The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems

Young Kwan Cho, Xiaolin Hu and Bernard P. Zeigler

SIMULATION 2003 79: 197
DOI: 10.1177/0037549703038880

The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems

Young Kwan Cho
Republic of Korea Air Force
Nonsan City
Choongnam, Korea 320-913

Xiaolin Hu
Bernard P. Zeigler
Arizona Center for Integrative Modeling and Simulation
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721
xiaolin@email.arizona.edu

The increasing complexity of large-scale distributed real-time systems demands powerful real-time object computing technologies. Furthermore, systematic design approaches are needed to support analysis, design, test, and implementation of these systems. The authors discuss RTDEVS/CORBA, an implementation of discrete event system specification (DEVS) modeling and simulation theory based on real-time CORBA communication middleware. With RTDEVS/CORBA, the software model of a complex distributed real-time system can be designed and then tested in a virtual testing environment and finally executed in a real distributed environment. This model continuity and simulation-based design approach effectively manages software complexity and consistency problems for complex systems, increases the flexibility for test configurations, and reduces the time and cost for testing. The layered architecture and different implementation issues of RTDEVS/CORBA are studied and discussed. An example application is then given to show how RTDEVS/CORBA supports a framework for model continuity in distributed real-time modeling and simulation.

Keywords: Distributed real-time systems, modeling and simulation, DEVS, CORBA, model continuity, simulation-based design, virtual testing environment

1. Introduction

Distributed real-time object computing technologies have been attracting lots of attention in the real-time computing area during recent years. Because of the effective object-oriented methodologies that enable engineers to reduce the development complexity and maintenance costs of large-scale software applications, object-oriented computing technology has been successfully applied to non-real-time software systems. However, real-time system engineering techniques have not fully adopted the concept of modular design and analysis, which are the main virtues of object-oriented design technologies. As a consequence, the demand for object-oriented analysis, design, and implementation of large-scale real-time applications has been growing.

This paper proposes that a discrete event system specification (DEVS)–based real-time modeling and simulation environment can provide capabilities required by real-time system engineering. DEVS is a sound formal modeling and simulation (M&S) framework based on generic dynamic systems concepts [1]. DEVS is a mathematical formalism with well-defined concepts of hierarchical and modular model construction, coupling of components, support for discrete event approximation of continuous systems, and an object-oriented substrate supporting repository reuse. DEVS is not, however, just a mathematical framework but also a practical M&S tool implemented in various object-oriented languages such as Scheme, C++, and Java. Recently, DEVS modeling and simulation environments also

SIMULATION, Vol. 79, Issue 4, April 2003 197–210
©2003 The Society for Modeling and Simulation International
DOI: 10-1177/0037549703038880

have been combined with middlewares such as High-Level Architecture (HLA) runtime infrastructure (RI) and Common Object Request Broker Architecture (CORBA) to support fast and easy construction of distributed models, as well as simulation of such models. These DEVS-based modeling and simulation environments have been shown to support an effective modeling and simulation methodology in various application areas, including design and implementation of real-time control systems [2].

Real-time systems design connotes an approach to software design in which *timeliness* (or *timing correctness*) is as important as the correctness of the outputs (or *logical correctness*) [3, 4]. Timeliness of response does not necessarily imply speed—although, this may be important—as much as predictability of response and reliable conformance to deadlines. For real-time systems, performance estimation to guarantee that the system under design meets performance requirements is crucial. Performance analysis often concerns schedulability, which involves checking the task schedule for feasibility or conformance with the required timing constraints. In distributed networked systems, quality of service (QoS) characteristics, such as the timely delivery of events between system components or priority-based bandwidth utilization, must necessarily be included in performance evaluation. To support the design and performance evaluation for a distributed real-time system, modeling and simulation technologies are developed.

Real-time considerations enter into the world of modeling and simulation in various ways. A real-time simulation is a real-time system in which some portion of the environment, or portions of the real-time system itself, is realized by simulation models [5]. When a simulation model interacts with a surrounding environment, such as software modules, hardware components, or human operators, the simulator must handle external events from its environment in a timely manner [6]. In more general terms, interfacing of abstract models with real-world processes requires that the (logical) time base of the simulation be synchronized as closely as possible to the clock time of the underlying computer system [2]. Work related to real-time simulation and control includes early research in DEVS-Scheme [2], the extension of the DEVS formalism to the DEVS real-time formalism [6], as well as its application to process control [7]. Current projects include the following: PORTS: A Parallel, Optimistic, Real-Time Simulation [5]; Operators Training Distributed Real-Time Simulation (OPERA) [8]; Ptolemy (Concurrent Discrete Event Simulation) [9, 10]; Time-Triggered, Message-Triggered Object (TMO)–Based Distributed Real-Time System Development Environment [11, 12]; and Cluster Simulation—Support for Distributed Development of Hard Real-Time Systems Using TDMA-Based Communication [13].

One concern of using modeling and simulation technology to support the system design of distributed real-time systems is how to maintain a model's continuity during the system's development process. This continuity means that the same model can be reused with minimal change during different design stages—from model design to performance evaluation and even to final execution. To support model continuity, one must develop techniques so that the modeling and simulation framework can treat the development of distributed real-time systems in a systematic way.

In this paper, we present how a distributed modeling and simulation framework, the real-time DEVS/CORBA (RTDEVS/CORBA), is developed to support the design of distributed real-time systems. To set the stage, we will review the concept of model continuity and show how RTDEVS/CORBA supports model continuity when traversing through different design stages. We then discuss different layers of the RTDEVS/CORBA modeling and simulation environment. With this background, we proceed to discuss in detail the implementation issues of RTDEVS/CORBA. Finally, an example is presented showing how a complex distributed real-time system can be developed using the framework of RTDEVS/CORBA and how model continuity is supported during the development process.

2. Support Model Continuity with RTDEVS/CORBA

Model continuity refers to the ability to use the same model of a system throughout its design phases. For the RTDEVS/CORBA environment, model continuity (or model transferability) means that models developed in any DEVS-based M&S environments, such as DEVS-Java, DEVS/HLA, or DEVS/CORBA, should be able to run in the RTDEVS/CORBA environment with minimal changes as long as they maintain the same interfaces [14]. With model continuity, distributed real-time systems can be designed, analyzed, and tested through DEVS-based modeling and simulation studies and then migrated with minimal additional effort to be executed in the distributed network. In other words, this framework provides generic supports to develop models of distributed real-time systems, evaluate their performance and timing behavior through DEVS-based modeling and simulation, and ease the transition from the simulation to actual executions.

Real-time systems deal with external stimuli from outside of the systems with time constraints. Therefore, real-time systems usually interact with environmental systems that could be hardware, software, human operators, and so forth that want to get timely responses from the real-time systems. When a real- time system is developed, the best way to evaluate and test the system under design might be for the system to interact directly with the existing environment. However, this is usually limited by cost and some other limitations such as safety and availability. Hence, it is good to provide a real-time modeling and simulation environment in which the environment model can be developed so that the system under design can be simulated

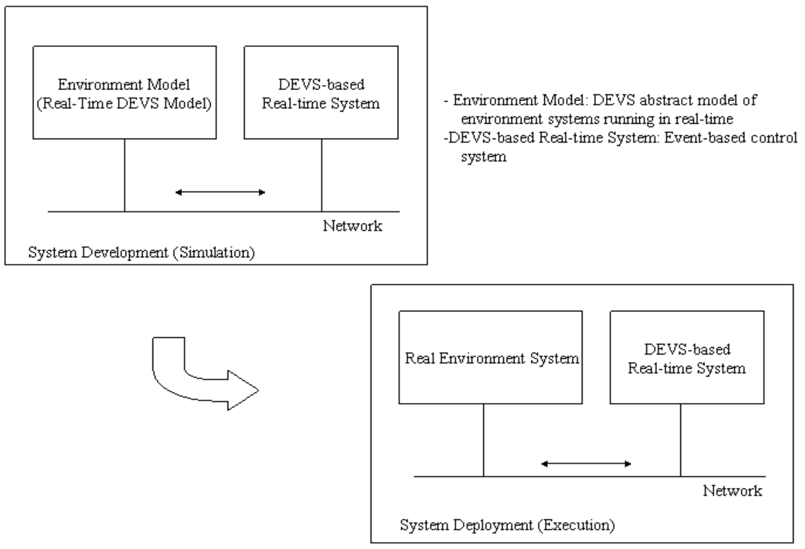
Environment Model (Real-Time DEVS Model)
DEVS-based Real-time System
Network
System Development (Simulation)
- Environment Model: DEVS abstract model of environment systems running in real-time
-DEVS-based Real-time System: Event-based control system
Real Environment System
DEVS-based Real-time System
Network
System Deployment (Execution)

Figure 1. Real-time simulation and execution

along with the environment model. Once the system has been tested by the simulation, it should be easily deployed in the execution environment, in which the interfaces to the system are the same as those between the environment model and the system.

Figure 1 illustrates a conceptual overview of the relationship between real-time simulation and real-time execution in the RTDEVS/CORBA environment. High-level abstract models that provide logical and timing behaviors like the real environment systems can be developed and simulated in this environment so that real-time systems can be developed based on the interactions with the environment models. After finishing the development, the real-time systems could be easily migrated into the execution environment with minimal modifications.

Figure 2 shows in detail how this process works. In the modeling stage, the modeler defines DEVS models that capture the logic behavior and temporal behavior of the system under design. Then these models will be simulated in the DEVS-Java environment, which run a logic simulation and thus can check the models' logical behavior. After the models are checked by DEVS-Java, these same models can be distributed on different computers and simulated by DEVS/CORBA, which is a distributed logical simulation environment. With DEVS/CORBA, the model under design's logical behavior can be further checked in the distributed environment. Although DEVS/CORBA can check the models' logical correctness in the distributed environment, it cannot check the models' temporal behavior. So the next stage of the development is simulating the models in RTDEVS/CORBA, a real-time distributed simulation environment. Because RTDEVS/CORBA provides time-sensitive and QoS-supported distributed real-time simulations, the models' temporal behavior can be checked in this environment. Furthermore, after the models under design pass all these design and test stages, the same models can also be executed by the RTDEVS/CORBA real-time execution engine in a distributed environment.

As can be seen, during the whole process, the model's continuity is maintained, as the same model goes through different development stages—from modeling to testing and to the final execution. This model continuity is supported in the DEVS methodology by choosing different simulation and execution environments during different development stages while keeping the same model unchanged.

3. A Layered Modeling and Simulation Framework

To address design issues for real-time systems in the distributed computing environment, we adopted a layered architecture to support software design in which there is a separation of concerns underlying each layer. Figure 3 shows these layers of the RTDEVS/CORBA modeling and simulation environment.

The heterogeneous network, or the lowest layer in Figure 3, represents the actual physical hardware, computers, cables, routers, and so forth.

The second, or the real-time middleware layer, refers to newly emerging software that provides services to mediate the communication among nodes in the network with some degree of assured performance levels. In the RTDEVS/CORBA implementation, the real-time middleware is Adaptive Communication Environment/The ACE

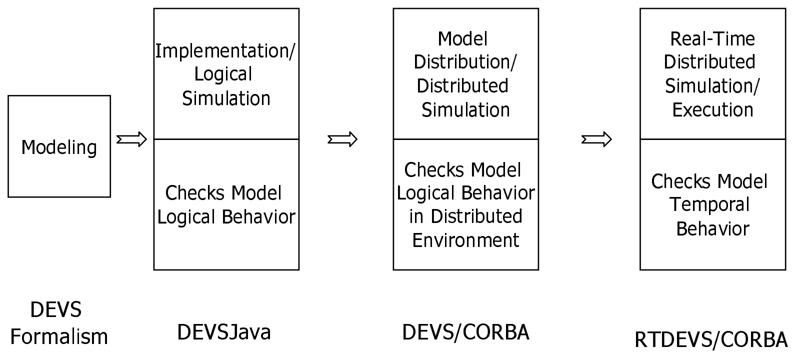
Modeling
Implementation/ Logical Simulation
Checks Model Logical Behavior
Model Distribution/ Distributed Simulation
Checks Model Logical Behavior in Distributed Environment
Real-Time Distributed Simulation/ Execution
Checks Model Temporal Behavior
DEVS Formalism
DEVSJava
DEVS/CORBA
RTDEVS/CORBA

Figure 2. Model continuity during different development stages

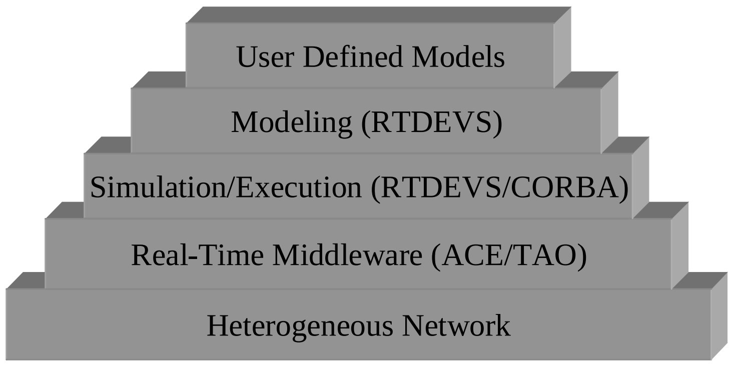
User Defined Models
Modeling (RTDEVS)
Simulation/Execution (RTDEVS/CORBA)
Real-Time Middleware (ACE/TAO)
Heterogeneous Network

Figure 3. Layers of the real-time discrete event system specification (RTDEVS)/Common Object Request Broker Architecture (CORBA) environment

ORB (ACE/TAO), which is an extension of CORBA developed to demonstrate the feasibility of using CORBA for real-time applications [15]. Because conventional CORBA lacks real-time programming features and QoS support, it is not suited for high-performance, real-time applications. These shortcomings can be overcome by adopting the *ACE ORB* (TAO)—a real-time CORBA-compliant Object Request Broker (ORB) end system developed based on the ACE framework, which is a highly portable object-oriented middleware communication framework [16]. Since TAO is an ACE-based ORB, it runs on a wide range of operating system (OS) platforms and is compliant with most of the features and components in the CORBA 2.4 specification. In addition, TAO also provides real-time CORBA services such as the real-time event service and the real-time scheduling service, which are used to support the RT-DEVS/CORBA environment.

On top of the real-time middleware layer reside the DEVS-based modeling layer (RTDEVS) and the simulation/execution layer (RTDEVS/CORBA). As the RT-DEVS modeling layer provides a modeling environment in which real-time control models can be specified and implemented, the RTDEVS/CORBA layer provides the simulation/execution capability in which DEVS models can be simulated and executed in real time. Finally, based on specific applications, a user can develop his or her models and use the services of the underlying layers to simulate, test, and execute the models under design.

Note that without this separation of concerns, one would have much greater difficulty in trying to formulate and disentangle the control and network response issues just mentioned. Moreover, the layering makes it possible to reuse the software developed at the layers or, indeed, to purchase commercial off-the-shelf (COTS) software if that is available.

4. Implementation of Distributed RTDEVS/CORBA

Implementation of distributed RTDEVS/CORBA is based on the previous framework for the DEVS/CORBA distributed modeling and simulation environment [17]. DEVS/CORBA is a logical simulation environment, whereas RTDEVS/CORBA is a real-time simulation environment. In the logical DEVS simulation environment, generation and delivery of events occur at the same logical time. In the distributed real-time simulation environment, however, this is not the case. There exist some time differences between event schedule, event generation, and event delivery in a real-time context. In a distributed simulation environment, time differences also exist between different distributed computers. Thus, the main concerns about the design and implementation of the real-time simulator in RTDEVS/CORBA are how to handle time in a distributed environment, how to synchronize simulation time with the physical time, and how to manage time differences in generating events at the scheduled time that is defined in the model and delivering them to the destination models in valid time boundaries, which requires QoS capabilities. In this section, these design and implementation issues are discussed.

4.1 Real-Time Simulator

In the distributed real-time DEVS simulation/execution context, running models in real time means that a model must be able to generate an “event” at the scheduled time and change its state based on the calculation caused by events. Furthermore, events generated by a model should be delivered to a remote model within some valid time boundaries specified as QoS by a modeler. The design and implementation of a real-time simulator focus on this fact.

The RTDEVS formalism defines two kinds of events—internal and external— which are handled by the internal transition function (δint) and the external transition function (δext), respectively. An internal event is an event generated by a model itself according to the time schedule specified in the model, whereas an external event is an event delivered from other models or from the external environment.

In Figure 4, the real-time simulator is illustrated as a model that shows the state transition flow of the simulator. The upper box in the figure represents the real-time simulator, and the other box is a DEVS model. The real-time simulator basically behaves almost the same as the previously implemented simulators, except that the time base is synchronized with physical time. When a simulation cycle begins, the simulator initializes and goes into the *nextTN* phase, in which the simulator calculates a time of next event (*tN*) based on the schedule specified in the DEVS model. Once the simulator gets the *tN*, it determines an amount of wait time by subtracting the current time from the *tN* and then goes into the *waitFor* phase, when the real-time simulator performs a *synchronization* step. In this phase, the simulator waits for either the *tN* (an internal event) or an external event. Either event triggers the real-time simulator to move out of the *waitFor* phase. Getting out of the *waitFor* phase is handled by two different simulator algorithms, as shown in Figure 5.

Figure 5a shows the internal transition-handling algorithm, whereas Figure 5b shows the external transition-handling algorithm. At the beginning of each algorithm, the simulator moves into the confluent phase in which the simulator is supposed to handle confluent problems caused by clock and network delay jitter. To compensate for jitter, the real-time simulator waits for a certain amount of time before it goes into the next step. This is explained in detail in the next section. After the confluent phase, the simulator goes through the same simulation steps such as *computeInputOutput*, *wrapDelFunc*, and *nextTN* as in the simulator of DEVS/CORBA.

The real-time simulator iterates one cycle of phases from *waitFor* to *nextTN*, while the simulator keeps interacting with the DEVS model whenever either an internal or external event arrives at the real-time simulator. In an ideal case, this iteration takes a zero time unit, as in the logical simulation. In the real world, however, each step of the iteration takes nonzero time units, which introduces the time discrepancy problem between the time of the event and the actual transition in the model. One aspect of this problem is delay accumulation. For example, suppose that a real-time simulator has reached time $t1$ and gets an internal event. So it needs to execute the internal transition-handling algorithm. Since executing this algorithm takes nonzero time ε1, the actual internal transition of the model is carried out at time $t1$ + ε1. Therefore, if the model calculates its new *tN* from this moment, then *tN* would be $t1$ + ε1 + *ta*, which is not $t1$ + *ta*, the ideal time according to the original schedule. To make sure *tN* is scheduled at $t1$ + *ta*, the simulator should calculate the delay ε1 and compensate this time of next event for the delay. Without delay compensation, accumulation of delay would continue to build up during the simulation cycle until the simulator misses the event schedule. A similar situation exists when an external event comes in and the external transition function needs to be called.

To compensate for the erroneous delays in the simulator, we record the time $t1$ when an internal or external event happens and set the last event time (tL) as $t1$. Then, we execute the event transition-handling algorithm and, when it is finished, schedule $tN = tL + ta$. This way, even the actual event transition happens at time $t1$ + ε1, and tN is set to $t1$ + *ta* instead of $t1$ + ε1 + *ta*. This is because tN is calculated from tL, which is independent of ε1, the time it takes to process the event transition-handling algorithm.

4.2 Time Synchronization and the Confluent Problem

4.2.1 Time Synchronization

In logical time DEVS simulation, the coordinator maintains the logical DEVS time and strictly controls the entire simulation cycle. All the participating simulators are closely synchronized with the logical DEVS time under the control of the coordinator. The same is true for any

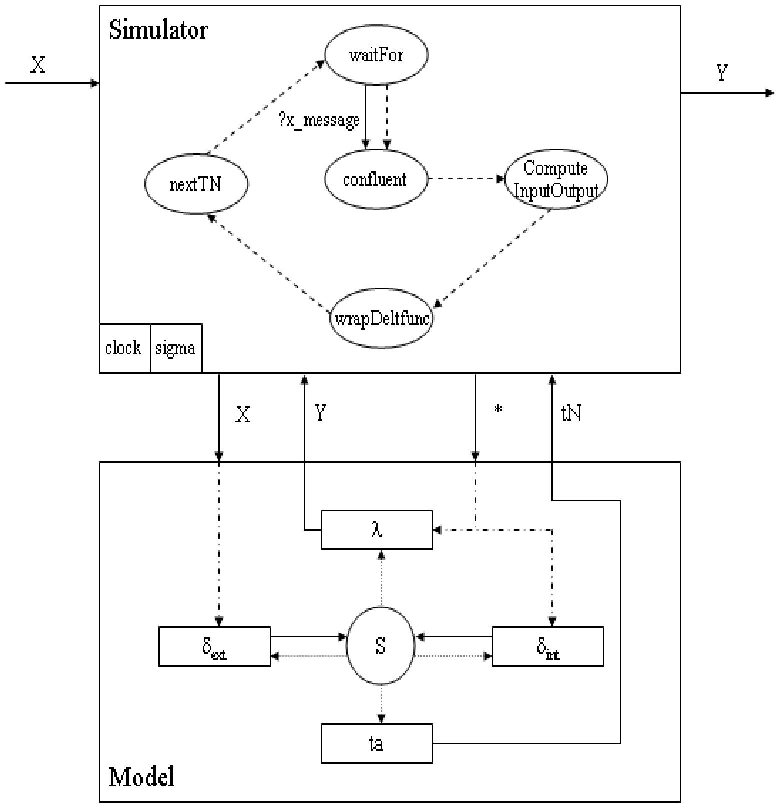

Figure 4. Real-time simulator and its model

```
while(true)
        wait for $\sigma$;
        t := checkCurrentTime();
        checkConfluent();
        y := $\lambda$(s);
        s' := $\delta_{int}$(s);
        tL := t;
        tN := tL + ta(s');
end
```

(a)

```
when receive an external event
        t := checkCurrentTime();
        checkConfluent();
        e := t – tL;
        s' := $\delta_{ext}$(s,e,x);
        tL := t;
        tN := tL + ta(s');
end
```

(b)

Figure 5. Simulator algorithm

conservative parallel DEVS implementation in the distributed computing environment (i.e., DEVS/HLA and DEVS/CORBA), which performs logical simulation. Although this central control approach manages the time by using one central coordinator, it brings problems such as response latency and central bottleneck. To avoid this disadvantage, in the RTDEVS/CORBA implementation, each computer has its own simulator/executor that simulates/executes the model of that computer autonomously. This autonomous timer approach supports fast response and scalability, but it brings the time synchronization problem in which time consistency has to be maintained among different simulators.

For distributed real-time simulation or execution, synchronization usually includes two aspects: one is how to keep clocks synchronized with each other, which is referred to as *internal synchronization*; another is how to keep them synchronized with the world standard time, which is referred to as *external synchronization*. External synchronization provides distributed systems with the use of a synchronized time base on a global scale, and it enables the systems synchronized with its surrounding environment.

Two ways for time synchronization have been studied by Cho [14]. The first one is “soft” synchronization, in which simulators or executors use TAO’s time service for time synchronization. The other one is “hard” synchronization, in which a utility such as “AboutTime” is used to reset the hardware clock of the computer, synchronizing with world standard time. With this approach, simulators or executors can get synchronized time reference directly from the local machine’s clock through system calls. RTDEVS/CORBA uses the second approach for time synchronization.

4.2.2 Time Confluence

In real-time simulation or execution, due to the nondeterministic delay or delay jitter, it is hard to get event messages as scheduled. According to the International Telecommunications Union (ITU-T) definition, *jitter* is defined as those phase variations with respect to a perfect reference that happen in a clock or data signal as a result of noise, patterns, or other causes. Jitter refers to delay variations in the network context. It describes the variations in the latency of a message transmission. In data networks, too much packet jitter causes a voice to sound garbled. Network components usually compensate for jitter with buffers. Jitter buffers store incoming packets and send them in a more constant stream. In this case, the size of jitter buffers affects the performance of the network. However, there is no optimal size of a jitter buffer because the buffer size will vary from network to network.

A similar concept is used in RTDEVS/CORBA to control time jitter in real-time simulation or execution. Sources of jitter in the RTDEVS/CORBA environment vary. One of the main sources of jitter is the instability in the computer clock. Other sources could be general-purpose OS, network configuration, and other software configurations that are used in the environment. Any system involved in the environment could be the source of jitter. The goal here is not to identify or control the major sources of jitter but to identify problems that jitter could introduce and provide the simulators or executors with a capability to handle jitter in a proper way.

Jitter hinders the real-time simulators or executors from getting messages as scheduled. This situation can be explained by the *GPT* model, which is shown in Figure 6a. This coupled *GPT* model has three atomic models: a *generator* model, a *processor* model, and a *transducer* model. The *generator* produces jobs every 1 second and then sends them to the *processor* and the *transducer*. The *processor* accepts incoming jobs when it is passive and then processes the job and sends the finished job to the *transducer*. The processing time for every job is 1 second, too. The *transducer* collects unprocessed and processed jobs and calculates the performance statistics of the *processor*.

In this example, the *generator* generates a job every 1 second, which is the same as the processing time of the *processor*. This means, in the ideal case, that the *processor* will get an internal event and external event at the same time. However, job generation in the *generator* does not occur at the exact moment that is scheduled in practice. Furthermore, it takes nonzero time to deliver the job to the *processor*. Therefore, as shown in Figure 6b, an external event can be delivered right before or after the internal transition has been executed, which forces the *processor* model to make unnecessary transitions. In the *transducer*, same thing happens. Two external events scheduled to arrive at the same time could be delivered within a small interval.

To reduce this kind of undesirable behavior, a confluent time window is introduced to act like the jitter buffer discussed earlier. Either an external or an internal event can trigger the simulator to perform a confluent checking routine. At the beginning of this confluent checking routine, the simulator or executor sets the confluent time window and waits for any successive events during this period. Figure 6c shows the confluent time windows set forth by an internal or external event in the *processor* and the *transducer*. At the end of the confluent checking routine, the real-time simulator or executor executes the confluent function, which specifies the state transition order between internal and external events that occur together.

The size of the confluent time window is provided by the modeler, and this size determines the time granularity of real-time simulation or execution. It is desirable for the modeler to provide an optimal size of the confluent window based on the delay characteristics of the network environment. If the size of the confluent window is too small compared to the jitter, it will not handle all the confluent problems. On the other hand, if the size is too large, this could result in overlapping with next scheduled event.

4.3 Delivering Messages in Real Time

Delivering messages in real time is crucial for the RTDEVS/CORBA framework. This feature can be enabled by using the real-time event service in TAO. The TAO’s real-time event service provides end-to-end QoS guarantees between objects communicating over the network. To get the desired QoS, the real-time requirements for each operation must be provided on the supplier side. A real-time scheduler propagates this information to consumer *RT_Infos* based on the dependency graph. The scheduler then uses the propagated information to order dispatches within a set of operations whose dependencies have been met.

We mapped DEVS ports to suppliers and consumers in the real-time event service, as shown in Figure 7. All the DEVS models that participate in a simulation session are coupled by DEVS ports so that messages generated by a source model can be routed correctly to the destination model based on the *RT_Info* specified. DEVS ports consist of two port classes: input ports and output ports. Output ports are mapped into suppliers, and input ports are mapped into consumers of the event service architecture. *pushInPort* class and *pushOutPort* class were created to incorporate these mapping relations in the framework. These two

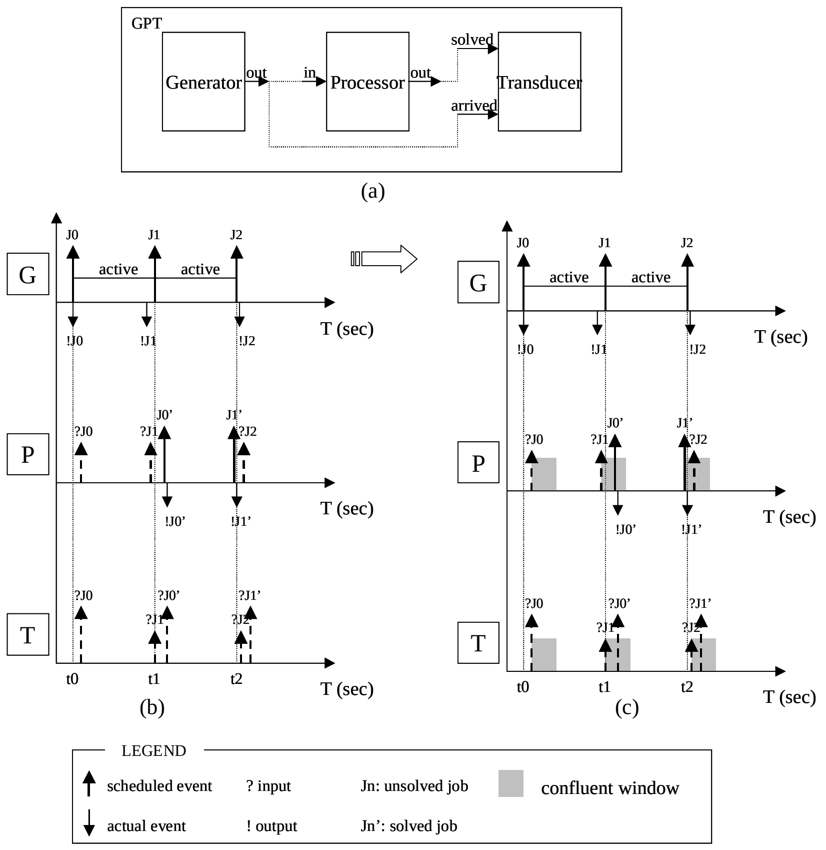

Figure 6. Confluent problem in real-time simulation or execution

classes maintain the port name, source identification, and *RT_Info* associated with an event to be handled as their member variables. When the coordinator downloads the coupling information at the initialization phase, the coordinator retrieves coupling information and the *RT_Info* table from the top coupled model and downloads it to the simulators. The *RT_Info* table contains real-time parameters associated with each event, and the events are distinguished by the name that is created by combining the model name and port name. The modeler must identify QoS information for each event before the assignment and provide the required QoS information through this *RT_Info* table. The coordinator parses the *RT_Info* table and then creates an event list based on the QoS information specified in the table before downloading the coupling information to the simulators. Upon receipt of the coupling information, each simulator creates *pushOutPort* and *pushInPort* accordingly and makes a connection to the event service, which makes a virtual event channel between the pair of *pushOutPort* and *pushInPort*. This virtual event channel is identified by the source identification assigned to the supplier connected to the event channel.

After downloading the coupling information, the coordinator requests the scheduler to compute priorities based on the given *RT_Info* and assigns dispatching priorities to each event. In this framework, each *pushOutPort* and *pushInPort* pair handles only one event, so that the priority assigned to the event can be inherited to the virtual event channel created between the *pushOutPort* and the *pushInPort*.

For DEVS messages to be delivered through the real-time event channel, the DEVS message must be converted into TAO's event format. The event consists of two fields: the event header and the payload. The event header contains routing information, and the data go into the payload. The data are serialized into a byte stream before they go into the payload.

4.4 Getting Data from External Environment Systems

For the real-time simulation and execution, there need to be means to get external input data from the sensors and to send out data to actuators. It is desirable for the

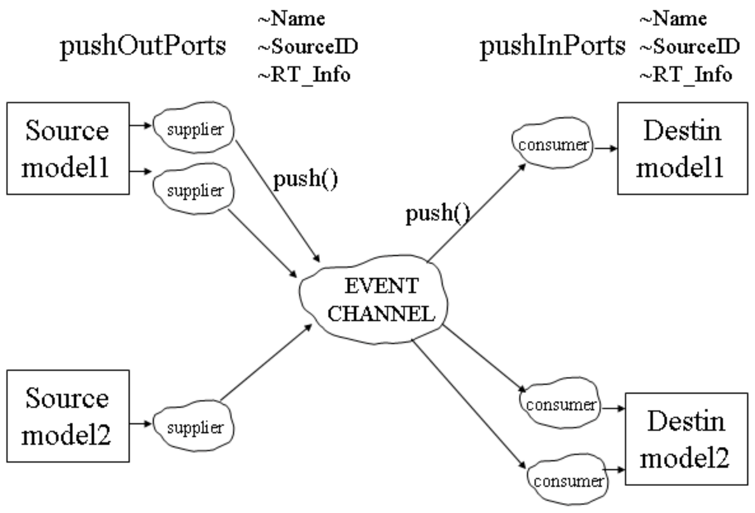

Figure 7. Mapping suppliers/consumers to discrete event system specification (DEVS) ports

real-time model to handle all the external events in the same external transition function in the same manner, regardless of their type. In RTDEVS/CORBA, a modeler can specify an external interface type using the *setExternalInterface()* method. When the simulator initializes the model, the simulator checks the specified external interface types and configures the interfaces accordingly. For example, as depicted in Figure 8, if the model expects external data through socket connections from the sensors, the modeler can specify a socket interface in the model, and the simulator sets up the connection. The real-time simulator creates proxies: sensor proxy and actuator proxy. These proxies are waiting for any data stream, and once data arrive, then they call the simulator's external input-handling method immediately. This method converts socket stream data into a DEVS message and puts it into the model as an external event message so that the model handles the external sensor data as a DEVS message. Of course, a model still can be coupled with other models while receiving data from the external environment.

User-friendly external interfaces are enabled by employing a specialized coupling concept, namely, reserved internal coupling (RIC). The RIC couples the ports attached to the model with the simulator. The real-time simulator has reserved internal ports such as *extInputFromSocket* and *extInputFromKeyboard* as input ports and *extOutputToSocket* and *extOutputToScreen* as output ports. The interface between activities and the model is defined in the same manner. In this case, the simulator starts an activity thread instead of proxies, and these are connected through other reserved internal couplings such as *resultFromActivity*, *cancelActivity*, and so forth.

4.5 Implementation of Activity

An activity is defined to incorporate real computations within simulation models in the real-time DEVS formalism. Such an activity is implemented as an activity thread in RTDEVS/CORBA so that a modeler can assign any computation tasks to the thread.

In the DEVS formalism, a model schedules the next event time using the *hold_in* method, which has the format of *hold_in(phase, sigma)*, where the *phase* is the control state in which the model must be kept for the amount of time specified by *sigma*. The *hold_in* method is also used in the RTDEVS/CORBA environment to schedule the next event time. In this case, however, *sigma* is mapped into the real-time clock (i.e., the value of 5 for *sigma* means 5 seconds in the physical real-time clock). To assign an activity to a certain state for a certain time period, the modeler can use the extended version of the *hold_in* method, which has the format of *hold_in(phase, sigma, activity)*. The activity is a real computation that must be performed to get some results. For example, this could be a DEVS simulation or a piece of computing software that takes some time to get the result. Through this version of the *hold_in* method, the modeler can assign a deadline to be associated with the activity. In this case, the modeler must make sure that the computation can be performed within the specified time window before he or she assigns the activity to the model.

The two versions of the *hold_in* methods also help to maintain the model continuity of the system under design. This is because when a real-time model is developed in the logical simulation environment such as DEVS-Java, the model can use the old version of the *hold_in* method. When the logical behavior of the model is verified and

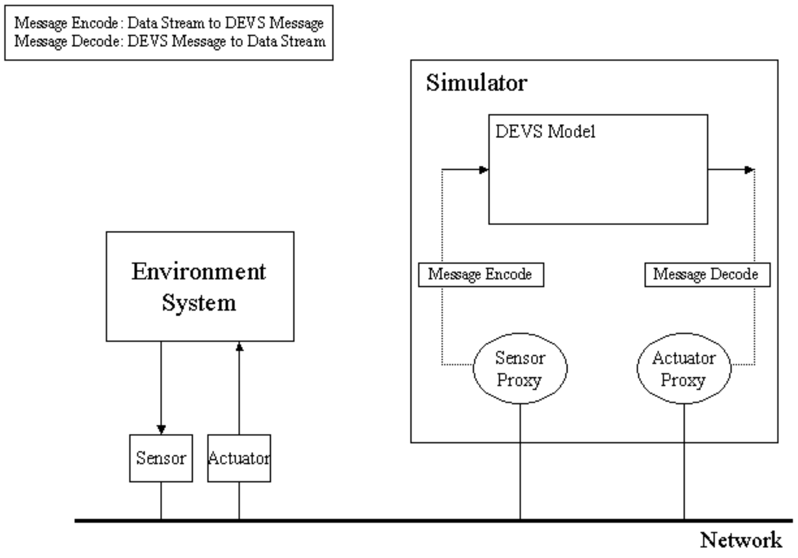

Figure 8. Design of an external interface for the real-time simulator

the model moves to the execution stage, the model can be migrated into the RTDEVS/CORBA environment with the extended *hold_in* method with activity.

5. An Example of Distributed Temperature Control with Hierarchical Scheduling

In this example, we show the ability of the distributed RT-DEVS/CORBA environment to support development of a complex distributed real-time system using real-time simulation. We show how model continuity may be maintained from the very beginning of the development stage to the execution of the model. The example is a distributed intelligent temperature control applied to a building using a hierarchical scheduling scheme. The building could be a hotel, an office structure, a hospital, or any other buildings that have many floors with multiple rooms on each floor. The requirement calls for each room in this building to be controlled individually, as specified by a manager at all times. The overall model architecture is depicted in Figure 9. The model consists of three components: the control system, the monitoring unit, and the building unit. The control system is the real-time system that needs to be developed, and the building unit is the environmental system model that would provide a virtual environment for the real-time system to be developed. Therefore, the building unit must provide outputs through the specified interfaces with highly accurate timing. The system also has a monitoring unit to provide the real-time monitoring capability.

The control system is further divided into the master scheduler and several control units, each of which consists of a master controller and subcontrollers. A master controller is assigned to each floor and controls the subcontrollers, each of which is assigned to a room. The master scheduler is the main interface unit through which the manager can specify the master control schedule, and the master scheduler maintains and distributes the master control schedule to each master controller accordingly. The master controller delivers the control schedule to each subcontroller and monitors the behavior of the subcontrollers. If an error should occur in a subcontroller, then the subcontroller would send out an error message to the master controller, and the master controller would process an error routine for the subcontroller. The building unit consists of multiple floors, each of which is again composed of many rooms.

To construct the model, we assumed that the building had the same floor plan on each floor. Since each floor of the building has the same structure, we only constructed a model for one floor with 10 rooms, as shown in Figure 10. One control unit is assigned to one floor unit model while each subcontroller is interacting with its counterpart, which is a room unit model. The real-time system, which would be developed and migrated from the simulation environment to the execution environment, is the control unit. As discussed before, the control unit model is composed of a master controller and subcontrollers, and the master controller is supposed to get updated control schedules from the master scheduler.

The master scheduler is to provide interfaces for a manager to input, as well as modify and retrieve the schedule. It is also required to store the schedule in an organized

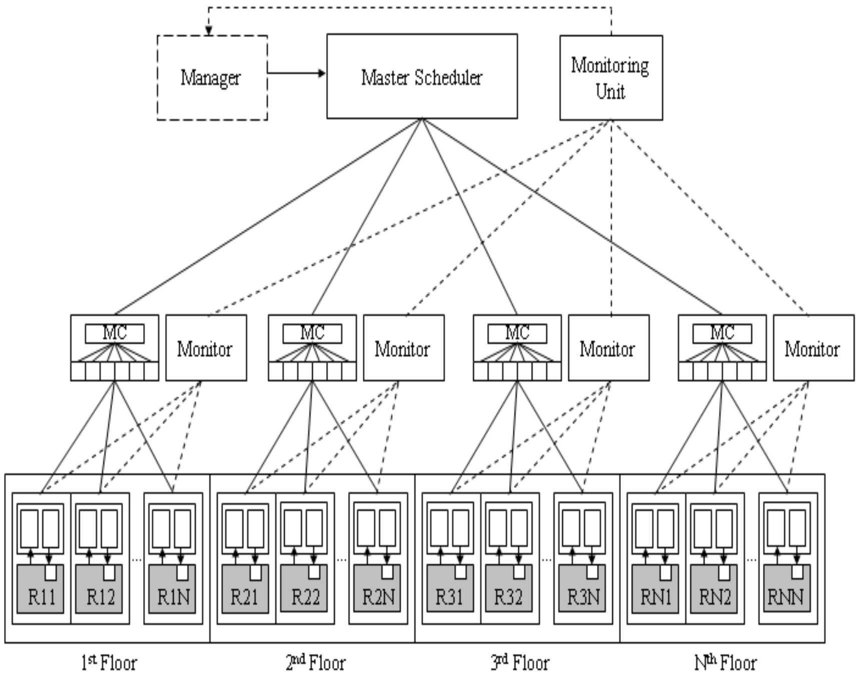

Figure 9. Overall structure of distributed temperature control model

manner and to distribute the schedule to control units when the time arrives. Kim [18] proposed a method for hierarchical scheduling in an environmental control. He used the system entity structure (SES) to organize the environmental control schedule in a hierarchical manner. This scheme is employed in the master scheduler to specify the overall control schedule in a hierarchically organized format, in which the control schedule for each room can be specified in multiple levels.

While the master scheduler maintains the master schedule, it periodically distributes the schedule to the master controllers. Upon receipt of the schedule, the master controller delivers it to subcontrollers, each of which controls its assigned room as scheduled. For example, the master scheduler dispatches a daily control schedule to each master controller, and then the master controller similarly dispatches different set points to each subcontroller according to this time schedule. The subcontroller is the main controller that maintains room temperature as specified in the schedule. Even though the subcontroller maintains room temperature, a guest can adjust room temperature at any time, and this adjustment overrides the regular schedule.

The floor unit is a real-time simulation model that provides an artificial environment for the control system. This floor unit model consists of multiple-room unit models, each of which is composed of a room model, a sensor model, and an actuator model. The room model calculates the current room temperature based on a given function, which is a simple linear function in this example, and provides the current room temperature upon request by the sensor model. The sensor model senses the room temperature and delivers it to the subcontroller every 10 seconds. The actuator model interprets commands from the subcontroller and delivers commands to the heater or the cooler in the room.

Figure 11 depicts the state transition diagram of the subcontroller model, which actually controls the room temperature. The subcontroller implements the event-based control scheme. The subcontroller only accepts sensing data during the *wait* phase, which is the time window (*t* seconds) for a valid input, and generates a command. Otherwise, it goes into the *error* phase and then generates an error message that goes to the master controller. When the master controller receives an error message from the subcontroller, it sends a *restart* signal along with a set point again. The master controller also provides a set point to the subcontroller at the beginning of every day.

First we built the model in DEVS-Java and checked its behavior to see if it worked correctly in terms of logical dynamics. Then the model was moved into the real-time DEVS/CORBA environment to check its timing behavior. In this example, a control schedule is provided to the master controller in a table format, which contains 10 different set points for each room. For the purpose of the example, we scaled down the time schedule for the master controller to distribute each set point every 10 minutes. This time

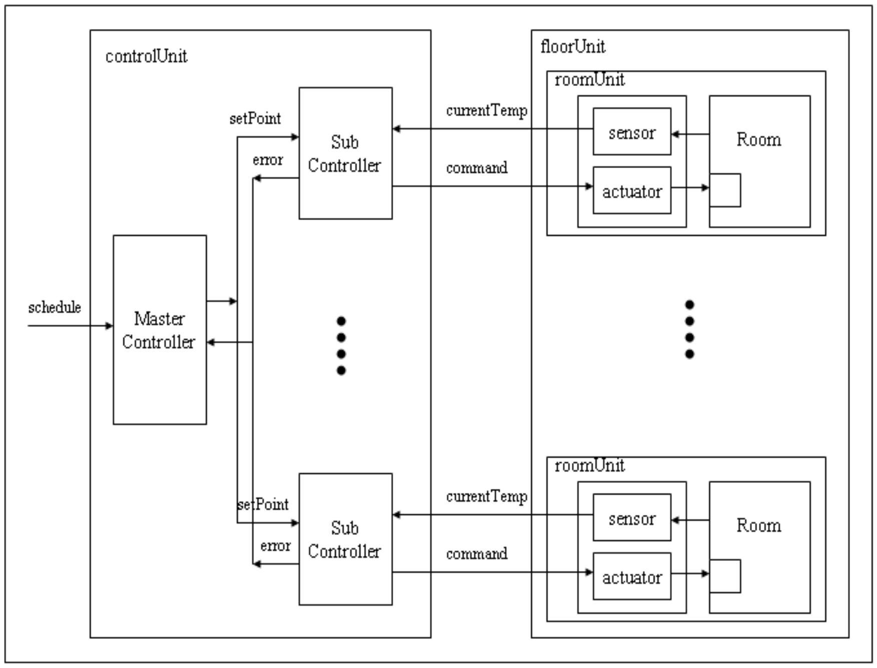

Figure 10. Minimized model architecture

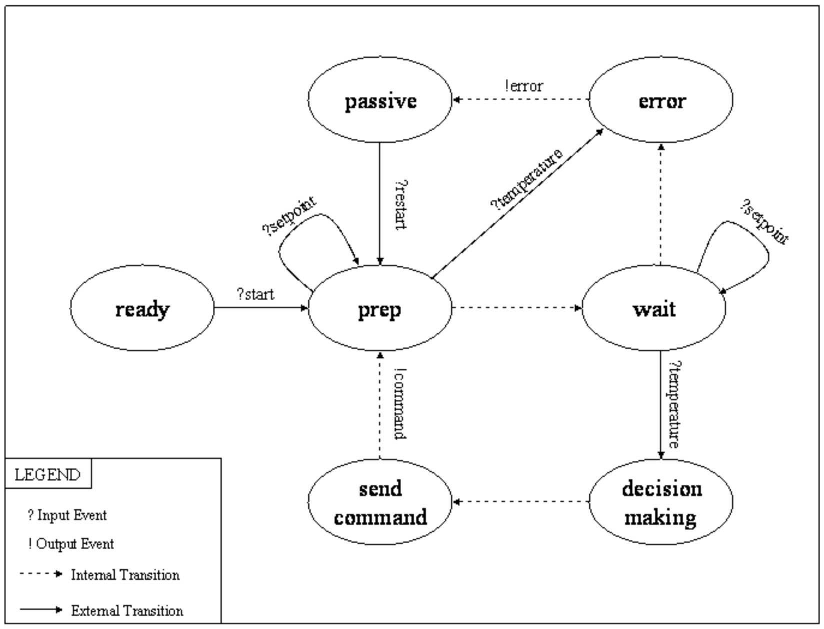

Figure 11. Subcontroller model

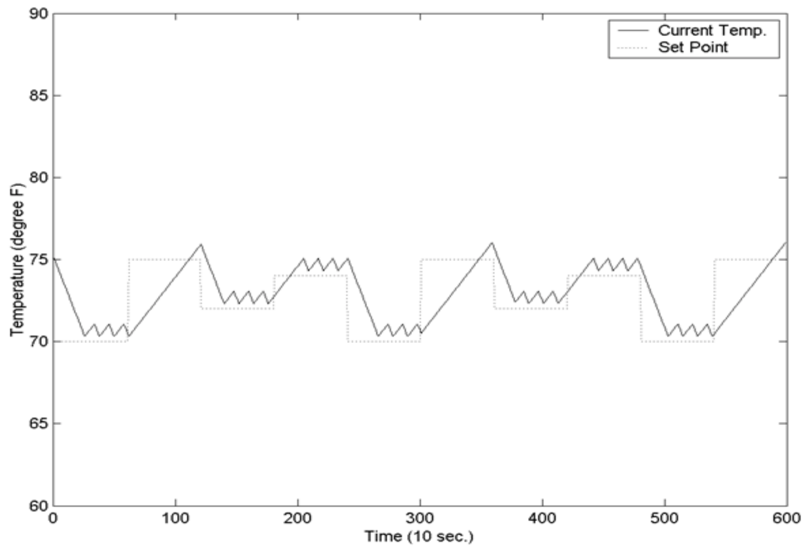

Figure 12. Result of temperature control without user interruption

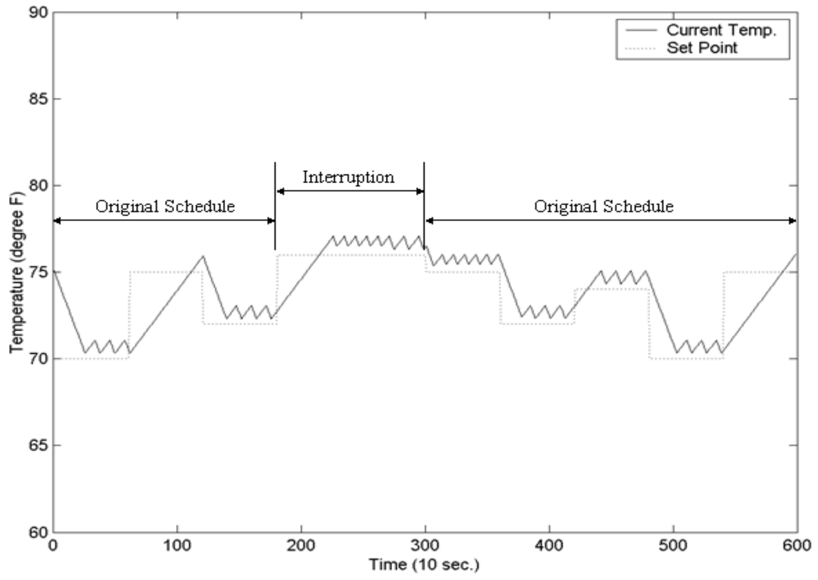

Figure 13. Result of temperature control with user interruption

schedule can be adjusted later on as needed. The model was executed in two ways: with and without user interruption. *User interruption* refers to the change of the set point by the user of the room. While running the model, we attached a monitoring window to each room to see room temperature changes in real time. One monitoring window is connected to the output port of each sensor model of the room unit so that the current room temperature can be monitored. Figure 12 shows the result of the temperature control of the first room without any interruption by the user, whereas Figure 13 shows the result with interruption. These results indicate that all the models are working correctly in terms of both logic and timing.

6. Conclusion

In this paper, a modeling and simulation framework, RT-DEVS/CORBA, was presented to support the development of distributed real-time systems. The framework supports model continuity for software development so that the same model can be reused with minimal change during different design stages—from model design to performance evaluation and even to final execution. To support model continuity, RTDEVS/CORBA adopted a layered architecture in which there is a separation of concerns underlying each layer. This layered architecture facilitates reuse of the software developed at different layers and makes it possible to choose different simulation and execution environments during different design stages while keeping the upper layer model unchanged.

To enable modeling and simulation in a distributed real-time environment, RTDEVS/CORBA has chosen ACE/TAO as the communication middleware, which supports distributed programming as well as real-time communication features such as real-time event service, real-time scheduling service, and QoS capabilities. Based on ACE/TAO, different implementation issues of RT-DEVS/CORBA such as the real-time simulator, time management techniques, and the real-time message delivery method have been studied and discussed in this paper.

7. References

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

[2] Zeigler, B. P., and J. Kim. 1993. Extending the DEVS-Scheme knowledge-based simulation environment for real-time event-based control. *IEEE Transactions on Robotics and Automation* 9 (3): 351-6.

[3] Laplante, Phillip A. 1997. *Real-time systems: Design and analysis*. 2d ed. Piscataway, NJ: IEEE Press.

[4] Krishna, C. M., and K. G. Shin. 1997. *Real-time systems*. New York: McGraw-Hill.

[5] Ghosh, K., K. Panesar, R. M. Fujimoto, and K. Schwan. 1994. PORTS: A Parallel, Optimistic, Real-Time Simulator. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pp. 24-31.

[6] Hong, J. S., and T. G. Kim. 1997. Real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems: Theory and Applications* 7:355-75.

[7] Cho, S. M., and T. G. Kim. 1998. Real-time DEVS simulation: Concurrent, time-selective execution of combined RT-DEVS model and interactive environment. In *Proceedings of SCSC '98*, Reno, NV, pp. 410-5.

[8] Uslander, T. 1999. OPERA: A CORBA-based architecture enabling distributed real-time simulation. In *Proceedings of ISORC '99*, May.

[9] Lee, E. A. 1998. Modeling concurrent real-time processes using discrete events. UCB/ERL Memorandum M98/7, March, Berkeley, CA.

[10] Edwards, S. A. 1997. The specification and execution of heterogeneous synchronous reactive systems. Ph.D. diss., University of California, Berkeley.

[11] Kim, K. H. 1997. Object structures for real-time systems and simulators. *IEEE Computer*, August, 62-80.

[12] Kim, K. H., J. Liu, and M. H. Kim. 2000. Deadline handling in real-time distributed objects. In *Proceedings of ISORC 2000 (IEEE CS 3rd International Symposium on Object-Oriented Real-Time Distributed Computing)*, Newport Beach, CA, pp. 7-15.

[13] Galla, T. M., and R. Pallierer. 1999. Cluster simulation—Support for distributed development of hard real-time systems using TDMA-based communication. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, England.

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

[15] Schmidt, D. C., D. L. Levine, and S. Mungee. 1998. The design of the TAO real-time object request broker. *Computer Communications* 21 (4): 294-324.

[16] Schmidt, D. C. 1994. ACE: An object-oriented framework for developing distributed application. In *Prodeedings of the 6th USENIX C++ Technical Conference*, Cambridge, MA.

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

[18] Kim, T. G. 1990. Hierarchical scheduling in an intelligent environmental control system. *Journal of Intelligent and Robotic Systems* 3:183-93.

***Young K. Cho** is a major in the Republic of Korea Air Force, Nonsan City, Choongnam, Korea.*

***Xiaolin Hu** is a PhD student at the Arizona Center for Integrative Modeling and Simulation, Department of Electrical and Computer Engineering, University of Arizona, Tucson.*

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