Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

INTRODUCING A UML MODEL FOR FASTER-THAN-REAL-TIME SIMULATION*

Dimosthenis Anagnostopoulos
George-Dimitrios Kapos

Harokopio University of Athens
70 El. Venizelou Str, 17671
Athens, GREECE

Vassilis Dalakas
Mara Nikolaidou

University of Athens
Panepistimiopolis, 15771
Athens, GREECE

ABSTRACT

Faster-than-real-time simulation (FRTS) is used when attempting to reach conclusions for the near future. FRTS experimentation proves to be the most demanding phase for conducting FRTS, since it requires concurrent monitoring and management of both the real system and the simulation experiments. Having previously introduced a conceptual methodology and specification for conducting FRTS experiments, we now propose an implementation framework, based on the Real Time Unified Modeling Language (RT-UML). The derived RT-UML model includes specific timing attributes and is independent of the application examined via FRTS. Thus, implementation of FRTS program modules can be analyzed and realized, following the guidelines of this model, ensuring the reliability of the results within predetermined time frames. A pilot application regarding FRTS implementation based on the proposed RT-UML model and related experience is also discussed in the paper.

1 INTRODUCTION

Real time simulation is widely used for the performance evaluation of systems behavior in real time. When attempting to reach conclusions for the near future, FRTS is used where the advancement of simulation time occurs faster than real world time. Making models run faster is the modeler's responsibility and rather demanding, since RT systems often have hard requirements for interacting with the human operator or other agents. Relevant methodological issues and some solutions have been discussed in Lee and Fishwick (1999) on the basis of selecting amongst models that employ a different level of abstraction. Current FRTS research directions involve the distribution of experiments over a network of workstations, intelligent control (Caia et al. 1996) and fault diagnosis (Norvilas et al. 2000), interactive-dynamic simulation (i.e., manipulation of the simulator by the user in RT) (Tyreus 1997). The increased interest in the field is reflected by the DDDAS program of the NSF (NSF 2005).

A conceptual methodology for conducting FRTS experiments, meeting the RT constraints, has been envisaged in Anagnostopoulos et al. (1999). The authors have developed in the past FRTS tools for different domains, such as networks (Anagnostopoulos and Nikolaidou 2001) and transportation systems. Recently, a first level description of an integrated specification for FRTS systems with RT-UML (Anagnostopoulos et al. 2004), leading to standardized implementations of such systems that meet strict time requirements was introduced. The profile, also used in Bertolino et al. (2002), enables the detailed specification of critical time and synchronization requirements for FRTS components and the overall performance evaluation. A detailed description of this framework is presented here. Moreover, an FRT simulator was implemented following these guidelines.

In Section 2 of the paper an FRTS overview is provided. Section 3 presents a short description of RT-UML profile used for our model. The UML model's high level description is given in Section 4 and a thorough specification is explained in Section 5. Section 6 presents an application example and some performance results from its execution within the proposed framework.

2 FRTS METHODOLOGY OVERVIEW

In Anagnostopoulos et al. (1999), a methodology that establishes a framework for conducting experiments dealing with the increased complexity and the hard RT requirements was introduced. The following simulation phases were identified: *modeling, experimentation and remodeling*. RT execution concerns only the two latter phases. During experimentation, both the system and the model evolve concurrently and are put under monitoring. Data depicting their consequent states are obtained and stored after predetermined, RT intervals of equal length, called *auditing*

*This research was supported in part by Pythagoras program (MIS 89198) co-funded by the Greek Government (25%) and the European Union (75%).

interval (*AudInt*). In case the model state deviates from the corresponding system state, remodeling is invoked. This may occur due to either modification of the system structure and operation parameters, or the stochastic nature of simulation. To deal such deviations, remodeling modifies the model in order to represent the current system state. This is accomplished without terminating the RT experiment, that is, without performing recompilation. When model modifications are completed, experimentation resumes from the current RT points, while all previous simulation results (i.e., predictions) are discarded. We use preconstructed models which are prevaluated. This is the only way to perform remodeling in real time. As already stated, model invalidity may be caused due to stochastic nature of simulation.

FRTS provides a natural way for accomplishing model validation through comparing the corresponding system data and model results. In case results (predictions for the near future) are valid over a number of consecutive time intervals, they may be used to reach conclusions and take appropriate actions for the system future state.

Experimentation in FRTS thus comprises monitoring, i.e., obtaining and storing system and model data during the auditing interval, and auditing, i.e., examining if the system has been modified during the last auditing interval or the model no longer provides a valid representation of the system due to stochastic nature of simulation.

To determine whether the system has been modified, specific system features are put under monitoring. The variables used to obtain the corresponding values are referred as *monitoring variables* (MVs). Auditing examines MVs corresponding to the same RT points (i.e., the current system state and simulation predictions for this point) and concludes for the validity of the model, as depicted in Figure 1. Auditing is performed at t_{n-1}, t_n, t_{n+1} and, thus, compares states S_x and R_n at time point t_n. If model validity is consecutively ensured at a number of consecutive auditing

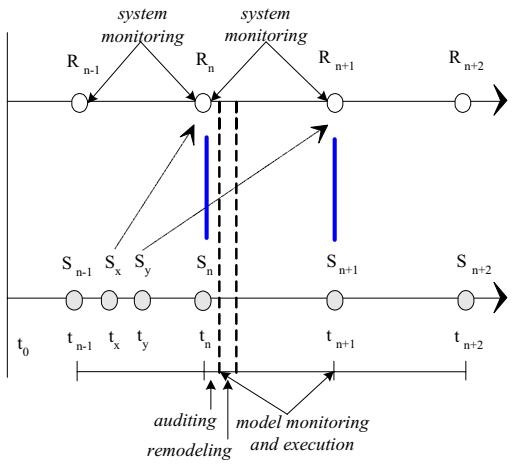

Figure 1: Experimentation in FRTS

intervals $[t_{n-2}, t_{n-1}], [t_{n-1}, t_n], \ldots$, simulation predictions are also considered to be valid. Assuming that auditing is invoked at t_{n-1} and that we wish to reach predictions for t_y within auditing interval $[t_{n-1}, t_n]$ (Figure 1), *prediction interval* (*PredInt*) is equal to $t_y - t_{n-1}$. Evidently, $PredInt > AudInt$. We usually choose a value p so that $PredInt = p \cdot AudInt,\ p \in N^*$.

Both model and system monitoring are executed while the model is running, that is, while predictions are reached within the given time frame (i.e., *AudInt*). Within this interval, auditing and remodeling (if necessary) must as well be completed. As auditing examines model validity at the end of the auditing interval, it is not possible to perceive system modifications occurring in the meantime. To deal with this, *state auditing* is an additional activity executed in predetermined *state auditing intervals* (*StAudInt*), that a) examines the current system state to detect system modifications, and b) consumes no time, as is executed without pausing the model. If the system has been modified, model execution terminates, results are discarded and remodeling is invoked, as in the case of auditing. Evidently, $AudInt = n \cdot StAudInt, n \in N^*$. As monitoring consumes time equal to model execution time (T_E), the condition for achieving FRTS within the given time frame is:

$$T_E + T_A + T_R \leq AudInt \tag{1}$$

where T_A, T_R are the time periods consumed for auditing and remodeling, respectively (Figure 1). A more detailed discussion on timing issues in FRTS can be found in Anagnostopoulos and Nikolaidou (2003).

Model validation activities are accomplished though comparing the corresponding values of MVs for the real system and the model. Each monitoring variable MV_i is characterized by the following: *name*, *system value* (r), *model value* (s), *deviation range* (dr) and a *state monitoring indication* (smi), which specifies whether this variable is used in state auditing. A simple comparison for single-valued variables examines if the model value ($MV_i.s$) is within an interval constructed around the system value ($MV_i.r$). *Deviation range* (dr) determines the half-length of the interval. In this way, for monitoring variable i, the model is considered to be valid when

$$MV_i.s \in [MV_i.r \cdot (1 - dr),\ MV_i.r \cdot (1 + dr)] \tag{2}$$

3 RT-UML MODELING FRAMEWORK

Unified Modeling Language (UML) is the result of an effort to unify concepts among distinct methodologies, made by the authors of three leading methodologies –Rumbaugh, Booch, and Jacobson (Rumbaugh et al. 1999, OMG 2001a). Currently, UML has been adopted as a standard by the Object

Management Group (OMG) and is considered a fundamental skill for software engineers.

UML does not provide the required degree of precision (regarding timing issues) for the specification of FRTS. Thus, we use RT-UML (OMG 2001b), which enhances UML diagrams. The profile, also used in Bertolino et al. (2002), enables the detailed specification of critical time and synchronization requirements for FRTS components and the overall performance evaluation. RT-UML does not propose new model analysis techniques, but it rather enables the annotation of UML models with properties that are related to modeling of time and time-related aspects. Therefore timing and synchronization aspects of FRTS components are defined and explained in terms of standard modeling elements. RT-UML has a modular structure that allows users to use only the elements that they need. It is divided into two main parts (General Resource Modeling Framework and Analysis Models) and is further partitioned in six subprofiles, dedicated to specific aspects and model analysis techniques. Since the emphasis of this work is on time and concurrency aspects of FRTS systems, we only use elements from the General Time Modeling and General Concurrency Modeling subprofiles.

Each subprofile provides several stereotypes with tags that may be applied to UML models. A stereotype can be viewed as the way to extend the semantics of existing UML concepts (activity, method, class, etc.). For example, a stereotype can be applied on an activity, in order to extend its semantics to include the duration of its execution. This is achieved via a new tag added to the activity, specifying the execution duration. Stereotypes define such tags and their domains.

The proposed FRTS model consists of RT-UML enhanced diagrams, which are annotated according to the conventions used in the RT-UML profile specification and its examples (OMG 2001b). Stereotypes applied to classes in class diagrams are displayed in the class box, above the name of the class (a in Figure 2). However, when tag values need to be specified for a certain stereotype, a *note* is also attached (b in Figure 2). In sequence diagrams, event stereotypes are displayed over the events, while method invocation and execution stereotypes are displayed in *notes* (c in Figure 2). In activity diagrams, *notes* are also used to indicate the application of a stereotype on an activity, state or transition (d in Figure 2).

The RT-UML stereotypes used in this paper focus on timing, concurrency and synchronization issues, providing considerable precision in the specified model. In class diagrams of this paper we use the CRconcurrent and RTtimer stereotypes. CRconcurrent is used for classes of objects that may be executed concurrently. A CRmain tag holds a reference to the method that should be invoked once the object moves to "executing" state. RTtimer models a timer mechanism. Tag RTduration specifies the duration of the

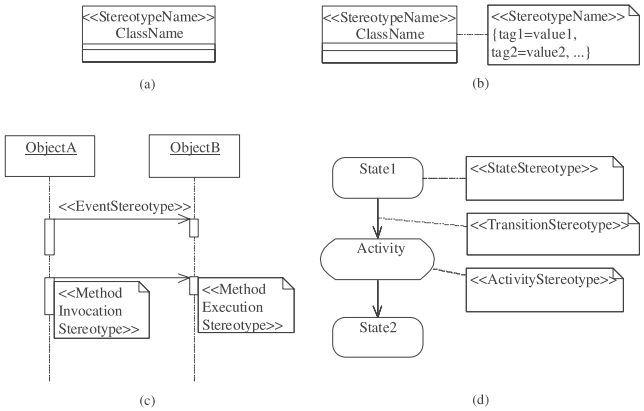

Figure 2: RT-UML Notation

timer mechanism, while RTperiodic indicates whether the timer is periodic or not.

In sequence diagrams we use the RTevent, CRimmediate, CRsynch, CRasynch, RTnewTimer, RTstart and RTaction. RTevent models events of message dispatches, specifying the time instance they occur (through the RTat tag). CRimmediate is also used for message dispatches to indicate that no time is consumed until the message reaches its destination. The CRthreading tag of this stereotype defines the thread that will execute a method (as a result of the message): the thread of the receiver (value "local") or the thread of the sender (value "remote"). CRsynch and CRasynch are used to indicate whether a method is invoked synchronously or not. Stereotype RTnewTimer models methods that create new timers and RTstart is used for events that start timing mechanisms. Finally, RTaction is used for methods, specifying the instance they start (tag RTstart) and their duration (tag RTduration).

In activity diagrams we use the RTaction and RTdelay stereotypes. RTaction was described earlier, while RTdelay is used for pure delay states, specifying their start, end and duration. Table 1 summarizes the RT-UML stereotypes used in the proposed FRTS model, their tags, the concepts applying to, and the diagram types they are used in.

4 FRTS: A HIGH-LEVEL DESCRIPTION

An object-oriented specification of FRTS is provided in this section. In Figure 3, a UML *use case* diagram is depicted, including all entities involved in FRTS. Both the system and the model, are separate from the main module of FRTS and handled independently. *System environment* (SE) represents the actual system and a surrounding mechanism facilitating system monitoring. It is considered as a separate entity that interacts with the FRTS system. *Model environment* (ME) includes the model and its execution environment (MEE), while the *FRTS System* process is the software module responsible for controlling FRTS. Finally, the user is the actor that enables the whole process, providing the case study.

Table 1: RT-UML Notation

Stereotype	**Tags**	**Applied to**	**Diagram type used in**
RTaction	RTstart, RTend, RTduration	Activity, Method	Activity and Sequence
RTdelay	RTstart RTend, RTduration	State	Activity
RTevent	RTat	Event	Sequence
RTnewTimer	RTtimerPar	Method	Sequence
RTstart	-	Event	Sequence
RTtimer	RTduration, RTperiodic	Class	Class
CRasynch	-	Method	Sequence
CRconcurrent	CRmain	Class	Class
CRimmediate	CRthreading	Event	Sequence
CRsynch	-	Method	Sequence

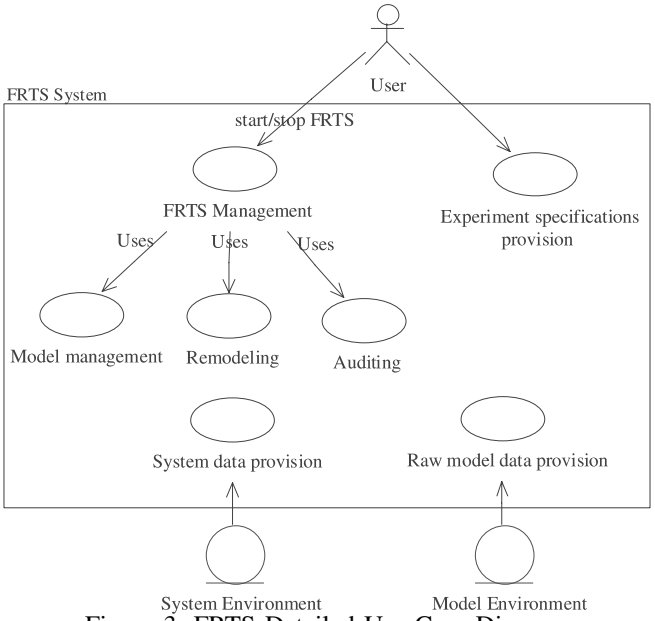

Figure 3: FRTS Detailed Use Case Diagram

The *user* provides the *experiment specifications* and manages the *FRTS System* process by starting or stopping the experiment.

System and *model environment* entities provide *raw system data* and *raw model data*, respectively. The *FRTS System* process performs auditing to identify potential deviations between the model and the system. In case such a deviation is indicated exceeding a respective remodeling threshold, remodeling is invoked (Remodeling), which results in the construction of a new model that replaces the one currently used (Model management).

We focus on the *FRTS System*, as the FRTS coordinating entity. The activity diagram depicted in Figure 4 provides a description of *FRTS System* process. The user is obliged to provide experiment specifications to the process with the

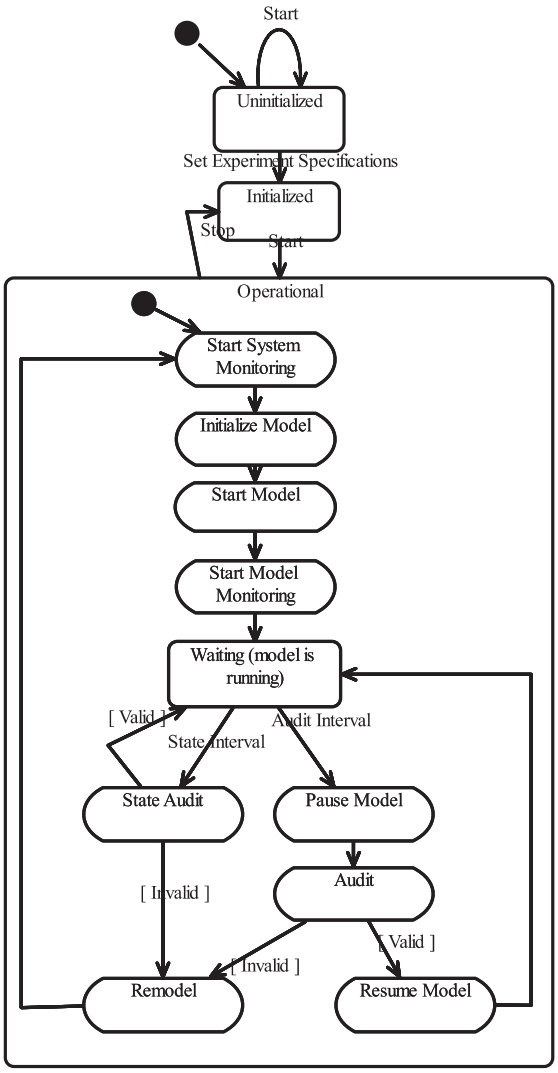

Figure 4: FRTS System Activity Diagram

SetExperimentSpecifications command. Then, *start* initiates the experiment, transiting to the *Operational* state.

System monitoring is considered to be performed autonomously by the real system with the aid of expert sensors that store monitoring information. The contribution of *Start System Monitoring* activity is restricted in stimulating the aforementioned sensors to start collecting and recording data by sending the appropriate event to *SE*.

Based on the experiment specifications, an initial model is being created (*Initialize Model* activity) using classes from predetermined libraries. Model environment is considered separate from the FRTS environment (e.g. it could be a DEVS-based execution environment). Therefore, *Start Model* activity simply tells *ME* to start simulation and is used for synchronization purposes. Model monitoring is considered to be performed by the *ME* which stores monitoring data. Thus, model and system monitoring are performed concurrently and autonomously, collecting data from both. Model monitoring is executed for a time period

equal to auditing interval, such as $[t_{n-1}, t_n]$ in Figure 1, during which the *FRTS System* process mainly remains in state *Waiting* (Figure 4). Model execution is then paused and *Audit* is invoked. *Audit* determines if the model still provides a valid representation of the system. If invalid, *Remodel* is invoked. Otherwise, *MEE* is informed to resume execution and monitoring of the model.

In a smaller time interval (*state interval*) than the auditing interval, the *FRTS System* process leaves *Waiting* state, to perform the *State Audit* activity. *State Audit* handles critical, such as structural, modifications of the real system, where remodeling must be performed instantly to restore consistency between the model and the system. Model monitoring is disabled during *Audit* and *Remodel*. On the other hand, system monitoring is never terminated, so that system changes can always be perceived. The only modification it experiences is that it is restarted for synchronization purposes after *Remodel*.

5 FRTS SYSTEM SPECIFICATION

UML semantics were adequate to represent FRTS system operation in a high-level of detail, since there was no need to represent timing constraints between FRTS specific activities and system/model environment.

In this section a specification for FRTS systems implementation is provided. First, *FRTS system* main classes and interfaces are presented in a class diagram. Then, FRTS main operations are presented using activity and sequence diagrams. RT-UML semantics are included in the diagrams in both cases mainly to indicate concurrent execution of activities, the need for synchronization and timing constraints.

5.1 FRTS Components

Figure 5 depicts the FRTS system design, based on a set of classes and interfaces. The classes are shortly described below (detailed descriptions are given in following subsections):

- *Context* is a utility class, used for storing the experiment specifications, references to the system monitor and the model environment, and monitoring variable values used for state auditing.
- *Control* class initiates the FRTS process.
- *StateAuditor*, *Auditor*, and *Remodeller* are responsible for performing the homonymous operations.
- *Timer* is responsible for producing *StateAudit* and *Audit* events, necessary for triggering *StateAuditor* and *Auditor*.
- Class *UserInterface* is simply the means for introducing user requests and data and therefore, is not further explained.

The following interfaces are also used:

- *IAuditor* interface defines the abstract behavior of an auditor and is implemented via *StateAuditor* and *Auditor* classes.
- *Monitor* interface models the abstract concept of a monitor for variables' values. Interfaces *SystemMonitor* and *ModelExecutionEnvironment* extend this interface to capture specific behavioral characteristics, required for system and model monitoring, respectively.

Classes *Control*, *Timer*, *StateAuditor*, *Auditor*, and *Remodeler* are intended to run on separate threads and therefore have the *CRconcurrent* stereotype. Objects of each of these classes operate independently and occasionally concurrently. The *CRmain* tag of CRconcurrent stereotypes indicates the method that is executed when objects of each class are activated. Class *Timer* has also the *RTtimer* stereotype, indicating that it is a timing mechanism that generates an event. Tags *RTduration* and *RTperiodic* further define the behavior of this timing mechanism, specifying its duration and indicating whether it is periodic or not.

No classes are specified for the system monitor and the model environment, since they are not part of the FRTS system. FRTS components require only communication interfaces with the system monitor and the model environment, denoted by *SystemMonitor* and *ModelExecutionEnvironment*.

5.2 Initiation of the FRTS Process

Figure 6 shows the sequence of messages exchanged by the FRTS system objects during initiation. This sequence diagram of the FRTS process starts when the user sends the *start()* event to the *Control* (through the *UserInterface*) at a random time instance t_y. The *start()* event causes the immediate execution of the homonymous method of the Control, as indicated by the *CRimmediateExecution* stereotype. Value '*local*' of the tag *CRthreading* shows that the *start()* method of *Control* is not executed by the thread of the invoking object (*UserInterface*), but by a separate, local thread of the *Control*. A '*remote*' value on this tag would indicate execution of the method by the thread of the invoking object. The *CRasynch* stereotype indicates that the invocation of the *start()* method is asynchronous, i.e., the invoking object does not wait for the execution of the method to be completed. At this stage several initiation messages are exchanged until the FRTS process reaches its stable state of periodic audits and state audits. This happens when the last message (*start()*) is sent to the *Timer* that will repeatedly produce state audit and audit events from this point on. All method executions are annotated with the appropriate *RTaction* stereotypes that indicate when

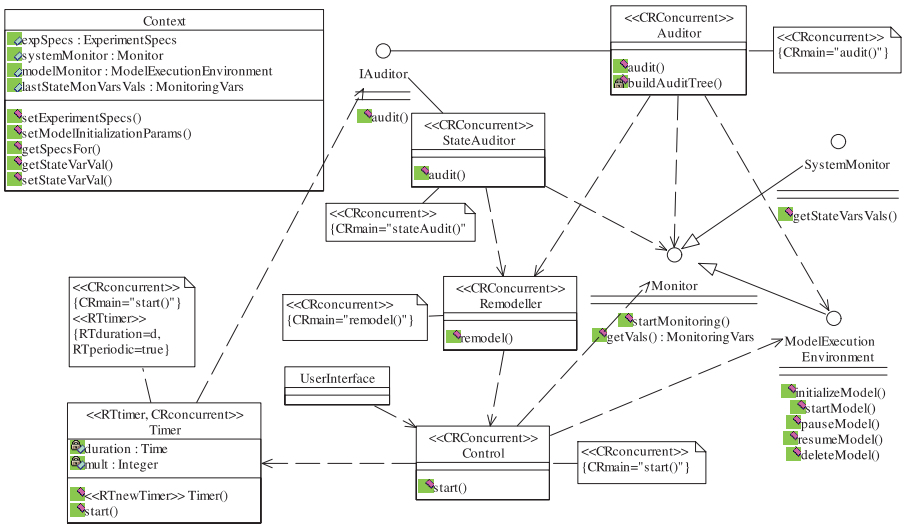

Figure 5: The Main FRTS System Classes

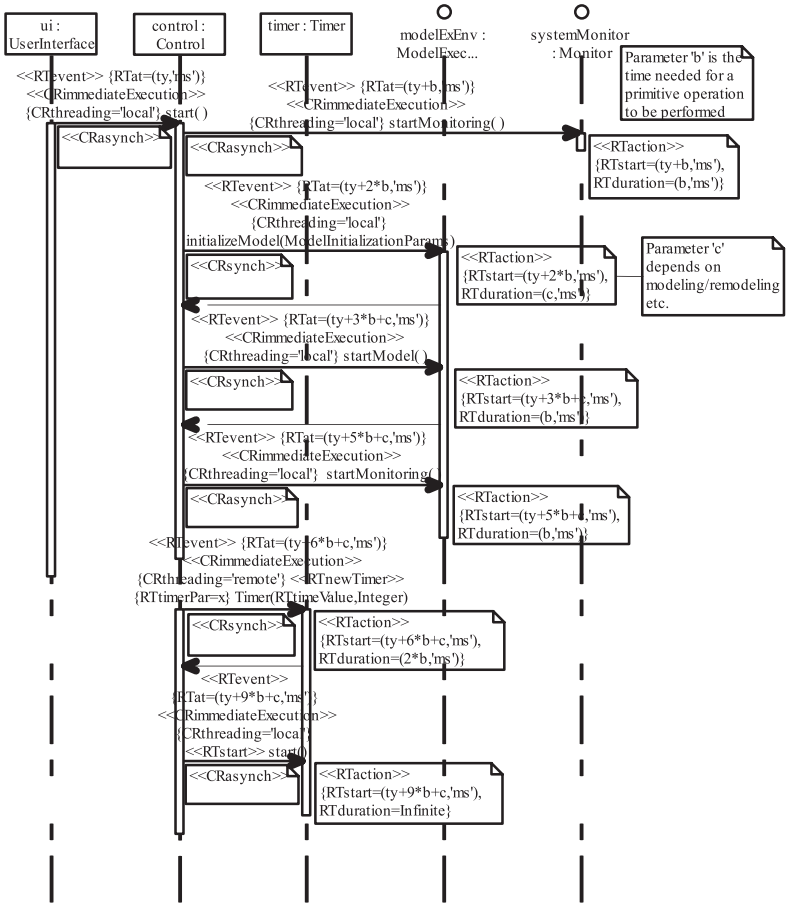

Figure 6: Sequence Diagram for Starting the FRTS Process

each execution starts (tag *RTstart*) and its duration (tag *RTduration*).

The use of RT-UML in the sequence diagram of Figure 6 clarifies thread synchronization and execution, determines event occurrence and action duration, and enhances its semantics. Thus, an in-depth and comprehensive view of the FRTS system is obtained.

The activity diagram of Figure 7 defines the functionality of the *start()* method of class *Control*. Each activity of

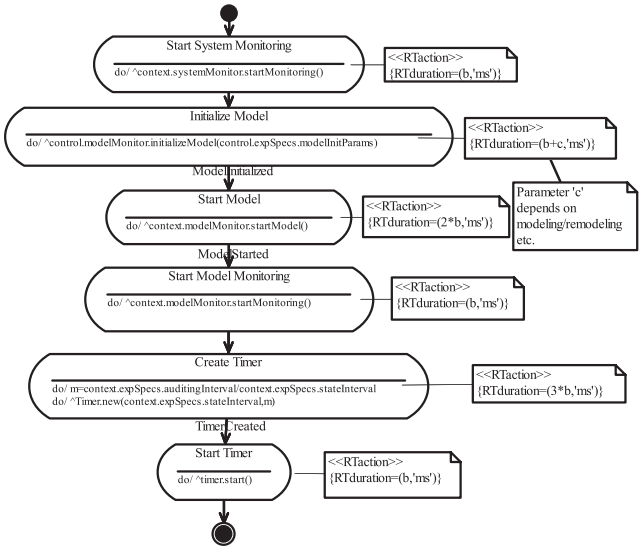

Figure 7: Activity Diagram for Method *start* of Class *Control*

the diagram is annotated with the appropriate *RTaction* stereotype note. Using this kind of stereotype and its *RTduration* tag, activities' durations are specified. The lower part (*do/*) of each activity defines the actions executed or messages sent. Message dispatches are denoted with the ^ symbol. The overall duration of *start()* method is $9 \cdot b + c$ ms, where b is the time needed for a basic operation to be performed (arithmetic operation, method invocation, etc.). Parameter c is the duration of model's initialization and depends on the experiment specification. The overall duration of *start()* refers to the duration from the time instance when the user sends a *start()* event until everything has been initialized and *Timer* is started.

5.3 Audit

Audit is the key experimentation activity determining model validity through comparing the corresponding system and model monitoring variables. Auditing is activated either after a *state interval* or an *audit interval*. Two distinct cases are thus considered: standard auditing and state auditing. Throughout this paper, the term *auditing* refers to standard auditing. *State auditing* is explicitly referenced.

During auditing, system modifications, involving its input data, operation parameters and structure, as well as deviations between the system and the model are examined to determine model validity. If remodeling is required, a *remodeling indication* is produced. All monitoring variables are used in this process.

Monitoring variable comparison is realized using the auditing tree, which is a conceptual tree structure. It is divided into two subtrees and includes two corresponding types of end nodes, *OR* and *AND*, as depicted in Figure 8. The audit activity constructs the auditing tree retrieving sys-

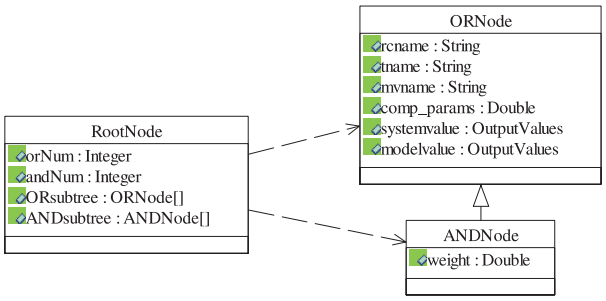

Figure 8: Auditing Tree Class Diagram

tem and model monitoring variable entries from the *System Monitor* and *Model Execution Environment*, respectively.

Both Audit and State Audit execution are restricted by strict timing constraints, since in both cases the auditing tree must be constructed in a small fraction of the audit/state audit interval. Furthermore, the auditing tree construction is bounded by system and model environments since monitoring variable values must be fetched from both of them. These restrictions are denoted in detail in corresponding sequence and activity diagrams, where RT-UML use offers the ability to estimate the time elapsed in separate activities or the whole auditing process in total. Hence, bottlenecks regarding the execution time of specific Auditing and Model/System Environment processes (e.g. comparing values of a monitoring variable) may be identified during analysis and Auditing implementation performance can be measured and validated with regard to Model/System Environment operation. For example, since the FRTS Modeler is able to realize the way the overall duration of audit depends on the number of monitoring variables or the fetching mechanism of System Environment, he/she may regulate the operation of all FRTS modules.

In Figures 9 and 10, the State Audit RT-UML sequence and activity diagrams are presented. As shown in Figure 9, *state audit* activity inspects the current system state to determine if reformations have occurred. In this case, the model no longer provides a valid representation and the relevant *remodeling indication* is produced. As indicated in the activity diagram in Figure 10, only variables designated as *state monitoring variables* are retrieved during *state audit*. Each of these variables is compared to its previous known value and the newer is stored. If the deviation between

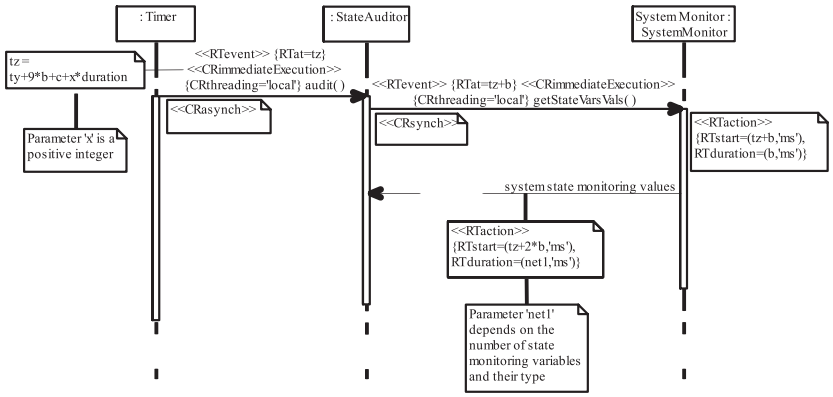

Figure 9: State Audit Sequence Diagram

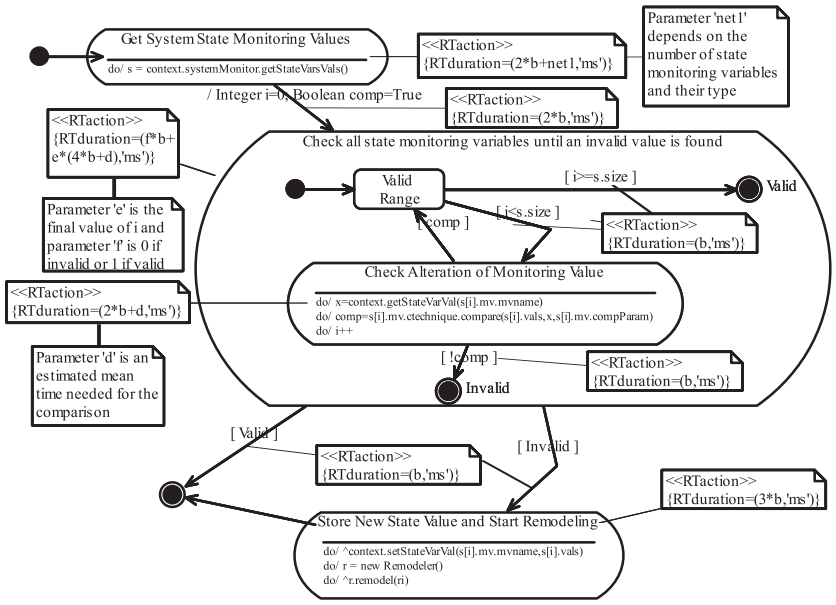

Figure 10: Activity Diagram for Method *audit* of Class *StateAuditor*

the two values supersedes the specified *compParam*, it is considered as invalid and the algorithm directly invokes remodeling to modify the model with minimum time overhead, without exhaustively examining the remaining state monitoring variables. Otherwise, the state auditor examines the remaining state monitoring variables.

As indicated in Figure 10, the overall duration of the *state audit* is $8 \cdot b + net1 + f$ ms, where $f \in [4 \cdot b + d, (4 \cdot b + d) \cdot e]$, d is the mean time for the comparison for one variable and e is the number of state monitoring variables.

6 AN FRTS APPLICATION EXAMPLE

Contacting FRTS experiments successfully, requires the execution time of specific activities to be similar to the time estimations reported in the model. An FRT simulator was implemented to coordinate an experimental system for model evaluation purposes. The FRT simulator was built in Java, following the guidelines of the RT-UML model presented in Sections 4 and 5. The Simulation Environment (SE) and simulation models were constructed in Java, as well. The simulator was easily implemented using Rational Rose platform that minimized programming effort. Although the FRT simulator implementation was based in the RT-UML model, the presented case study is not described in terms of UML diagrams as it deals with the execution of the derived system.

In the following, FRTS is applied in a two node web site, where the second node is used only in cases of heavy load (that is when FRTS predicts that each node load is over a certain threshold). Suppose that visitor inquiries are two kind of processing jobs J_1 and J_2 that fill two separate queues Q_1 and Q_2 respectively. Each job has an inter-arrival time λ_i and a predetermined service time ϵ_i ($\epsilon_1 \geq \epsilon_2$). Both queues are connected with a server S_i as illustrates Figure 11. Thus, the web site can be modeled as a Multi-Queue, Multi-Server System.

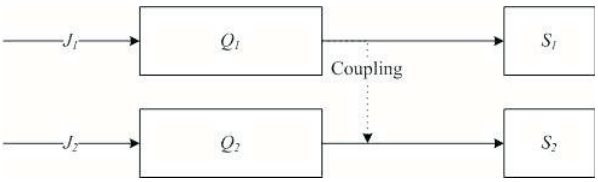

Figure 11: Example Topology

Denoting the average queue delay as d_i, we may define the following scenario for our case study. In the beginning each server serves only its associated queue (Coupling does not exist). However, if $d_1 \geq M_1$ and $d_2 \leq M_2$ we activate the coupling among the first queue Q_1 and the second server S_2, activating a mechanism that enables S_2 to serve one job from queue Q_1 each time its queue (Q_2) is empty. This mechanism is deactivated in case $d_1 \leq M_1$ or $d_2 \geq M_2$. Then again, each server serves only its associated queue (Coupling does not exist).

In order to conduct the experiment, detailed description of Monitoring Variables and Remodeling Conditions was needed during the initialization phase. As model initialization parameters, the following variables were used:

- 2 job types
- 2 queues
- 2 servers
- Inter-arrival parameter for job 1 = 3
- Inter-arrival parameter for job 2 = 3
- Service time for job 1 = 10 sec
- Service time for job 1 = 1 sec
- Average queue delay in queue 1 = 60 sec
- Average queue delay in queue 1 = 5 sec

thus, we have *modelInitParams* = (2, 2, 2, 3, 3, 10, 1, 60, 5). Having the model variables and the initialization parameters the MEE can now build the model and execute it.

Checking values for both system and model of these monitoring variables during auditing we apply remodeling following the described scenario and in case a server is down. Table 2 presents the results of the experimentation with the example described in this section and the FRTS simulator we built. Experimentation was conducted within Sun's Netbeans IDE and the Netbeans Profiler plug-in. Measured times (third column) are presented against estimated durations (second column) by the RT-UML FRTS model analysis. The table contains the most important time periods:

(i) Execution time of a basic operation. It entirely depends on the computer configuration where the experimentation is conducted. No theoretical estimation can be made. The measured value is substituted in the formulas that estimate other time periods.

(ii) Audit and state audit intervals.

(iii) Audit and state audit durations. A fundamental requirement is that state audit duration is less than state audit interval.

For each time period both estimated and measured, an average, a minimum and a maximum value are given.

Table 2: Basic FRTS Time Attribute Comparison

Time interval	**Duration in msec**					
	Theoretical Estimation			**Measured Time**		
	avg	min	max	avg	min	max
Time for basic operation	(b)	(b)	(b)	0.0377	0.0011	0.8229
Audit interval	5000	5000	5000	5004	4871	5278
Audit duration	5.479	0.155	123.441	2.700	0.090	53.400
State audit interval	1000	1000	1000	999	891	1106
State audit duration	0.565	0.017	12.344	0.233	0.087	10.700

As far as the estimated time periods are concerned, audit and state audit intervals are explicitly defined rather than estimated. Also, since the basic operation duration (b) is not estimated, but the measured time is used in other formulas.

State audit duration is estimated by the formula $8 \cdot b + net1 + f$, where $f \in [4 \cdot b + d, (4 \cdot b + d) \cdot e]$, d is the mean time for the comparison for one variable and e is the number of state monitoring variables. In our example there is only one state monitoring variable ($e = 1$) and the mean time for the comparison of the integer variable is two basic operations ($d = 2 \cdot b$). Also, as there is not any factor that would introduce delays in the reception of state monitoring variable values from the system, parameter $net1$ can be estimated to be equal to one basic operation duration ($net1 = b$). Therefore, the formula estimating the state audit duration becomes $15 \cdot b$. The respective cell is filled using the measured value for the basic operation duration (b).

Similarly, for the estimation of the audit duration, the formula $14 \cdot b + g + net2 + net3 + k$ is used. Parameter g is the time for the audit tree to be built, $net2$ and $net3$ depend on the number ($h = 9$) and the type of the monitoring variables, and $k \in [b + h \cdot (4 \cdot b + d), b + h \cdot (5 \cdot b + d)]$. Parameter g can be estimated to be $6 \cdot b$ times the number of monitoring variables ($6 \cdot b \cdot h = 54 \cdot b$). Like $net1$ in state audit duration, $net2$ and $net3$ are considered to be equal to $h \cdot b = 9 \cdot b$ each. Considering that $h = 9$ and $d = 2 \cdot b$, $k \in [55 \cdot b, 64 \cdot b]$. Therefore, audit duration

belongs in $[141 \cdot b, 150 \cdot b]$. The respective cell is filled using the measured value for the basic operation duration (b).

Comparing the theoretical estimations with the measured times in Table 2, the following conclusions are reached: a) audit and state audit intervals are quite accurate, b) estimated audit and state audit durations are comparable to the measured ones. Also, estimations for maximum audit and state audit durations are higher than the measured ones, indicating that the estimated maximum values may be used as the lower limit for audit and state audit duration.

7 CONCLUSIONS

The main objective of the work presented in this paper was to introduce a specification for FRTS experimentation, which was not domain-oriented and establishes common guidelines for developing FRT simulators. We adopted RT-UML to provide a thorough and complete model for FRT simulators emphasizing timing and concurrency issues. RT-UML enabled the description of time constraints imposed in FRTS, while modeling process was straight-forward, and no extensions were needed to describe FRTS. Detailed RT-UML diagrams specify how each FRTS component operates in terms of events, activities, and actions and infers estimations about time consistency and overall behavior of specific FRTS simulators. The behavior of FRTS simulators, apart from their implementation, strongly depends on the application domain and the experiment specifications used. Thus, time consistency of FRTS simulators may be completely justified only in the context of an application domain and specific experiment specifications. To this direction the proposed model quantified this interdependence and facilitates the evaluation of FRTS simulators in certain contexts.

REFERENCES

Anagnostopoulos, D., V. Dalakas, G. D. Kapos, and M. Nikolaidou. 2004. An RT-UML model for building faster-than-real-time simulators. In *Proceedings of Eurosim04*. Paris.

Anagnostopoulos, D., and M. Nikolaidou. 2001. An object-oriented modeling approach for dynamic computer network simulation. *International Journal of Modeling and Simulation* 21 (4): 249–257.

Anagnostopoulos, D., and M. Nikolaidou. 2003. Timing issues and experiment scheduling in faster-than-real-time simulation. *SCS Transactions on Computer Simulation* 79 (11): 613–625.

Anagnostopoulos, D., M. Nikolaidou, and P. Georgiadis. 1999. A conceptual methodology for conducting faster-than-real-time experiments. *Trans. Soc. Comput. Simul. Int.* 16 (2): 70–77.

Bertolino, A., E. Marchetti, and R. Mirandola. 2002. Real-time UML-based performance engineering to aid manager's decisions in multi-project planning. In *Workshop on Software and Performance*, 251–261.

Caia, Z.-X., Y.-N. Wangb, and J.-F. Caia. 1996. A real-time expert control system. *Artificial Intelligence and Engineering* 10 (4): 317–322.

Lee, K., and P. A. Fishwick. 1999. OOPM/RT: A multi-modeling methodology for real-time simulation. *ACM Trans. Model. Comput. Simul.* 9 (2): 141–170.

Norvilas, A., A. Negiz, J. DeCicco, and A. Hinar. 2000. Intelligent process monitoring by interfacing knowledge-based systems and multivariate statistical monitoring. *Journal of Process Control* 10 (4): 341–350.

NSF 2005. Dynamic Data Driven Applications Systems. *Available online via http://www.cise.nsf.gov/dddas [accessed June 1, 2005]*.

OMG 2001a. OMG Unified Modeling Language Specification, version 1.5. *Available online via http://www.omg.org/docs/formal/03-03-01.pdf [accessed June 1, 2005]*.

OMG 2001b. UML Profile for Schedulability, Performance, and Time Specification, version 1.0. *Available online via http://www.omg.org/docs/formal/03-09-01.pdf [accessed June 1, 2005]*.

Rumbaugh, J., I. Jacobson, and G. Booch. 1999. *The Unified Modeling Language reference manual*. Essex, UK: Addison-Wesley Longman Ltd.

Tyreus, B. D. 1997. Interactive, dynamic simulation using extrapolation methods. *Computers & Chemical Engineering* 21 (Supplement 1): S173–S179.