the simulation models. This discontinuity between the implementation artifacts and analysis, design, and modeling artifacts is a common deficiency of most design methods [1], [2]. It results in inherent inconsistency among design, test, and implementation artifacts. Design of real-time systems can be improved by supporting consistent artifacts among these design stages, a quality referred as "model continuity." This quality of model continuity becomes ever more important as real-time systems become more and more complex. For example, a dynamic distributed real-time system might include hundreds of computing nodes, smart sensors, and actuators, and continuously reconfigure itself in an uncertain or even hostile environment. Without the support of model continuity, it is very difficult to manage the software's complexity during development of systems of this kind.

This paper describes a simulation-based software development methodology for distributed dynamically reconfiguring real-time systems. This methodology supports model continuity because the same control models that are designed and tested by simulation methods can be deployed to the real target system for execution. The methodology is based on discrete event system specification (DEVS) modeling and simulation framework [3], specifically the DEVSJAVA [4] environment. It provides a "modeling–simulation–execution" process that includes several stages to develop real-time software. Variable structure modeling capability [5]–[7] has been integrated into the methodology so that real-time systems with self-reconfiguration capabilities can be naturally modeled and designed. Furthermore, stepwise simulation methods have been developed so that the control models of a real-time system can be tested and analyzed incrementally. Because the control model remains unchanged from the design stage to implementation stage, no transformation or reconstruction is needed. With this approach, designers can have more confidence that the final system in operation is the system that was designed and will carry out the functions as tested by simulation.

This paper is organized as follows. Section II discusses related work in model continuity and real-time system development. Section III elaborates the model continuity methodology for dynamic distributed real-time systems. Section IV shows how a simulation-based testing process can be applied to incrementally test the control models under development. Section V discusses the features that have been implemented to support model continuity in the DEVSJAVA environment. Section VI describes a robotic "dynamic team formation" example to demonstrate the methodology. Section VII provides conclusions.

Manuscript received May 30, 2003; revised February 7, 2004. This work was supported by the National Science Foundation under Grant DMI-0122227, "Discrete Event System Specification (DEVS) as a Formal Modeling and Simulation Framework for Scaleable Enterprise Design." This paper was recommended by Associate Editor W. A. Gruver.

X. Hu is with the Computer Science Department, Georgia State University, Atlanta, GA 30303 USA (e-mail: xhu@cs.gsu.edu).

B. P. Zeigler is with the Arizona Center for Integrative Modeling and Simulation, Electrical and Computer Engineering Department, University of Arizona, Tucson, AZ 85721 USA (e-mail: zeigler@ece.arizona.edu).

Digital Object Identifier 10.1109/TSMCA.2005.851283

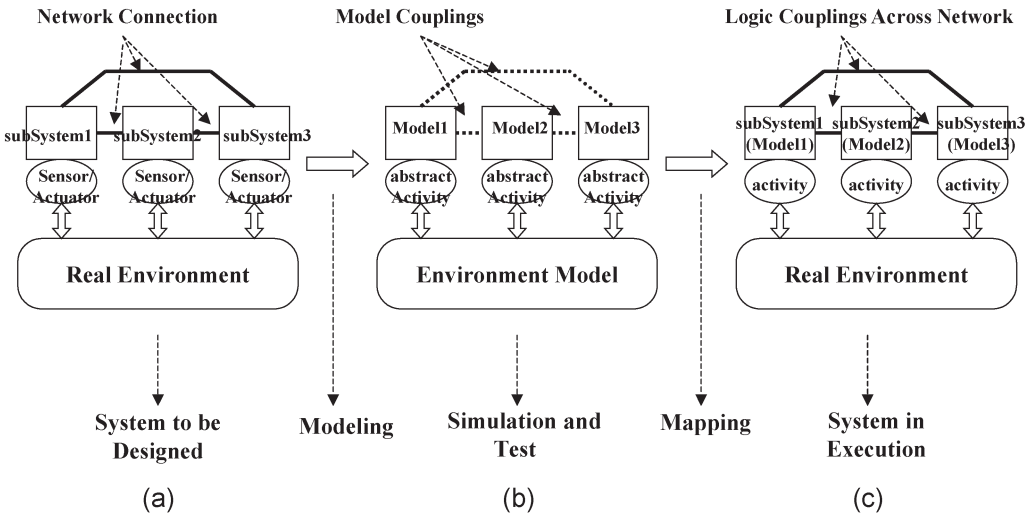


Fig. 1. Modeling, simulation, and execution of a distributed real-time system.

II. Related Work

Ensuring consistency among different development stages has been a research issue in various areas of design. In software engineering, traceability, in the form of requirements traceability [8] or design-code traceability [9], has been advocated to ensure consistency among software artifacts of subsequent phases of the development cycle. Boyd [10] shows how traceability can be achieved when designing reactive systems. In hardware/software codesign, Janka *et al.* [11] described a methodology that allows the specification stage and design stage to work together coherently when designing embedded real-time signal processing systems. The preceding approaches use different artifacts in different stages. In contrast, our approach allows the same simulation models to be used in the design and implementation stages. The following research efforts are more closely related to our work by applying simulation-based design. Bagrodia and Shen [12] describe an approach that supports the design of distributed systems via iterative refinement of a partially implemented design where some components exist as simulation models and others as operational subsystems. Gonzalez and Davis [13] present a simulation and control tool that provides the capability to model, as well as to control, real-world systems. Our work extends the applicability of simulation-based design in new significant directions. It is based on a formal modeling and simulation framework that supports variable structure modeling. Furthermore, it provides a stepwise simulation-based test process to allow the control models of a real-time system to be tested and analyzed incrementally. This incremental testing process extends Bagrodia and Shen's work on iterative transformation of performance models by focusing on system testing as opposed to performance evaluation.

Other methods for real-time software system development have focused on exploring the modeling capabilities for real-time systems. For example, the unified modeling language for real time (UML-RT) [14] extends UML models to address special aspects of designing real-time systems. Kim [15] uses the time-triggered message-triggered object (TMO) model to capture the timeliness and concurrency properties of real-time software systems. Several other models and design methods for real-time systems have been surveyed by Gomaa [16], [17]. None of the above methods use simulation-based design in their approaches. The Ptolemy project [18], while supporting heterogeneous modeling and simulation for concurrent embedded systems design, does not address the issue of model continuity throughout the development process.

III. Model Continuity Methodology

Model continuity refers to the ability to transition as much as possible a model specification through the stages of a development process. In the context of this paper, it means that the control models of a dynamic distributed real-time system can be designed and tested in a simulation environment, and then, without essential change, be executed, as the working software, in a distributed environment [19], [20].

A. Modeling a System and Its Variable Structure

Real-time systems are computer systems that monitor, respond to, or control an external environment. This environment is connected to the computer system through sensors, actuators, and other input–output interfaces [21]. A real-time system from this point of view consists of sensors, actuators, and the real-time control and information-processing unit. A distributed real-time system is composed of a collection of subsystems. Fig. 1(a) shows an example with three subsystems. As shown in Fig. 1(a), each subsystem has its own control and information processing unit and interacts with the external environment through sensors/actuators. These subsystems are connected by a network and communicate and cooperate to finish systemwide tasks. For dynamic distributed real-time systems, the connections among their subsystems may change dynamically, resulting in different system configurations. Distributed real-time systems are much harder to design and test because one subsystem's behaviors may affect those of other subsystems. These subsystems influence each other not only by explicit communications, but also by implicit environment change as they all share the same environment. For example, in Fig. 1(a),

if subSystem1 changes the environment through its actuators, this change will be noticed by the sensors of subSystem2, thus affecting subSystem2's decision making. An integrated design approach is needed to design this kind of system.

In our approach, a distributed real-time system is modeled as a coupled model that consists of several subcomponents [as shown in Fig. 1(b)]. Each subcomponent corresponds to a subsystem of the distributed real-time system. These subcomponents are coupled together so they can communicate (by coupling a model's output ports to other models' input ports). The couplings among the models correspond to the communication connections among the subsystems in the real world.

To model dynamic reconfiguration of a system, four structure change operations have been developed in the DEVSJAVA environment. These operations are addModel()/removeModel() to add/remove components of a system; and addCoupling()/removeCoupling() to add/remove connections among two components. Based on the situation, a model may call these methods to initiate a structure change. For example, in Fig. 1(b), if a component model, saying Model1, notices that the environment has changed, it can call a structure change method, saying removeCoupling(), to remove the connections between Model2 and Model3. In this case, Model1 acts as a supervisor to monitor the environment change and to transform the system into a new structure that might better suit it to the new environment. More information about variable structure modeling can be found in [5]–[7].

B. Control Model and Sensor/Actuator Activity

For each subsystem, sensors and actuators are modeled as DEVS activity, a concept introduced by RTDEVS for real-time system specification [22]. The control and information processing unit is modeled as a control model that might be an atomic model or a coupled model with multiple subcomponents. In this modeling approach, the control model acts as a brain to process data and to make decisions. Sensor/actuator activity acts as the hardware interface to provide a set of functions for the control model to use. To give an example, consider the design of a mobile robot. A motor activity that drives the robot's motors may be developed. Some typical functions for this motor activity could be move(), stop(), turn(), etc. The definition of an activity and its functions depend on how the designer delineates the "control model–activity" boundary. For example, the designer can model a sensor module that has its own control logic as a sensor activity. Or he can include the logic in the control model and only model the sensor hardware as an activity.

The clear separation of sensor/actuator activity (hardware interface) from the control model makes it possible for a designer to focus on his design interest. In the context of dynamic distributed real-time systems, the control logic is typically very complex, as the systems may operate in a dynamic, uncertain environment. Thus, the control models are the main interest of design and test. The separation from the hardware interface also enables the control models to be disengaged from the hardware, and thus to be modeled in adequate detail in the design stage. Techniques have been developed so that the control models can be tested by simulation methods and then mapped to the real target system for execution without any change and transformation. This capability to preserve the same control models in the transition from simulation-based design to real execution is the most salient feature of this model continuity methodology. Note that because a system's dynamic properties are captured by the control models, maintaining the control models' continuity also implies that the dynamic behavior/structure of the system will be preserved from the design stage to the final execution stage.

C. Environment Model and Virtual Sensor/Actuator

Simulation-based test is used to test and evaluate the control models under development. As shown in Fig. 1(b), simulation-based test is conducted in a virtual test environment. To build this test environment, a model of the real environment is built. This environment model is a reflection of how the real environment affects or is affected by the system under design. Meanwhile, a "simulated" sensor/actuator hardware interface is built for the control models to interact with the environment model. This "virtual sensor/actuator" interface is implemented as abstractActivity. In contrast to an activity, which drives real hardware and executes in real time, an abstractActivity imitates the corresponding activity's behavior and interfaces and is only used during simulation. A sensor abstractActivity gets input from an environment model just as a sensor activity gets input from the real environment. An actuator abstractActivity does similar things as an actuator activity. Note that it is important for an activity and its abstractActivity to have the same interfaces, which are used by the control model in simulation and real execution. By imposing this restriction, the control model can be kept unchanged in the transition from simulation to execution (it interacts with the environment model and real environment using the same interfaces).

Within this test environment, different simulation strategies can be used to test the control models. At the same time, different design alternatives and system configurations can be applied to experiment and exercise the system under design. As will be discussed in Section IV, stepwise simulation methods have been developed so that a model can be simulated and tested incrementally before its real execution. Note that each subcomponent can also be tested and simulated independently.

D. Mapping to a Distributed Environment for Execution

After the models are tested through simulations, they are mapped to the real hardware for execution. During this process, the "virtual sensor/actuator" interface that is used in simulation is replaced by real sensor/actuator interface activity. Specifically, each subsystem goes through "activity mapping" to associate the sensor/actuator activity to the corresponding sensor/actuators hardware. In addition, as the control models are actually executed in a distributed environment, a "model mapping" procedure is applied to deploy the control models to their corresponding host computers. It is important to note that during this process, the designed-in couplings among models are preserved, even though models actually reside on different computers. As such, model continuity in the methodology

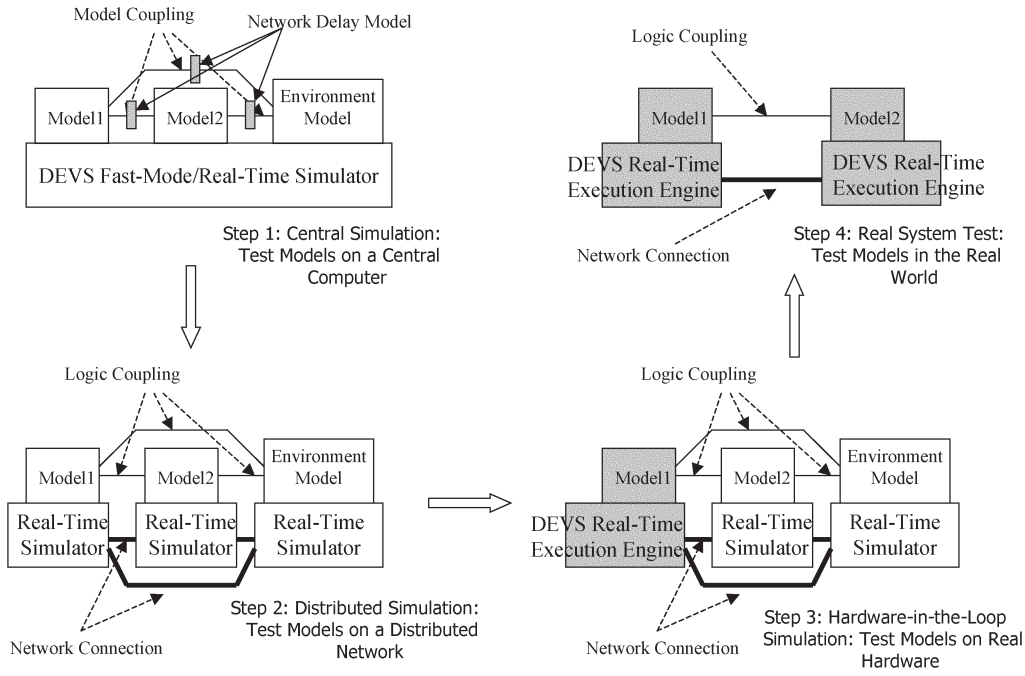


Fig. 2. Stepwise simulation-based test process.

means that not only the control models, but also the couplings among them, are maintained from simulation to distributed execution. By associating the models and activities to their corresponding hardware, the system can be executed in a real environment.

In execution, models are executed within a real environment as shown in Fig. 1(c). The control model of each subsystem makes decisions based on its control logic. It interacts with the real environment through sensor/actuator `activity`. If a model sends out a message, based on the logic coupling, the message will be sent across the network and put to another model's input port. Because these control models have been tested by simulation methods and are invariant in the transition from simulation to execution, they should carry out the control logic just the same way they did when simulated. In practice, one may not be able to model the real environment in adequate detail, and there will be potential for design problems to surface in real execution. When this happens, reiteration through the stages can be more easily achieved with the model continuity approach.

IV. Incremental Simulation and Test

Simulation technologies are being increasingly recognized in industry as a useful means to assess the quality of design choices [23]–[25]. For dynamic distributed real-time systems whose complexity is typically too large to allow an analytical solution, simulation provides an effective way to analyze and test the control models and system configurations before a system's real implementation. In [3], the concept of experimental frame is presented. While experimental frame provides a general guideline to establish simulation-based test environments, this work focuses more on the process to conduct those tests for distributed real-time software systems. Specifically, a stepwise simulation-based test process has been developed so that the control models of a dynamic distributed real-time system can be tested incrementally. This process includes four steps namely central simulation, distributed simulation, hardware-in-the-loop (HIL) simulation, and real system test. Below, Fig. 2 was used as an example (with two subcomponents) to walk through this process.

The first step in the process is central simulation (step 1 of Fig. 2). In central simulation, the two models and the environment model are all in one computer. The control models interact with the environment model through sensor/actuator `abstractActivity`. As will be further discussed in Section V, special couplings between `abstractAcitivity` and the environment model are established to allow them to exchange messages. To model the network latency between the two models that are actually executed on different computers in real execution, network delay models are inserted into the couplings between models. In central simulation, fast-mode simulator and real-time simulator can be chosen to simulate and test the models. As fast-mode simulation runs in logical time (not connected to a wall clock), it generates simulation results as fast as it can. Based on these results, the designers can analyze the data to see if the system under test fulfills the dynamic behavior as desired. In real-time simulation, the simulation speed is synchronized with the wall clock time. This provides designers the flexibility to trace the simulation trajectory in real time. For example, a graphic user interface can be developed to display the state changing of each model in real time.

While in central simulation, network delay models are used to model the network latency between different subsystems, in distributed simulation, the control models are tested on the real network. As shown in step 2 of Fig. 2, in distributed simulation, two models reside on two different computers. The environment model may reside on another computer or on the same computer with one of the models. All of these models are

simulated by real-time distributed simulators. These real-time simulators take care of the underlying network synchronization/coordination and make it transparent to the models. The network delay models are no longer needed because the models are tested in a real network. Note that in step 2, distributed simulation has to run in a real-time fashion. This is because part of the real physical world, the real network, is involved in this simulation-based test.

In central simulation and distributed simulation, the control models are tested on computers. However, these computers often do not represent the real hardware that the models will actually be executed on. It is known that for real-time systems, the executing hardware may have significant impact on how well a model's functions can be carried out. For example, processor speed and memory capacity are two typical factors that can affect the performance of an execution. Thus, to make sure that the control models, having been tested in central and distributed simulation, can also execute correctly and efficiently in the real hardware, hardware-in-the-loop (HIL) simulation [25], [26] is adopted. As shown in step 3 of Fig. 2, in HIL simulation, the environment model is simulated by a DEVS real-time simulator on one computer. One or more control models under test can be deployed to their hardware to be executed by DEVS real-time execution engines. These DEVS real-time execution engines are stripped-down versions of DEVS real-time simulators. They provide compact runtime environments to execute DEVS models [27]. In the example of Fig. 2, `Model1` along with its real-time execution engine reside on the real hardware. `Model2`, environment model, and their real-time simulators reside on other computers. These models still retain the original couplings. A model on the real hardware may interact with the environment model through `abstractActivity`, which acts as virtual sensors or actuators. In the meantime, real sensors or actuators can also be included into HIL simulation by using sensor/actuator `activity` to interact with the real environment. The decision of which sensor/actuator is real hardware and which sensor/actuator is virtual hardware is dependent on the test engineer's test objectives. With different test objectives, different combinations of real sensor/actuators and virtual sensor/actuators can be chosen to conduct an exhaustive test of the control model. For distributed real-time systems, another valuable benefit of HIL simulation is that it allows a subsystem to be tested without waiting for all other subsystems to be completely built. This is because HIL simulation still works within a virtual test environment that can provide virtual subsystems. Thus, in HIL simulation, real and virtual subsystems can work together to conduct a meaningful systemwide test. Section VI gives an example to demonstrate how that can be achieved.

The final step is real system test, where all models are tested on the real hardware within the real environment. As shown in step 4 of Fig. 2, DEVS real-time execution engines execute the models and take care of the underlying network synchronization/coordination. The environment model is no longer needed as the system is tested in the real environment. This is also the same setup as that in real execution where all models interact with the real environment through sensor/actuator activities.

One of the basic rules to conduct these stepwise simulation-based test methods is to put as much as possible of the test in the early steps. This is because the later the step is, the more costly and time consuming it is to set up the test environment. Unfortunately, in reality, many engineers jump directly to step 4 to start their test.

V. Implementation of Features to Support Model Continuity

The proposed methodology is based on the latest version of DEVSJAVA modeling and simulation environment, which is developed from the previous work of Real Time DEVS/CORBA (RTDEVS/CORBA) [28]. Several important features have been taken from the work of RTDEVS/CORBA to build an efficient real-time modeling, simulation, and execution infrastructure. These features include a layered implementation architecture, real-time simulation and execution algorithms, time management techniques, etc. A detailed description of these features can be found in [28]. This section describes two new features that have been implemented: `abstractActivity` and network delay model. These two features are developed specifically to support model continuity, which allows control models to be tested using simulation methods and then easily migrated to execution.

A. *abstractActivity*

The basic idea behind an `abstractActivity` is to provide an abstract sensor/actuator to imitate the behavior and interface of an `activity` that drives real sensors/actuators. In this sense, an `abstractActivity` can be viewed as a virtual sensor/actuator that is used in simulation. As mentioned before, an `abstractActivity` should guarantee that a control model interacts with the environment model in the same way as the control model interacts with the real environment through an `activity`. To support this, `ActivityInterface` is developed to make sure that a DEVS model can treat `abstractActivity` and `activity` in the same way. This `ActivityInterface` defines a set of methods that should be implemented by both `activity` and `abstractActivity`.

```
public interface ActivityInterface{
  public void
    setActivitySimulator(CoupledSimulator sim);
  public String getName();
  public void kill();
  public void start();
  public void returnTheResult(entity myresult);
}
```

A brief description of these methods is given below. For simplicity, the authors use `Activity` to refer to both `activity` and `abstractActivity`

1) Method `setActivitySimulator()`: Set the atomic model's simulator in `Activity`.
2) Method `getName()`: Get the name of an `Activity`.
3) Method `kill()`: Stop an `Activity`.
4) Method `start()`: Start an `Activity`.
5) Method returnTheResult(): Returns result to the DEVS model.

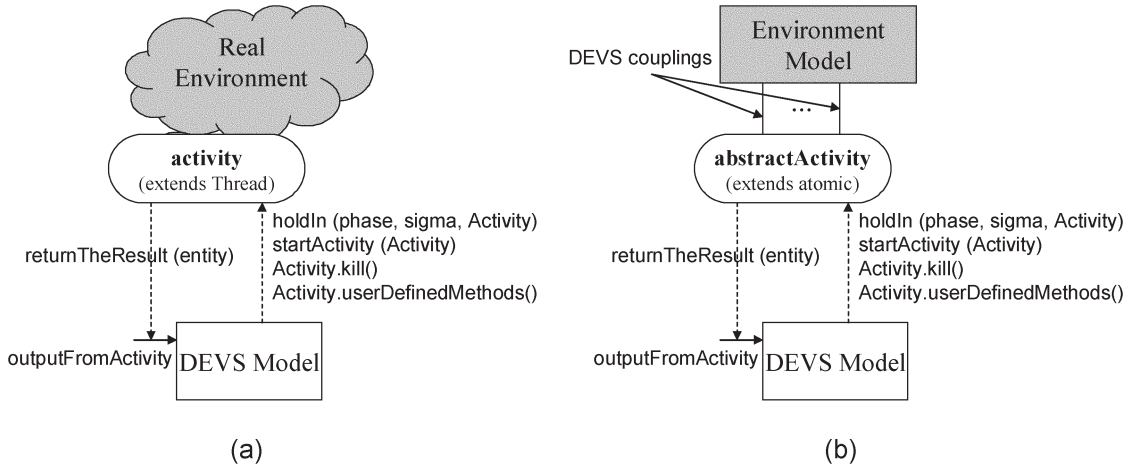


Fig. 3. (a) `activity`. (b) `abstractActivity`.

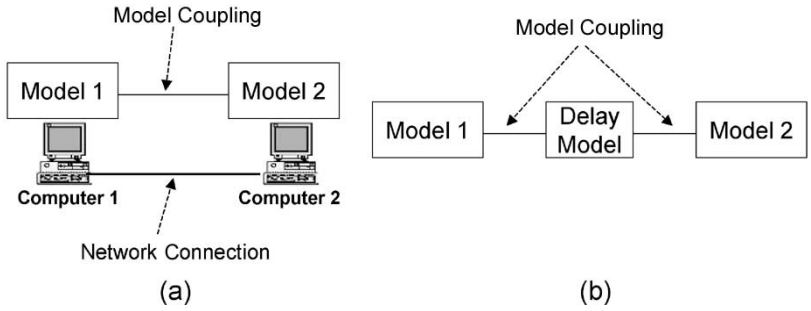


Fig. 4. Network delay model. (a) Distributed execution of a coupled model. (b) Central simulation of a coupled model.

With these methods, the relationship among the control model, `activity`, `abstractActivity`, the environment, and the environment model is shown in Fig. 3.

Fig. 3 shows that both `activity` and `abstractActivity` have the same interfaces with the DEVS control model. Specifically, a DEVS model can start an `Activity` by calling `holdIn()` or `startActivity()`. It can stop an `Activity` by calling the `Activity`'s `kill()` method. An `Activity` can return results to the model by calling the method `returnTheResult()`, which sends the result as a message to the model's reserved input port `outputFromActivity`. This message, as an external event, triggers the model's external transition function `deltext()`, which processes the message and gets the result from `Activity`. Besides these standard-defined methods, Fig. 3 shows that an `activity` can have user-defined methods, such as `move()`, `stop()`, etc. This is because an `activity` is a thread (not a DEVS model) that can have arbitrarily defined methods. These same user-defined methods should also be defined by the corresponding `abstractActivity`.

In the current implementation, both an `abstractActivity` and the environment model are DEVS models. To allow interaction between `abstractActivity` and the `Environment` model, a method `addActivityCoupling()` is developed to add couplings between an `abstractActivity` and the `Environment` model. By calling this function, an `abstractActivity` establishes a "direct" communication channel with the `Environment` model so the message exchange between them does not interfere with the control models, which are the ones that need to be maintained with model continuity. Auxiliary functions `sendInstantOutput()` and `putInstantInput()`were developed to allow an `abstractActivity` to imitate the behavior of the user-defined methods that are defined by `activity`. These functions allow the `abstractActivity` to generate and pass DEVS messages to the `Environment` model.

B. Network Delay Model

The network delay model is developed to simulate the network latency so that a distributed real-time system can be accurately tested in central simulation. In the model continuity methodology, components of a distributed real-time system are modeled as DEVS models and their connections are modeled as DEVS couplings. These couplings are established by method `addCoupling()`. In real execution, the models are mapped to network computers and their couplings remain unchanged. This is shown in Fig. 4(a). However, due to the network latency, the actual time for a message to pass through a coupling (0) may not well represent the latency (> 0) in real execution. This is why network delay models are developed to be used in central simulation. Currently, the delay model uses a fixed deterministic delay, but stochastic delay could easily be added in the future. Furthermore, a new method `addCouplingWithDelay()` is implemented. This method automatically creates and adds a network delay model in the middle of a coupling path as shown in Fig. 4(b). As the `addCouplingWithDelay()` method makes this process transparent to the user, it eases

the transition of models from central simulation to distributed execution—a user only needs to change the code from addCouplingWithDelay() to regular addCoupling().

VI. Dynamic Team Formation Distributed Robotic System

To illustrate the model continuity methodology, this section describes the development of a distributed robotic system, which is considered as a special kind of distributed real-time system where each subsystem is a robot. These systems often exhibit dynamic reconfiguration as robots interact and make decisions in dynamic environments. Specifically, the system described in this section concerns dynamic team formation in which independent mobile robots search for each other, form teams dynamically, and then conduct a leader–follower march. A detailed description of the system and its models can be found in [29]. The following discussion focuses on the development process, especially the stepwise simulation-based test methods, to demonstrate the model continuity methodology.

A. Modeling the Dynamic Team Formation Process

The team formation process starts with both robots moving around and trying to find each other. Initially, there is no connection between the 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 team. Then they begin the leader–follower march with the follower always duplicating the movement of the leader. During movement, the leader makes decisions to go ahead or turn away from an obstacle based on its sensor data. It also passes its motion parameters to the follower. 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. The robots used in this example are car type mobile robots with wireless communication capability. They can move forward/backward and rotate around the center. The robots are equipped with whisker sensors and infrared sensors [30].

Based on the description of this system, three basic components can be recognized: the Manager that resides on a laptop (computer), robot1 and robot2 that reside on mobile robots. Fig. 5 shows the model of this system, where the Manager is an atomic model and each robot is a coupled model. The system has a variable structure because the couplings among its components change dynamically during runtime. Initially, the coupling of the system is established as follows (R1 stands for robot1; R2 stands for robot2 and man stands for Manager; distData, report, check, etc. refer to the port names):

```
addCoupling(R1, "distData", man, "R1Data");
addCoupling(R1, "report", man, "R1Report");
addCoupling(man, "R1Check", R1, "check");
addCoupling(R2, "distData", man, "R2Data");
addCoupling(R2, "report", man, "R2Report");
addCoupling(man, "R2Check", R2, "check");
```

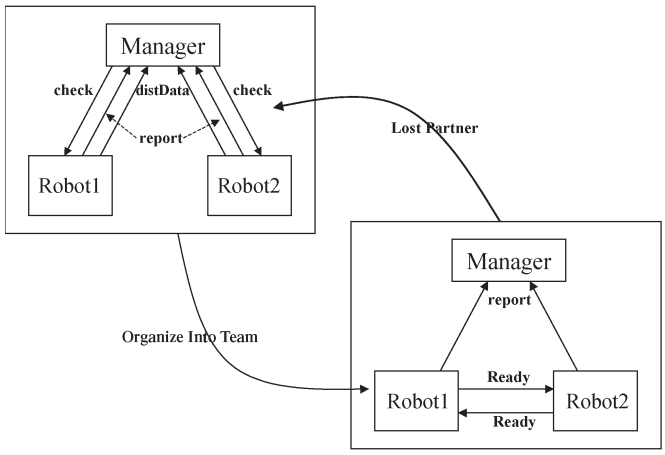


Fig. 5. Dynamic team formation of a two-robot system.

As can be seen initially, there is no coupling between robot1 and robot2. Each robot has output ports distData 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 line-of-sight. The robots return the result of such checking through the report port. When it happens that the report messages returned from the robots are both positive, this indicates that the robots are close enough to 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, the Manager executes the following DEVSJAVA code:

```
removeCoupling(R1, "distData", man, "R1Data");
removeCoupling(man, "R1Check", R1, "check");
removeCoupling(R2, "distData", man, "R2Data");
removeCoupling(man, "R2Check", R2, "check");
addCoupling(R1, "readyOut", R2, "readyIn");
addCoupling(R2, "readyOut", R1, "readyIn");
```

After executing the DEVSJAVA code, bidirectional connections are established by coupling the robots' Ready port to each other. Now, the robots can communicate directly. The distData 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.

The Robot model is based on Brooks' subsumption architecture [31]. It is shown in Fig. 6. There are four components in this model: Avoid, Wander, March, and Monitor. Each of them is responsible for a specific function. For example, the Wander model is responsible to move the robot around without hitting things; the March model organizes robots into a team and then moves the robot in a leader–follower fashion; the Monitor

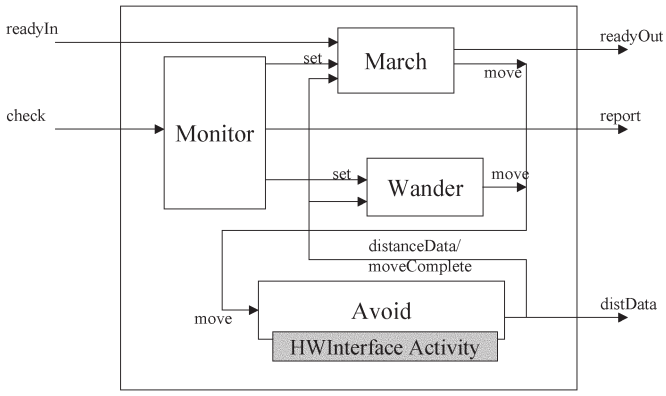


Fig. 6. Robot model.

model checks if two robots really see each other and reports to the Manager; the Avoid model controls the robot to avoid contact with objects based on the whisker sensor data. These models and the couplings among them form the control model of the robot.

The Avoid model also starts the HWActivity and interacts with it. This HWActivity acts as the interface between a robot's control model and sensor/actuator hardware. It drives the motors of the robot and reads data from the robot's whisker sensors and IR sensors. Conceptually, for each of these sensors or actuators, there should be an activity, such as motorActivity, whiskerSensorActivity, or IRSensorActivity, corresponding to it. In this example, the authors combine them into one HWActivity. As will be discussed later, separating these different activities becomes necessary in the robot-in-the-loop simulation, where various combinations of real and virtual sensors/actuators may be employed.

The psuedocode below shows how the Avoid model interacts with HWActivity in its external transition function.

```
public void deltext(double e, message x){
      ... ... ...
  if (messageOnPort(x, "outputFromActivity", i)){
      sensorData
        = x.getValOnPort("outputFromActivity", i);
      ... ... ...
      if (frontTripped)
        HWA.move("backward", speed, dist);
      else if (backTripped)
        HWA.move("forward", speed, dist);
  }
      else if (messageOnPort(x," move", i))
        HWA.move(direction, speed, distance);
      ... ... ...
}
```

In the code, HWA is an instance of HWActivity. As can be seen, in its external transition function deltext(), the Avoid model processes the sensor data (from port outputFromActivity) returned from HWActivity. It calls HWActivity's move() to move backward/forward if its front/back Whisker sensors are tripped. Meanwhile, if there is message on the move port (sent from Wander model or March model), the Avoid model will call HWActivity's move() to move the robot.

To start the HWActivity, the Avoid model initializes an instance of HWActivity and calls the startActivity() method in its initialize() function. This is shown below.

```
HWActivity HWA = new HWActivity();
startActivity(HWA);
```

B. Stepwise Simulation-Based Test

To simulate and test the control models of this system (including the robot models, the Manager model, and their dynamic couplings), an Environment model is developed. The main function of the Environment model is to represent the time duration of robots' movement, and to return sensor data (whisker sensor and infrared sensor) to the control models. As shown in Fig. 7, there are three components in this Environment model: TimeManager1, TimeManager2, and SpaceManager. A TimeManager models a robot's movement times. The SpaceManager models the shape, dimension, and position of robots, obstacles, and the moving field.

Four simulation-based test steps are applied to simulate and test the control models. These steps are central simulation, distributed simulation, robot-in-the-loop simulation, and real system test.

1) Central Simulation: In central simulation, all the models, Robot1, Robot2, and Manager, along with the Environment model reside in a single computer. A virtual hardware interface abstractHWActivity is developed. This abstractHWActivity shares the same interface functions as HWActivity. Thus, to regress from simulation-based test to real execution, the deltext() of Avoid model (as discussed before) does not need to be changed, nor do any other parts of the control logic. The initialize() function of Avoid model is changed to initialize an instance of abstractHWActivity instead of HWActivity. This is shown below:

```
abstractHWActivity HWA
     = new abstractHWActivity ();
startActivity(HWA);
```

In central simulation, network delay models are used to model network latencies. These models are automatically inserted by the addCouplingWithDelay() method of the teamFormation coupled model as shown below. (For demonstration purposes, a deterministic 0.2-s network delay is used. It was noted that in this case study, the network delay does not have direct impact on system's behavior. However, for the systems that have larger network delay or have stricter time constrains, the impact of network delay will be more significant.)

```
public teamFormation(String nm){
  double delay = 0.2; // 0.2 s delay

  Environment env = new Environment("Env");
  add(env); //add the environment model
```

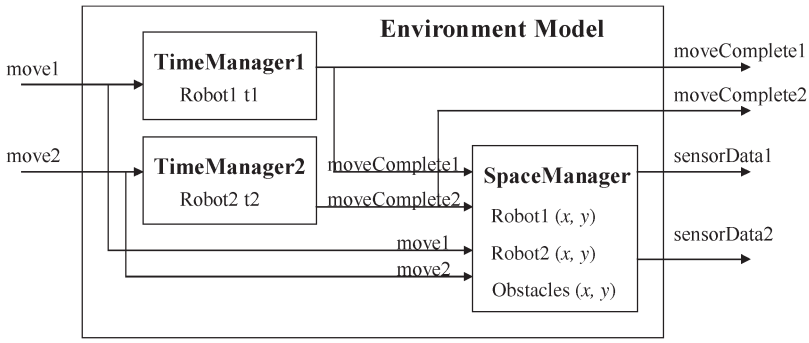


Fig. 7. Environment model.

```
  Robot R1 = new Robot("Robot1");
    add(R1); // add robot1
  Robot R2 = new Robot("Robot2");
    add(R2); // add robot2
  Manager man = newManager("Manager");
  add(man); //add manager

  addCouplingWithDelay
    (R1," distData", man," R1Data", delay);
  addCouplingWithDelay
    (R1," report", man," R1Report", delay);
  addCouplingWithDelay
    (man," R1Check", R1," check", delay);
  addCouplingWithDelay
    (R2," distData", man," R2Data", delay);
  addCouplingWithDelay
    (R2," report", man," R2Report", delay);
  addCouplingWithDelay
    (man," R2Check", R2," check", delay);
}
```

Note that the Environment model was added to the system. This Environment model will interact with abstractHWActivity. Specifically, couplings between them are added by abstractHWActivity using the addActivityCoupling() method. With these couplings, when the Avoid model calls abstractHWActivity's move() method, a message will be passed to the Environment model, which then updates the corresponding robot's position after a period of motion. These couplings also allow sensor data to be passed from the Environment model to the abstractHWActivity.

In central simulation, fast-mode simulator and real-time simulator are applied to simulate and test the control models of the system. Within real-time simulation, a graphic user interface was developed to show how robots move and react to the environment in real time.

2) Distributed Simulation: In distributed simulation, the three components of the system, Robot1, Robot2, and Manager, are distributed on different desktops and laptops across a local area network. Because the system is simulated in a real network, the network delay models are no longer needed. Thus, the addCouplingWithDelay() functions of the teamFormation model are changed back to addCoupling(). With this change, the teamFormation model looks like:

```
  public teamFormation(String nm){
    Environment env = new Environment("Env");
    add(env); //add the environment model

    Robot R1 = new Robot("Robot1");
      add(R1); // add robot1
    Robot R2 = new Robot("Robot2");
      add(R2); // add robot2
    Manager man = new Manager("Manager");
    add(man); //add manager

    addCoupling(R1, "distData", man, "R1Data");
    addCoupling(R1, "report", man, "R1Report");
    addCoupling(man, "R1Check", R1, "check");
    addCoupling(R2, "distData", man, "R2Data");
    addCoupling(R2, "report", man, "R2Report");
    addCoupling(man, "R2Check", R2, "check");
  }
```

Distributed real-time simulators are chosen to simulate and test the system in a distributed environment. In simulation, the two robots still share the same virtual environment as represented by the Environment model. In this way, when model Robot1 moves, model Robot2, which is simulated on a different computer, will notice it.

3) Robot-in-the-Loop Simulation: In HIL—here called robot-in-the-loop simulation—one or both of the models, Robot1 and Robot2, are downloaded to real robot hosts. These control models of these robots are hosted on Dallas Semiconductor's Tiny InterNet Interface (TINI) chips. It was noted that the model code is not altered in this migration since the TINI chip supports the Java-implemented DEVS real-time execution environment [27]. Other models such as the Environment model can reside on other networked computers and are driven by the DEVS distributed real-time simulators. Depending on the desired configuration, the DEVS model resident on real robot may use a robot's real sensor/actuator (implemented by activity) to interact with the real environment or use abstractActivity as virtual sensor/actuators to interact with the Environment model. This capability allows the control

model within the TINI environment to be tested with both simulated and real sensors or actuators.

Below, an experimental setup to see how robot-in-the-loop simulation can be achieved was considered. In this experiment, model Robot1 is downloaded to a real robot robot1 and is executed by a real-time execution engine that runs on the TINI chip. Model Robot2, Manager, and the Environment are simulated on computers; among them, Robot2 uses abstractHWActivity to interact with the Environment model. The authors configure Robot1 to use its real motors to move within a real physical world, and to use virtual whisker sensors and IR sensors to get sensor data from the Environment model. To serve this purpose, motorActivity and sensorAbstractActivity are developed. The motorActivity defines the move() method that drives the real robot's motors; the sensorAbstractActivity combines the functions of virtual whisker sensors and virtual IR sensors. It gets the virtual sensor data from the Environment model and calls returnTheResult() to send the data to the Avoid model. To support model continuity, the motorActivity and sensorAbstractActivity together should have the same interface functions as those of the abstractHWActivity that is used in the previous steps.

After defining the motorActivity and sensorAbstract Activity, the Avoid model can use them. Specifically, in its initialize() function, the Avoid model initializes and starts the motorActivity and sensorAbstractActivity. This is shown below.

```
motorActivity motorA = new motorActivity();
startActivity(motorA);
sensorAbstractActivity sensorA
  = new sensorAbstractActivity();
startActivity(sensorA);
```

Then, the motorActivity and sensorAbstractActivity are used by the Avoid model's external transition function deltext() as shown below.

```
public void deltext(double e, message x){
    ... ... ...
if (messageOnPort(x,″ outputFromActivity″, i)){
  sensorData
    = x.getValOnPort(″outputFromActivity″, i);
  ... ... ...
  if (frontTripped)
    motorA.move(″backward″, speed, dist);
  else if (backTripped)
    motorA.move(″forward″, speed, dist);
}else if (messageOnPort(x,″ move″, i))
    motorA.move(direction, speed, distance);
      ... ... ...
}
```

As can be seen, the Avoid model processes the sensor data returned from virtual sensors sensorA. It calls motorA's move() to move the real robot backward/forward if the (virtual) whiskers sensors are tripped. Meanwhile, if there is a message on the move port (sent from Wander model or March model), the Avoid model calls motorA's move() to move the real robot. It was noted that in the robot-in-the-loop simulation, a virtual counterpart of the real robot is needed in the virtual environment (the Environment model). Thus, when motorA gets the move command from the Avoid model, a corresponding message is sent to the Environment model. A standard way to realize this is under development so that the movement of the real robot and its agent in the virtual environment can be effectively synchronized.

This experimental setup shows that the Avoid model uses its virtual sensor interface sensorA to get sensor data from the virtual environment and uses its real motor interface motorA to move the robot. As a result, the real robot robot1 moves in a physical field based on the sensor data from a virtual environment. Within this virtual environment, robot1 can "see" virtual obstacles and other robots, such as Robot2, which are simulated on computers.

4) Real System Test: In real system test, all the models are deployed to their real hardware and tested in a real physical environment. In this example, the Robot1 and Robot2 models are downloaded to the TINI chips on the respective real robots while the Manager model is downloaded to a wireless laptop. All the models are executed by real-time execution engines. For both robots, virtual sensor/actuator interfaces (abstractHWActivity) are replaced by real sensor/actuator interfaces (HWActivity). The Environment model is eliminated since the robots now operate in the real world. With these changes, the teamFormation model is shown below.

```
public teamFormation (String nm){
  Robot R1 = new Robot(″Robot1″);
    add(R1); // add robot1
  Robot R2 = new Robot(″Robot2″);
    add(R2); // add robot2
  Manager man = new Manager(″Manager″);
  add(man); //add manager

  addCoupling(R1, ″distData″, man, ″R1Data″);
  addCoupling(R1, ″report″, man, ″R1Report″);
  addCoupling(man, ″R1Check″, R1, ″check″);
  addCoupling(R2, ″distData″, man, ″R2Data″);
  addCoupling(R2, ″report″, man, ″R2Report″);
  addCoupling(man, ″R2Check″, R2, ″check″);
}
```

Note that this is also the same setup as in the final execution, where robots move within a physical field and respond to a real environment.

C. Results and Discussion

This example shows how a "dynamic team formation" system can be modeled and then tested by simulation methods. One of the results for this example is to check if robots exhibit the same (similar) behavior in real execution as that in simulation-based test. For that purpose, a movie [32] has been recorded to show the team formation process in simulation as

well as in real execution. This movie clearly demonstrates the continuity between these two stages. It was noted that although the above example does not involve a complex environment setting, it is expected that this "continuity" will be preserved even in complex environments.

VII. Conclusion

In this paper, a software development methodology for dynamic distributed real-time systems was presented. The methodology is based on DEVSJAVA modeling and simulation environment. It supports model continuity so that a dynamic distributed real-time system can be designed, analyzed and tested by simulation methods, and then migrated to be executed in a distributed network while preserving its control models. To handle the dynamic properties of a distributed real-time system, the variable structure modeling capability is integrated into the proposed methodology. Stepwise simulation methods such as central simulation, distributed simulation, and hardware-in-the-loop (HIL) simulation are developed to incrementally test the control models in a virtual environment. A distributed robotic "team formation" example was developed and presented in the paper to demonstrate how this dynamic system can be developed by applying the proposed methodology in different stages.

References

[1] E. Gery, D. Harel, and E. Palachi, "Rhapsody, a complete life-cycle model-based development system," in *Proc. 3rd Int. Conf. Integrated Formal Methods (IFM)*, Turku, Finland, 2002, pp. 1–10.

[2] W. Schultein, "Why doesn't anyone use formal methods?" in *Proc. 2nd Int. Conf. Integrated Formal Methods (IFM)*, Dagstuhl Castle, Germany, 2000, pp. 297–298.

[3] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd ed. New York: Academic, 2000.

[4] Artificial Intelligence and Simulation Research Group, DEVS-Java Reference Guide. Tucson, AZ: Univ. Arizona, 1997. [Online]. Available: www.acims.arizona.edu

[5] F. J. Barros, "Modeling formalisms for dynamic structure systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 4, pp. 501–515, 1997.

[6] A. M. Uhrmacher, "Dynamic structures in modeling and simulation—A reflective approach," *ACM Trans. Model. Simul.*, vol. 11, no. 2, pp. 206–232, Apr. 2001.

[7] X. Hu, B. P. Zeigler, and S. Mittal, "Variable structure in DEVS component-based modeling and simulation," *Simulation: Trans. Soc. Model. Simul. Int.*, vol. 81, no. 2, pp. 91–102, 2005.

[8] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 58–93, Jan. 2001.

[9] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-code traceability for object-oriented systems," *Ann. Softw. Eng.*, vol. 9, no. 1–4, pp. 35–58, 2000.

[10] J. L. Boyd, *Designing Reactive Systems for Strong Traceability*. Ottawa, ON, Canada: Carleton Univ., 1993.

[11] R. S. Janka, L. M. Wills, and L. B. Baumstark, "Virtual benchmarking and model continuity in prototyping embedded multiprocessor signal processing systems," *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, pp. 832–846, Sep. 2002.

[12] R. L. Bagrodia and C. Shen, "MIDAS: Integrated design and simulation of distributed systems," *IEEE Trans. Softw. Eng.*, vol. 17, no. 10, pp. 1042–1058, Oct. 1991.

[13] F. G. Gonzalez and W. J. Davis, "A new simulation tool for the modeling and control of distributed systems," *Simulation: Trans. Soc. Model. Simul. Int.*, vol. 78, no. 9, pp. 552–567, 2002.

[14] B. Selic, "The emerging real-time standard [UML]," in *Proc. 6th Int. Workshop Object-Oriented Real-Time Dependable Systems*, Rome, Italy, 2001, pp. 3–9.

[15] K. H. Kim, "APIs for real-time distributed object programming," *IEEE Comput.*, vol. 33, no. 6, pp. 72–80, Jun. 2000.

[16] H. Gomaa, *Software Design Methods for Concurrent and Real-Time Systems*. Boston, MA: Addison-Wesley Longman, 1993.

[17] ——, *Designing Concurrent, Distributed, and Real-Time Applications With UML*. Boston, MA: Addison-Wesley Longman, 2000.

[18] E. A. Lee, Overview of the Ptolemy Project Technical Mem. Berkeley, CA: Univ. California, Mar. 6 2001. [Online]. Available: http://ptolemy.eecs.berkeley.edu/

[19] Y. K. Cho, X. Hu, and B. P. Zeigler, "The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems," *Simulation: Trans. Soc. Model. Simul. Int.*, vol. 79, no. 4, pp. 197–210, 2003.

[20] X. Hu and B. P. Zeigler, "An integrated modeling and simulation methodology for intelligent systems design and testing," in *Performance Metrics Intelligent Systems Workshop*, Gaithersburg, MD, 2002.

[21] S. C. Shaw, *Real-Time Systems and Software*. New York: Wiley, 2001.

[22] J. S. Hong and T. G. Kim, "Real-time discrete event system specification formalism for seamless real-time software development," *Discret. Event Dyn. Syst.: Theory Appl.*, vol. 7, no. 4, pp. 355–375, 1997.

[23] R. Sinha, V. C. Liang, C. J. J. Paredis, and P. K. Khosla, "Modeling and simulation methods for design of engineering systems," *J. Comput. Inf. Sci. Eng.*, vol. 1, no. 1, pp. 84–91, 2001.

[24] S. Schulz, K. J. Buchenrieder, and J. W. Rozenblit, "Multilevel testing for design verification of embedded systems," *IEEE Des. Test Comput.*, vol. 19, no. 2, pp. 60–69, Mar.–Apr. 2002.

[25] M. Gomez. (2001, Dec.) Hardware-in-the-loop simulation. *Embedded Syst. Program*. [Online]. *14(13)*. Available: http://www.embedded.com/story/OEG20011129S0054

[26] R. B. Wells, J. Fisher, Y. Zhou, B. K. Johnson, and M. Kyte, "Hardware and software considerations for implementing hardware-in-the-loop traffic simulation," in *27th Annu. Conf. IEEE Industrial Electronics Society (IECON)*, Denver, CO, 2001, vol. 3, pp. 1915–1919.

[27] X. Hu, B. P. Zeigler, and J. Couretas, "DEVS-on-a-chip: Implementing DEVS in real-time Java on a tiny internet interface for scalable factory automation," in *IEEE Int. Conf. Systems, Man, and Cybernetics*, Tucson, AZ, Oct. 2001, pp. 3051–3056.

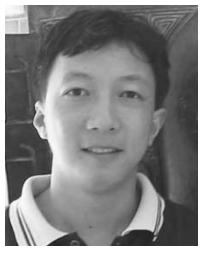
[28] Y. K. Cho, "RTDEVS/CORBA: A distributed object computing environment for simulation-based design of real-time discrete event systems," Ph.D. dissertation, Univ. Arizona, Tucson, AZ, 2001.

[29] X. Hu and B. P. Zeigler, "Model continuity to support software development for distributed robotic systems: A team formation example," *J. Intell. Robot. Syst., Theory Appl., Special Issue: Multiple and Distributed Cooperating Robots*, vol. 39, no. 1, pp. 71–87, Jan. 2004.

[30] J. Peipelman, N. Alvarez, K. Galinet, and R. Olmos. (2001, May). TINI I: Artificial intelligence robots. Dept. Elect. Comput. Eng., Univ. Arizona. [Online]. Available: http://www.cs.gsu.edu/~cscxlh/TechnicalReport.doc

[31] R. A. Brooks, "A robust layered control system for a mobile robot," IEEE J. Robot. Autom., RA-2, vol. 2, no. 1, pp. 14–23, Apr. 1986.

[32] X. Hu. (2002, Oct.). Multi-robot system. [Online]. Available: http://www.acims.arizona.edu/PROJECTS/MultiRobot.mpg



Xiaolin Hu (S'02–M'04) received the Ph.D. degree in electrical and computer engineering from the University of Arizona, Tucson, in 2004.

He is an Assistant Professor of the Computer Science Department at Georgia State University, Atlanta. His research interests are in system modeling and simulation, software engineering, and their applications to distributed real-time systems and distributed robotic systems.

Bernard P. Zeigler (M'87–SM'87–F'94) is Professor of Electrical and Computer Engineering at the University of Arizona, Tucson, and Director of the Arizona Center for Integrative Modeling and Simulation. He is internationally known for his 1976 foundational text Theory of Modeling and Simulation, recently revised for a second edition (Academic Press, 2000). He has published numerous books and research publications on the Discrete Event System Specification (DEVS) formalism.

Prof. Zeigler was named Fellow of the IEEE in recognition of his contributions to the theory of discrete event simulation in 1995. In 2000, he received the McLeod Founder's Award by the Society for Computer Simulation, its highest recognition, for his contributions to discrete event simulation. In June 2002, he was elected President of the Society (recently renamed The Society for Modeling and Simulation, International). In 2003, his autobiographical retrospective on the evolution of the theory of modeling and simulation appeared in the *International Journal of General Systems* [Vol. 32 (3)].