A Formal UML Package for Specifying Real-Time System Constraints

Gopal Raghavan and Maria M. Larrondo-Petrie
Department of Computer Science & Engineering
Florida Atlantic University
Boca Raton, FL-33431-0991, USA
{raghavag,maria}@cse.fau.edu

September 25, 1999

Abstract

An object-oriented approach provides a natural way to model a real-time system characterized by timing, resource and precedence constraints. Multiple simultaneous processes competing for resources and constrained by strict timing deadlines add to the complexity of modeling such systems. Unified Modeling Language (UML) is a very general language that supports powerful extension mechanisms that includes: stereotypes, tagged values and constraints. In this paper, we use these extension mechanisms to model real-time system constructs. The constructs are then formalized using Object Constraint Language(OCL) and made available in the form of a UML package. The behavior models were simulated and formally validated for correctness. Such constructs are then stereotyped for use as design patterns when developing real-time applications.

Keywords: Real-Time Systems, Formal Methods, Object-Orientation, Unified Modeling Language, Object Constraint Language.

1 Introduction

Real-time systems host multiple simultaneous processes that are triggered by unpredictable events and are competing for resources. In most cases timing is a very critical factor and it is absolutely necessary to complete such tasks in a timely fashion. Based on the severity of impact the system faces on missing timing deadlines, real-time systems are classified into two categories: soft real-time system and hard real-time system. In soft real-time systems tasks are expected to meet the deadline, but skipping a deadline would not result in severe damage to the system. On the otherhand, missing deadlines could be catastrophic in the case of hard real-time systems. In either case the complexity of building and analyzing the model is mainly due to a tight dependency that exists between the functional and non-functional properties of the real-time systems. Most traditional methods for developing real-time systems are very efficient in modeling these properties in isolation, but fail to explicitly capture their interdependency. The novelty of our approach is in bridging this gap by providing appropriate constructs

using UML's extension mechanisms. UML is a standard, accepted by the Object Management Group (OMG)[1], for specifying object-oriented systems. UML provides a set of artifacts to model structural, behavioral and architectural aspects of a system [2]. In addition, it supports a powerful extension mechanism which can be used to extend UML in a controlled fashion.

In this paper we propose a UML package composed of real-time constraint elements. The package elements are modeled using UML's extension mechanisms. These constraints are formalized using Object Constraint Language (OCL), which is an artifact of UML, and validated using Telelogic Tau [11] [1]. In Section 2, we define a UML package for real-time systems. The proposed formal representation of time is discussed in Section 3. Section 4 provides a formal representation of periodic and aperiodic processes. A formal representation for resource is specified in Section 5. In Section 6 the formal representation of precedence constraint is discussed. Section 7 discusses our fromal validation process and presents results. Our conclusions and future work is provided in Section 8.

2 UML Package for Real-Time System Constraints

In UML, package is a general purpose mechanism for organizing elements [2]. Packages are identified by a simple name or a path name. A simple name provides just a name for the package, while a path name specifies the package name along with the enclosing package name in which this package lives using a scope resolution operator. Packages are composed of other elements with a life-time binding relationship, which means that when a package is destroyed the elements of the package are also destroyed. The visibility of package elements can be specified using appropriate symbols for public (+), private (-) and protected (#) access. Packages are scalable and can be imported or exported. They also follow other structural relationships like generalization and dependency. A package is represented in UML as a tab-folder with package name and containing elements.

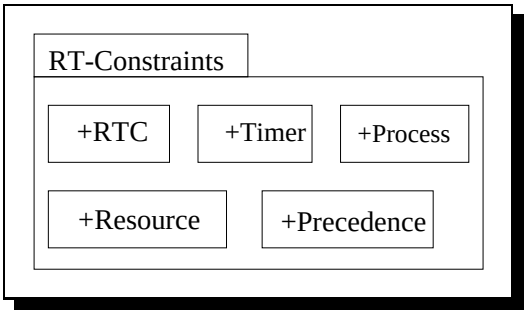

Figure 1: UML package for real-time constraints

For real-time systems we define a package called *RT-Constraints*, which is composed of UML constructs that formally specify real-time constraints. The *RT-Constraints* package is shown in Fig. 1. The elements of this package include: RTC, Timer, Process, Resource and Precedence. We will discuss, in detail, these constraints in the following sections.

3 Formal Representation of Time

In most real-time systems the real-time clock is implemented in hardware and is driven by a crystal clock. The current value of time can be read directly from the free counter register. In addition to the free counter register most hardware support compare registers from which one-shot, fixed-interval and variable-interval timers can be implemented. One shot timers are implemented by loading the register with a value and counting down to zero. An interrupt is produced by the timer to indicate that the count value has reached zero. Fixed-interval timers are implemented by loading the compare register with a value corresponding to the interval. When the free counter register value equals the compare register value an interrupt is produced.

[1]OMG web site http://www.omg.org

A variable-interval timer is similar to the fixed-interval timer except that it supports reloading of the compare register during each period.

Our Real Time Clock(RTC) object is an abstraction of the underlying hardware clock. We assume that the RTC object represents an ideal global real-time clock. The RTC class along with OCL formalism is shown in Fig. 2 [11]. The RTC class has three attributes: *time*, *compare* and *outSig*. The *time* attribute represents the current global value of real-time that is stored in the free counter register. The *compare* attribute stores a relative time value that is stored in the compare register. The *outSig* attribute represents the signals that can be generated by an RTC object. The RTC class has two global interfaces: now() and load(). The method now() is used to access the value of time from any object in the system. The method load() is used to load the compare register with a relative time value.

In our system, time is an absolute, discrete, monotonically increasing quantity represented by an integer. The value of time is incremented by the hardware clock. RTC is a singleton class and will have only one instance in the system. It is used as a reference by all other time related objects in the system. Since clock updates are managed at the hardware level, a software process need not be dedicated to update clock values. In fact, if the RTC object were to be modeled as a process then the scheduler would interleave it with other processes in the system and would cause delays in updating the time value.

All objects in the system can access the RTC via the global method now(). In the RTC, now() will be implemented as a static method. In UML, static methods are differenciated from other methods by explicitly underlining them. The interface will therefore be a class method rather than an instance method. Hence object identifiers are not required to access this method [3][4][7]. The now() method can be accessed using the class name and the scope resolution operator as follows: RTC::now(). The load() method is also implemented in a similar fashion and is globally accessible.

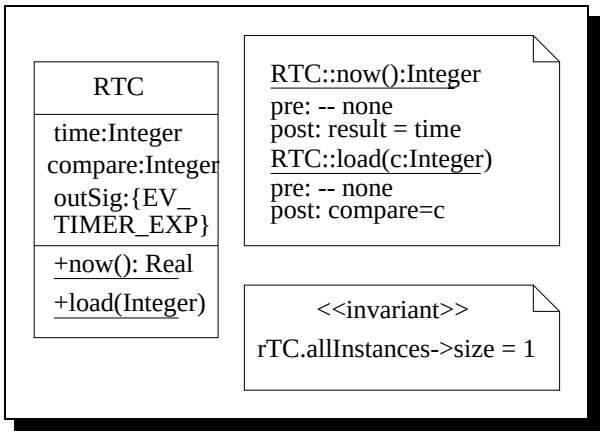

Figure 2: Real Time Clock

We now specify a Timer class, as shown in Fig. 3, that uses the global functions supported by the RTC. The Timer object is very commonly used, in one form or the other, in most real-time systems. The timer keeps track of the start time, end time and the number of timer ticks produced while it is active. The interval between two timer ticks is specified by the attribute *period*. The timer can receive EV_TIMER_EXP as an incoming signal from the hardware indicating expiration of the timer value. Any periodic timer can be easily derived based on this Timer object. For each method supported by the Timer class a formal pre/post condition semantics is specified using OCL notation in Fig. 3. The timing characteristics of the Timer is shown in Fig. 4.

The behavior of the Timer object is shown using the Statechart in Fig. 5. The Timer object is initially in the UNINIT state where the timer variables are initialized. When the start() method is invoked, its state changes to INIT where the *startTime* is initialized to the absolute global value of time using the RTC::now() method. The *endTime* and *tickCount* are also initialized in this state. The RTC is then loaded with the relative time value specified by the attribute period. Once the initializations are complete the Timer waits for the event EV_TIMER_EXP to occur. When this event oc-

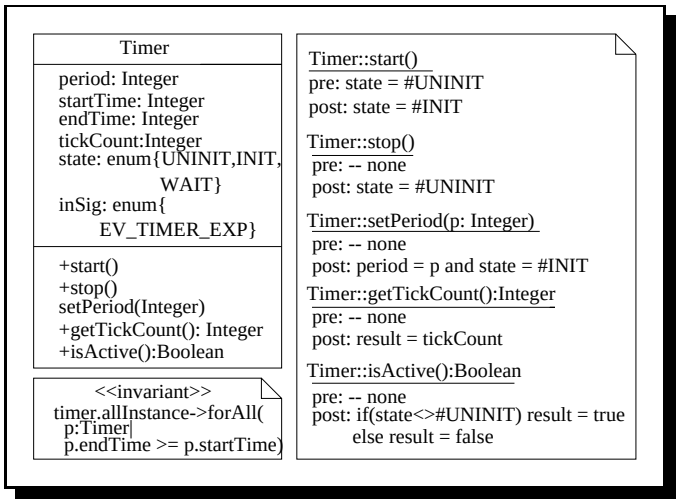

Figure 3: Timer class and OCL formalism for its methods

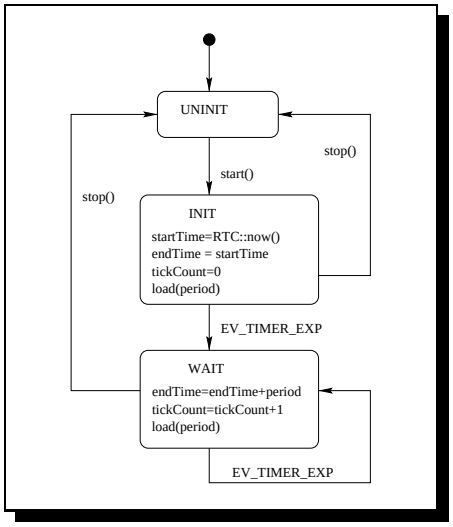

Figure 5: Statechart for Timer class

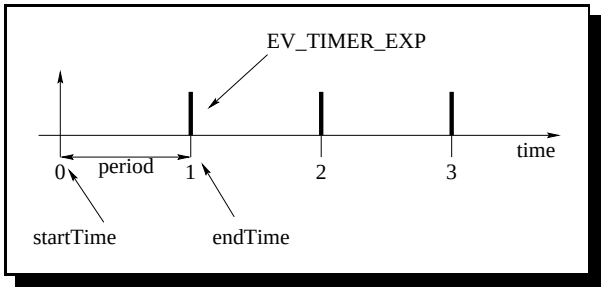

Figure 4: Timing characteristics of the Timer class

curs the object will transition from INIT state to WAIT state. In the WAIT state the *endTime* and *tickCount* are updated. The RTC is then loaded with the relative time value and waits for the event EV_TIMER_EXP to occur. For each occurance of the event the Timer object updates the *endTime* and *tickCount*. When the Timer is in INIT or WAIT state, if the stop() method is invoked the Timer object will transition to UNINIT state.

4 Formal Representation of Process

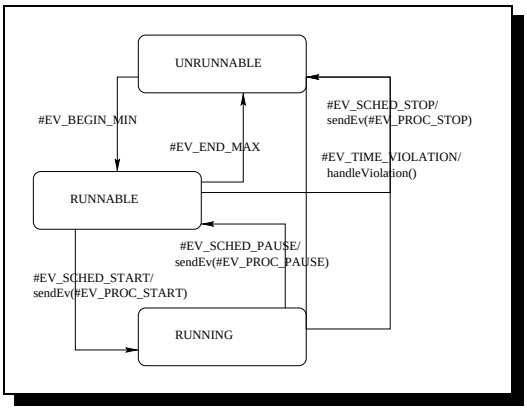

Figure 6: Statechart showing the behavior of periodic process

A process is a set of actions executed in a particular fashion on a processor. Processes are inherently concurrent and such concurrency is simulated on a uniprocessor system by interleaving multiple concurrent processes based on time slicing. Due to stringent memory re-

quirements real-time systems can support only a minimal operating system, known as a *Kernel*, which is capable of scheduling process and handling interrupts. As shown in Fig. 6, a process is state based and is primarily in one of the following three states: UNRUNNABLE, RUNNABLE and RUNNING. When a process is created it is initially in UNRUNNABLE state where the process sleeps and is unschedulable. When an event (EV_BEGIN_MIN) wakes the process up, it goes into RUNNABLE state where it becomes schedulable. The scheduler schedules a process that is in RUNNABLE state by issuing an event EV_SCHED_START, based on time slicing. On receiving this event the process moves to RUNNING state. The process can be paused by the scheduler by issuing an event EV_SCHED_PAUSE. The process will then stop execution and transition to RUNNABLE state. In either the RUNNABLE or RUNNING state, if the process receives EV_SCHED_STOP or EV_TIME_VIOLATION it will stop further execution and transition to UNRUNNABLE state. In a real-time environment, a process needs to meet strict timing and resource constraints. In UML, a process is an active class stereotype whose instances represent heavyweight flow [2]. In the next sections we show how timing constraints can be associated with periodic and aperiodic process.

4.1 Periodic Process

A periodic process usually deals with passive devices. Input/Output (IO) devices such as sensors are considered as passive devices because they require frequent polling in order to determine their status [5]. In many situations, a periodic process can also be used to perform internal functions. Periodic processes are regular and are spaced at equal intervals of time. They are associated with a PeriodicTimer which specifies the timing constraint, as shown in Fig. 7 [9]. We will discuss, in detail, the structure and behavior of a PeriodicTimer in this section since it is closely associated with a periodic process. The periodic process

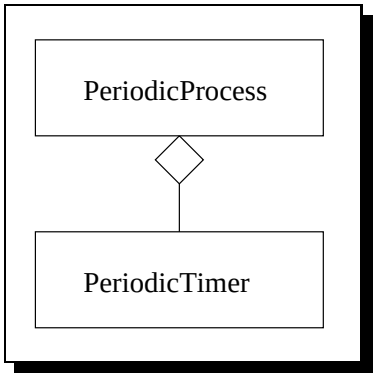

Figure 7: Periodic process associated with a periodic timer

acts as a client to the PeriodicTimer and uses the timing constraints to model the behavior.

The PeriodicTimer class is shown in Fig. 8. The PeriodicTimer is based on the RTC class and makes use of the global methods now() and load(). It keeps track of various critical timing points that occurs during a periodic cycle and generates corresponding events. The PeriodicTimer receives events from RTC and the client process. In turn, the PeriodicTimer sends events out to the client process. The attribute *periodStart* represents the absolute value of period starting time, *periodEnd* represents absolute value of period ending time, *beginMin* represents the absolute minimum time before the process can start, *endMax* represents the absolute maximum time by which the process should complete its task. The attribute *period* represents relative time value of the cycle period, *relBeginMin* represents the relative minimum time after which the process should start and *relEndMax* represents the relative maximum time by which the process should finish. The attribute *inSig* identifies all in-coming events and *outSig* identifies all the out-going events. The PeriodicTimer supports five operations, namely, init(), start(), stop(), recvEv() and sendEv(). The OCL formalization for the class invariant and some of the methods are shown in Fig. 9. The class invariant captures the timing constraint that the Periodic-

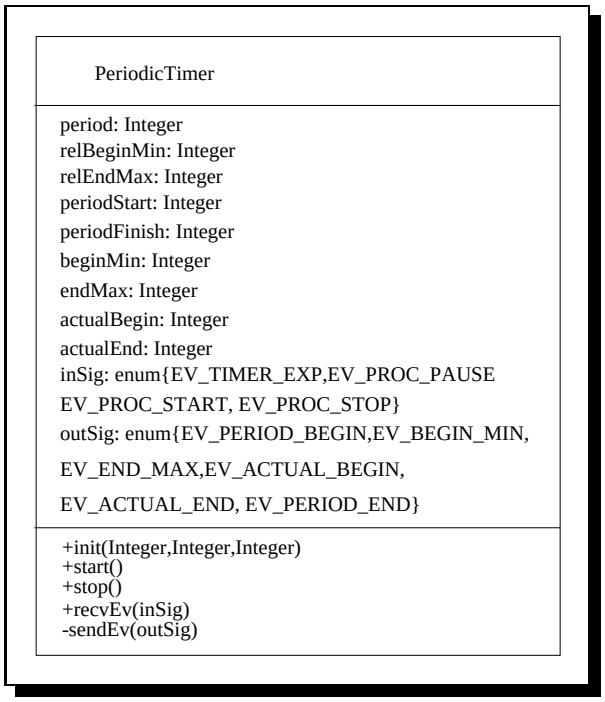

Figure 8: PeriodicTimer class

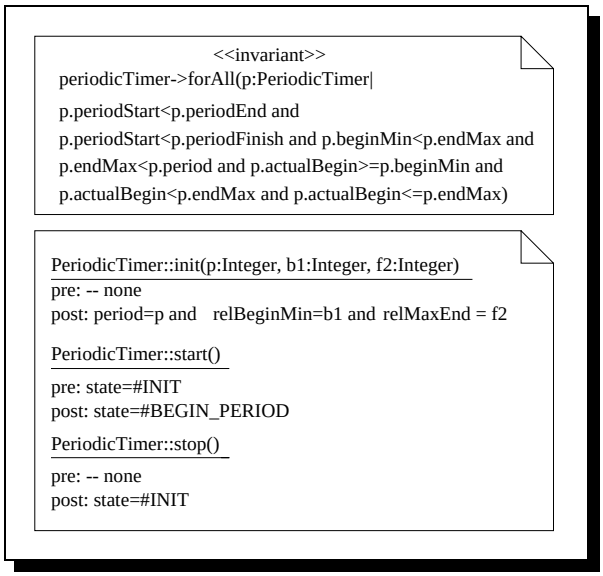

Figure 9: Formalization of PeriodicTimer class invariant and operations using OCL

Timer should follow throughout its lifecycle. The client process communicates with the PeriodicTimer and strictly follows the timing constraint. The timing characteristics of a periodic process is shown in figure Fig. 10. The period of the cycle is given by 'p' and the worst case execution duration is given by 'd'. For a successful schedule the periodic process should always start after 'b1' and finish execution before 'f2'. If the periodic process fails to meet the constraint then it will not be possible to meet the deadline.

The behavior of a PeriodicTimer is as shown in Fig. 11. When initialized the timer goes to INIT state. In this state all the relative time values are initialized with the parameters supplied to the init() function. When the start() method is invoked the timer is started and its internal state transitions from INIT to BEGIN_PERIOD. In this state all absolute time values are updated and the timer moves to the BEGIN_PERIOD state. The RTC is now loaded with a compare register value corresponding to the minimum begin time. The client process is not expected to start before the EV_TIMER_EXP event is received. In case the client process starts execution the the PeriodicTime reports a violation by sending an EV_TIME_VIOLATION event and transitions directly to the END_MAX state, by-passing the intermediate states. On the otherhand, if the timer expires it would mean that the client process has successfully reached the minimum begin time without violating the constraint. The timer will then transition to ACTUAL_BEGIN

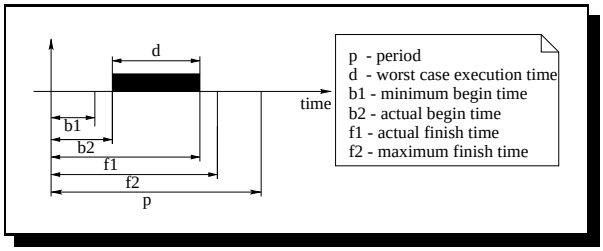

Figure 10: Timing characteristics of periodic process

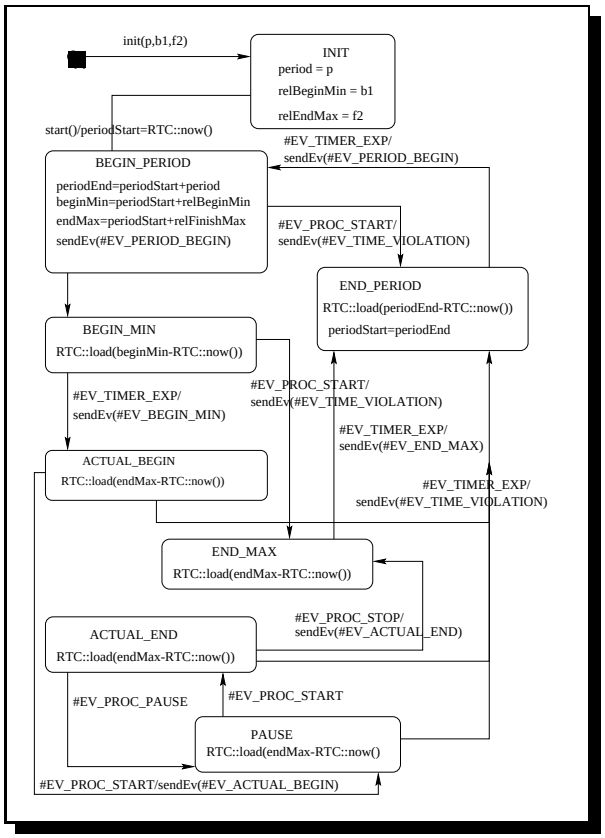

Figure 11: Statechart showing the behavior of the PeriodicTimer

state, which is a valid state for the client process to start execution. If the client process fails to start and the timer expires, the timing constraint is violated. Instead if the client process is started during this period the timer will move to ACTUAL_END state and wait for the client process to issue EV_PROC_STOP event and stop execution. On receiving the EV_PROC_STOP event the timer will move to END_MAX state. But if the timer expires before receiving the EV_PROC_STOP event, then the timing constraint is violated and the timer transitions to END_PERIOD state. The PeriodicTimer also supports pausing during the execution period by transitioning between the PAUSE and ACTUAL_END state. During the pause period, a different process can actually be scheduled by interleaving. In the END_MAX state the RTC is loaded with a time value so as to expire by the maximum finish time. On receiving the EV_TIMER_EXP event, the timer will transition to END_PERIOD state. The RTC is now loaded with a value corresponding to the end of the periodic cycle. This time a timer expiration event will take the timer back to the BEGIN_PERIOD state. This cycle keeps repeating as long as the PeriodicTimer is active. The client process receives all the events that are sent from the PeriodicTimer and behaves accordingly.

4.2 Aperiodic Process

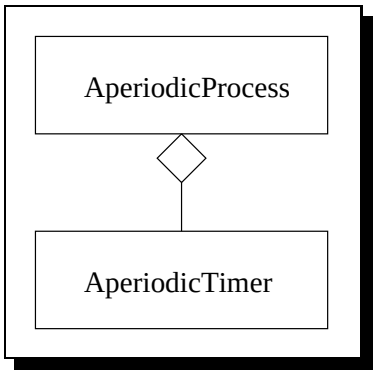

Figure 12: AperiodicProcess associated with an AperiodicTimer

Aperiodic processes are associated with input devices that generate interrupts or with asynchronous internal activities. They do not occur at periodic intervals, but in most cases, aperiodic processes have a minimum time interval between consecutive requests [12]. From a scheduling perspective it is possible to approximate an aperiodic process to behave as a periodic process. Once we make such approximations the analysis procedure that was used for periodic process will hold good for aperiodic processes. Aperiodic process are associated with an aperiodic timer as shown in Fig. 12. In addition, it is possible to buffer the external or internal events that trigger aperiodic process and handle them

periodically [5]. The timing characteristics of an aperiodic process is similar to the one shown if Fig. 10, except that 'p' now represents the minimum time between executions. The state machine behavior of an aperiodic process will be very similar to the one shown in Fig. 6. The behavior of the timer associated with an aperiodic process will however differ from the behavior of the timer associated with a periodic process. The noticible differenc between the two timers is that the AperiodicTimer will not cycle, unlike a PeriodicTimer. The behavior of an AperiodicTimer will look similar to the one shown in Fig. 11. In the case of an AperiodicTimer the occurance of EV_TIMER_EXP event in the END_PERIOD will not result in a state transition to BEGIN_PERIOD. Instead, the timer will come to a halt.

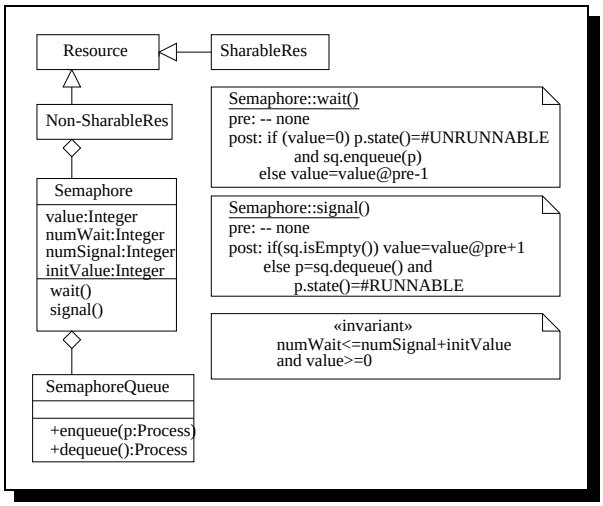

Figure 13: Formal representation of resource constraints using OCL

5 Formal Representation of Resource Constraint

In a real-time system multiple concurrent tasks compete for resources. Tasks are created in the system to achieve a specific job and usually the number of tasks that simultaneously exist will be much larger than the available resources. During its life-time, a task will consume some resources and will release the resources back to the system as soon as its processing is complete. Resources within a system can be classified as sharable and non-sharable. Sharable resources are those that can be used simultaneously by several concurrent processes. Examples of shared resource includes CPU, read-only memory, read-only files, etc. A non-sharable resource on the other hand allows only one process to access it at a time. Example of non-sharable resources include printers, memory, system bus etc.

Since sharable resources allow simultaneous access to concurrent processes there is no need to lock the resource. On the other-hand, non-sharable resources will contain sections of code or operations that need mutually exclusive access. These sections are called critical sections and can be implemented using semaphores. The object model of a semaphore and a formal specification of its operations, wait() and signal(), using OCL are shown in Fig. 13. A semaphore is a non-negative integer, which holds an initial value (*initValue*)that signifies the number of available resources. The value of a semaphore can be changed only via the atomic operations wait() and signal(). The critical section usually begins with a wait and ends with a signal operation on a semaphore. When the wait operation is invoked, it checks to see if the value of the semaphore is zero. If it is zero then the associated process is made UNRUNNABLE and appended to the semaphore queue. The queuing mechanism can be simple FIFO or based on some priority. However, if the semaphore value is not zero then the process is allowed to enter the critical section. During this period the resource will be owned by the process, hence the semaphore value is decremented. Once the processing is complete the signal() operation is invoked on the semaphore to release the resource. The signal() checks the semaphore queue to see if there are any pending processes. If it is found empty then the value of the semaphore is incremented. However, if the

semaphore queue is not empty then the process at the front of the queue is removed and made RUNNABLE. The number of wait() and signal() operations executed on a semaphore is stored in the attributes *numWait* and *numSignal* respectively. This type of semaphore with primitive operations wait() and signal() was introduced in 1965 by Dijkstra and is supported by most operating system [8][10].

6 Formal Representation of Precedence Constraint

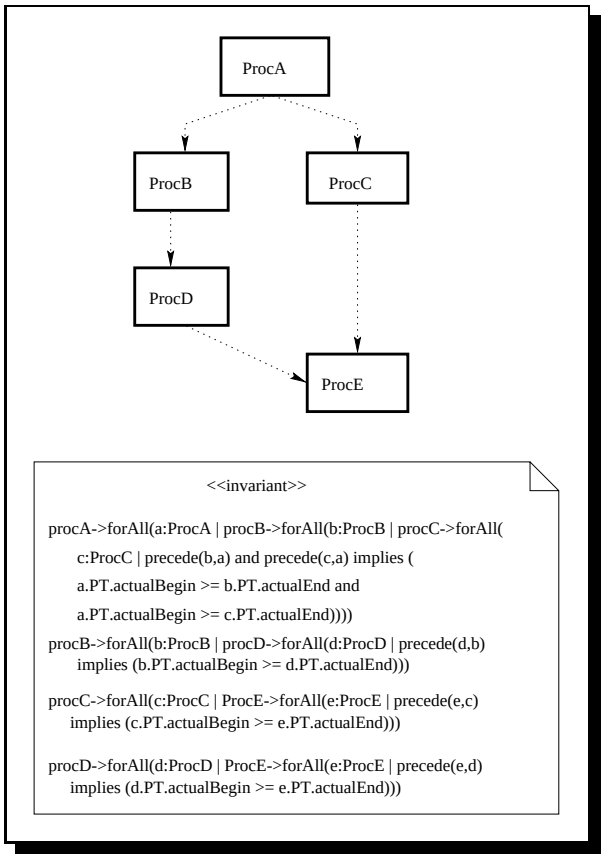

Figure 14: Formal representation of precedence constraints using OCL

Precedence constraints specifies the relationships between a set of processes. This type of relationship can be represented using an acyclic directed graph. A process 'p' is said to *precede* process 'q', if 'q' can start its execution only after 'p' has completed its computation. We define a function precede(p,q) which returns boolean true if p precedes q and boolean false otherwise. In UML, a precedence graph can be represented using dependency relation as shown in Fig. 14, where the rectangular boxes represent a process and the directed, dotted line connecting the boxes represent dependency. In this case, ProcA depends on ProcB and ProcC. ProcB depends on ProcD, which in turn depends on ProcE. Also, ProcC depends on ProcE. Fig. 14 also shows the OCL formalization associated with each precedence relation. In the formalization 'PT' represents the timer associated with the process. We stereotype this flavor of dependency as $\ll$ precedence $\gg$. In a similar fashion other relationships between processes such as, p *excludes* q, p *includes* q, p *overlaps* q etc., can be modeled using a dependency relation and formalized using OCL.

7 Formal Validation of the Package Elements

Validation is a process of making sure that the specifications and models capture the requirements consistently [6]. Formal validation can be conducted in several way based on the designers experience. We have chosen Telelogic Tau [1] as our tool for Formal verification and validation. Tau is a Software Development Tool (SDT) developed by Telelogic AB. Tau supports object oriented requirements capture and analysis. It provides a rich environment for modeling real-time systems using SDL (Specification and Description Language) and MSC (Message Sequence Charts). SDL is an ITU-T standard (recommendation Z.100) for specifying communicating systems. SDL is an extended finite state machine used for developing real-time, interactive systems. Tau provides tools for editing, anaylsing, simulting and validating a system. The timing and process models that we developed as

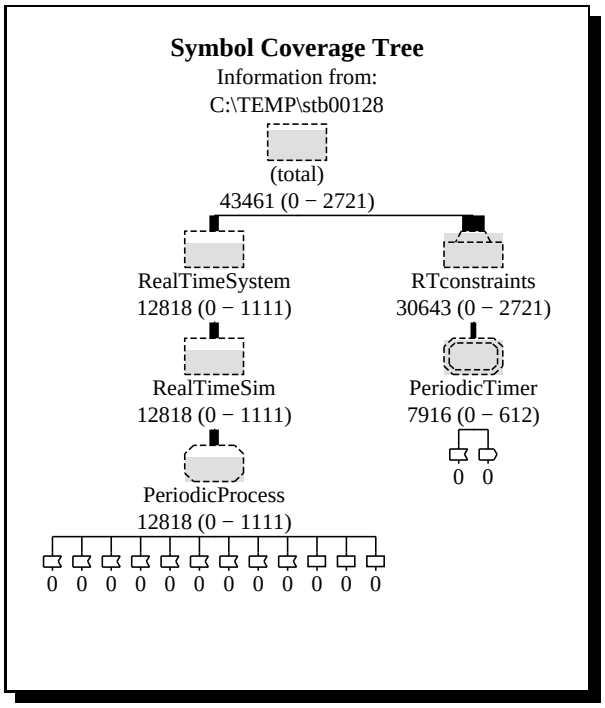

Figure 15: Coverage showing symbols executed least number of times

part of the UML package were translated to SDL models and captured using SDT editor. The SDT editor provides a user friendly environment to graph the SDL model. It also identifies syntax errors during the process of editing. The SDT analyzer was then used to uncover syntactic and semantic errors in the model. Once the model was error free, the simulator was used to further analyze the model. The simulator will allow to step through the model and send signals via channels. The simulator will display the current state of the system and also provide the simulation trace in the form of MSC. For simulation and validation purpose SDL's timer features were used to model the RTC. The SDT validator was then used to conduct a state-space analysis. The validator can perform exhaustive, bit-state and random state exploration. In each case the validator will examine each system state that is encountered and will provide statistics. The validator views the state machine as a behavior tree

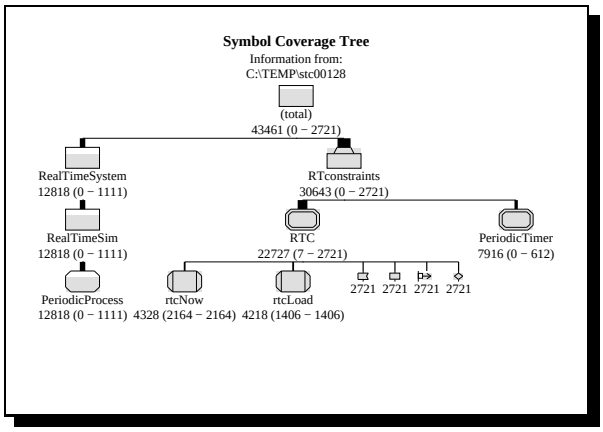

Figure 16: Coverage showing symbols executed most number of times

and traverses every possible path. The tool provides an option to vary the depth of the tree. The results that were obtained after a bit-state exploration and exhaustive exploration are shown below.

```
** Starting bit state exploration **
Search depth    : 400
Hash table size : 1000000 bytes

** Bit state exploration statistics **
No of reports: 0.
Generated states: 2941.
Truncated paths: 13.
Unique system states: 2536.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 4990
Collision risk: 0 %
Max depth: 400
Current depth: -1
Min state size: 152
Max state size: 228
Symbol coverage :  89.79

** Starting exhaustive exploration **
Search depth : 300

** Exhaustive exploration statistics **
No of reports: 0
```

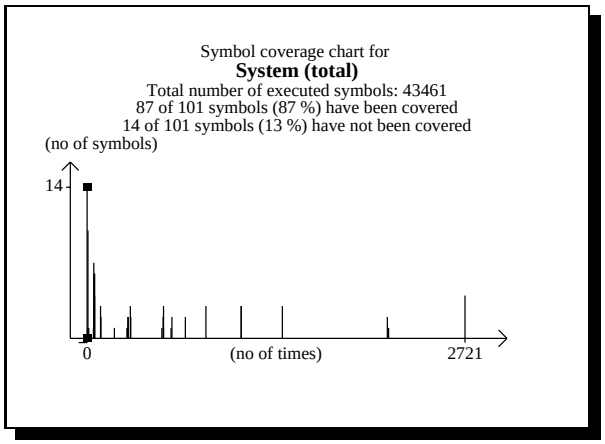

Figure 17: Coverage details

```
Generated states: 2129
Truncated paths: 15.
UnSymbol coverage :  89.79

** Starting exhaustive exploration **
Search depth : 300

** Exhaustive exploration statistics **
No of reports: 0
Generated states: 2129
Truncated paths: 15.
Unique system states: 1837.
Size of hash table: 100000 (400000 bytes)
Current depth: -1
Max depth: 300
Min state size: 152
Max state size: 228
ique system states: 1837.
Size of hash table: 100000 (400000 bytes)
Current depth: -1
Max depth: 300
Min state size: 152
Max state size: 228
Symbol coverage :  89.79
```

The Coverage viewer, a part of Tau suite, was then used to identify the parts of system covered during simulation in terms of the executed transition or symbols. The symobls that were used the least number of times was captured using the coverage viewer and is shown in Fig. 15. Similarly the symbols that were used the most number of times was captured using the coverage viewer and is shown in Fig. 16. Coverage details, including the number of transitions that were executed a certain number of times is shown using a coverage chart in Fig. 17.

8 Conclusion and Future Work

Real-time systems are complex and are subjected to a number of constraints. Object-oriented methods provide a natural way of modeling such systems. Although most real-time system related constructs are not directly handled by UML, it is possible to use the extension mechanism to derive new constructs. In this paper, we have formalized a few real-time system constraints using OCL and packaged them using UML's package. All constructs, that we introduced, are supported with appropriate behavior models and formal specifications. We have used OCL, where necessary, to formalize the operational semantics and constraints. The behavior models were formally validated using Telelogic Tau. The models were simulated both manually and automatically. They were found to be stable and consistent with the specifications. These real-time constraints can be stereotyped and included in the real-time package.

In the future, we plan to formalize and stereotype other aspects of real-time systems such as, scheduling and fault-tolerance. We also plan to feed the OCL specifications through an OCL parser to identify semantic errors [2]. All these stereotypes and associated formalisms will be included in the real-time package. Such packages can be directly imported into the application system and used for development. We believe, this will result in controlled and accelerated development of real-time systems.

[2] OCL web site http://www.software.ibm.com/ad/ocl

References

[1] *Telelogic Tau 3.5: ORCA and SDT Manual.* Telelogic AB, 1999.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Longman, Inc., Reading, Massachusetts, 1999.

[3] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, 1997.

[4] A. Goldberg and D. Robson. *Smalltalk-80 The Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[5] H. Gomma. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[6] C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing: An Overview.* John Wiley & Sons, 1995.

[7] S. B. Lippman and J. Lajoie. *C++ Primer.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[8] A. M. Lister and R. D. Eager. *Fundamentals of Operating Systems.* The Macmillan Press Ltd, Hampshire, London, 1992.

[9] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9), 1992.

[10] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[11] J. Warmer and A. Kleppe. *The Object Constraint Language Precise Modeling with UML.* Addison-Wesley Longman, Inc., Reading, Massachusetts, 1999.

[12] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1), 1993.