Cost Enforcement and Deadline Monitoring in the Real-Time Specification for Java

Andy Wellings, University of York, U.K. {andy@cs.york.ac.uk}
Greg Bollella, Sun MicroSystems {greg.bollella@sun.com}
Peter Dibble, TimeSys Corporation {peter.dibble@timesys.com}
David Holmes, DLTeCH Pty Ltd {dholmes@dltech.com.au}

Abstract

Modern real-time programming languages and operating systems provide support for monitoring the amount of CPU time a thread consumes. However, no system in widespread use fully integrates this monitoring with the scheduling facilities. The Real-Time Specification for Java (RTSJ) provides an integrated approach to scheduling periodic threads and monitoring their CPU execution time. It supports *a cost enforcement model* whereby a periodic thread is suspended when it consumes more time than it requested. Version 1.0 of the RTSJ is under specified and it is difficult to understand the full model. This paper clarifies the position and defines the conditions under which a real-time thread is resumed. The model presented is the one that will be fully defined in version 1.0.1 of the RTSJ. Unfortunately, version 1.0.1 of the specification will not have a general model for handling cost enforcement and deadline monitoring for all schedulable objects. This paper proposes extensions to the RTSJ that allow the cost enforcement model and deadline monitoring model to be consistently applied across all schedulable objects, and for it to be fully integrated with scheduling.

1 Introduction

Real-time systems must be able to interact with their environments in a timely and predictable manner. The days when *real-time* simply meant *fast* have long gone. It is no longer acceptable to build systems and hope that they meet their timing requirement. Instead, designers must engineer analysable systems whose timing properties can be predicted and mathematically proven correct (possibly from within a probabilistic framework). Advances in scheduling have brought about this change in practice.

Scheduling is the ordering of thread/process executions so that the underlying hardware resources (processors, networks, etc.) and software resources (shared data objects) are efficiently and predictably used. In general, scheduling consists of three components [3]

- an algorithm for ordering access to resources (scheduling policy)
- an algorithm for allocating the resources (scheduling mechanism)
- a means of predicting the worst-case behaviour of the system when the policy and mechanism are applied (schedulability analysis — called feasibility analysis by the Real-Time Specification for Java (RTSJ) [2]).

Once the worst-case behaviour of the system has been predicted, it can be compared with the system's timing requirements to ensure that all deadlines will be met.

There have been many different scheduling approaches developed over the last 10-15 years [3], for example, cyclic scheduling, fixed priority scheduling, earliest deadline first, value-based, etc. Most current real-time programming languages and operating systems in widespread use today support fixed priority scheduling. Some provide support for monitoring the amount of CPU time a thread consumes; a few allow the programmer to undertake actions if the amount of CPU time a thread consumes exceeds a defined value. However, no system fully integrates the monitoring with the scheduling facilities. *Consequently, the system cannot ensure that the analysis that has been performed to guarantee the application's deadline will not be undermined by the run-time execution of the program.* The RTSJ is the first main stream real-time programming language that attempts to fully integrate its scheduling with the CPU execution time monitoring facility.

Unfortunately, version 1.0 of the RTSJ is under specified and it difficult to understand the full cost enforcement and deadline monitoring model. The initial aim of this paper is to clarify the position and define the conditions when a periodic real-time thread is descheduled due to a cost overrun and describe the mechanisms that allow the program to respond. The model presented is the one that will be fully defined in version 1.0.1 of the RTSJ. However, version 1.0.1 will not have a general model for handling cost enforcement and deadline monitoring for all schedulable objects. Such a change is beyond the allowable changes for a 1.0.1 release. This paper proposes extensions to the version 1.0.1 (in particular, the `Schedulable` interface and the `RealtimeThread` and `AsyncEventHandler` classes) that allow the cost enforcement model and deadline monitoring model to be consistently applied across all schedulable object, and for it to be fully integrated with scheduling. The goal is to begin the discussions for a future release of the RTSJ with more substantial changes.

The paper is structured as follows. The remainder of this section characterizes the properties of a schedulable

object that must be made available to any real-time scheduler that wants to provide guarantees (either offline or online). As such, it provides a rationale for the overall RTSJ model that is summarised in Section 2. Section 3 then focuses on the cost enforcement and deadline-monitoring model for real-time threads and asynchronous event handlers, indicating the support provided by version 1.0.1 of the specification. Section 4 proposes extensions to the model. Section 5, addresses the problem of schedulable objects with aperiodic release parameters. Section 6 discusses related work. Finally, section 7 presents the conclusions.

2.1 Scheduling Model

Most modern approaches to scheduling view the system as consisting of a number of real-time threads. Each thread is characterized by the following properties [11].

Release profile. Typically after a thread is *started*, it waits to be *released* (or may be released immediately); when released, it performs some computation and then waits to be released again (the time at which it waits is often called its *completion time*). The release profile defines the frequency with which the releases occur; they may be *time triggered* or *event triggered*. When time triggered releases occur on a regular basis, they are called *periodic* releases. Event-triggered releases are typically classified into *sporadic* (meaning that they are irregular but with a minimum inter-arrival time) or *aperiodic* (meaning that no minimum inter-arrival assumptions can be made) - of course, event-triggered periodic releases and time-triggered sporadic and aperiodic release are also possible. Once a schedulable object has been released, it is eligible for execution. During its execution, it may be blocked waiting for a resource (for example, a mutual exclusion lock). When the resource becomes available, the thread is again eligible for execution.

Processing cost per release. This is some measure of how much of the processor's time is required to execute the computation associated with the thread's release (this may be a worst-case value or an average value depending on the feasibility analysis being used). It is often referred to as a CPU budget.

Other hardware resources required per release This is some measure of the hardware resources needed (other than the processor). For networks, it is usually the time needed (or bandwidth required) to send the thread's messages across the network. For memory, it is the amount and type of memory required by the thread.

Software resources required per release. This is a list of the non-shareable resources that are required for each release of the thread and the processing cost of using each resource. Access to non-shareable resources is a critical factor when performing schedulability analysis. This is because non-shareable resources are usually non pre-emptible. Consequently, when a thread tries to acquire a resource, it may be blocked if that resource is already in use. This blocking time has to be taken into account in any analysis. If the list of software resources is not available then a maximum blocking time must be provided.

Deadline The time that the thread has to complete the computation associated with each release. Where the deadline of a thread is greater than its minimum period between releases (or it has overrun its deadline, and the application has decided to let it continue), the thread may be released even though the execution associated with the previous release has not completed. In this case, when the thread does complete, it is immediately re-scheduled for execution (re-released).

Value A metric that indicates the thread's contribution to the overall functionality of the application. It may be: a very coarse indication (such as safety critical, mission critical, non critical), a numeric value giving a measure for a successful meeting of a deadline, or a time-valued function which takes the time at which the thread completes and returns a measure of the value (for those systems where there is no fixed deadline).

One of the key characteristics of schedulability (feasibility) analysis is whether the analysis is performed off-line or on-line. For safety critical systems, where the deadlines associated with schedulable objects must always be met (so-called *hard* real-time systems), off-line analysis is essential, as the system must not enter service if there is a possibility of deadlines being missed. Other systems do not have such stringent timing requirements or do not have a predictable worst-case behavior. In these cases, on-line analysis may be the only option available. These systems must be able to tolerate threads not being feasible (that is, failing the schedulability analysis) and offer degraded services. Furthermore, they must be able to handle deadlines being missed or situations where the assumed worst-case loading scenario has been violated.

2 Schedulable Objects and Scheduling in the RTSJ

The RTSJ provides a framework from within which on-line feasibility analysis of priority-based systems can be performed for single processor systems. The specification also allows the real-time JVM to monitor the processing resources being used and to release asynchronous event handlers if this use of resources goes beyond that specified by the programmer.

The RTSJ incorporates the notion of a *schedulable object* rather than considering just threads. A schedulable object is any object that implements the `Schedulable` interface. The current specification supports only two types of object that implement this interface, `RealtimeThread`s and `AsyncEventHandler`s.

They have the following associated attributes (represented by classes).

`ReleaseParameters` — Giving the processing cost for each release (its CPU budget) and the deadline; if the object is released periodically or sporadically then subclasses allow an interval to be given. (For sporadic release parameters, the RTSJ provides facilities to ensure that the associated schedulable objects are not released more often than the minimum inter-arrival time.) Event handlers can be specified for the situation where the deadline is missed or the processing resource consumed becomes greater than the cost specified. *Note, there is no requirement for a real-time JVM to monitor the processing time consumed by a schedulable object. If it does, then there is a requirement that a schedulable object be given no more than cost processing units each release (see Section 3).* Note also, the RTSJ makes no mention of blocking time in any of the parameters associated with schedulable objects. The assumption is that a particular implementation will subclass `ReleaseParameters` to bring in this data.

`SchedulingParameters` — This class is abstract; however subclasses allow the priority of the object to be specified along with its importance to the overall functioning of the application. Although the RTSJ specifies a minimum range of real-time priorities (28), it makes no statement on the allowed values of the importance parameter. Indeed, the default priority scheduler is not required to use importance in any of its scheduling decisions.

`MemoryParameters` — Giving the maximum amount of memory used by the object in its default memory area, the maximum amount of memory used in immortal memory, and a maximum allocation rate of heap memory. An implementation of the RTSJ is obliged to enforce these maximums and throw exceptions if they are violated.

`ProcessingGroupParameters` — Allowing several schedulable objects to be treated as a group (which has an associated period, cost and deadline, see Section 5).

The methods in the `Schedulable` interface can be divided into three groups.

- Methods that will communicate with the scheduler and will result in the scheduler either adding or removing the schedulable object from the list of objects on which it performs schedulability analysis (called its *feasibility set*), or changing the parameters associated with the schedulable object (but only if the resulting system is feasible).
- Methods that get or set the parameter classes associated with the schedulable object. If the parameter object being set is different from the one currently associated with the schedulable object, the previous value is lost and the new one will be used in any *future* feasibility analysis performed by the scheduler. Note, these methods do not result in feasibility analysis being performed and the parameters are changed even if the resulting system is not feasible.
- Methods that get or set the scheduler. For systems that support more than one scheduler, these methods allow the scheduler associated with the schedulable object to be manipulated.

Changing the parameters of a schedulable object whilst it is executing can potentially undermine any feasibility analysis that has been performed, and cause deadlines to be missed. Consequently, the RTSJ provides methods that allow changes of parameters to occur only if the new set of schedulable objects is feasible. Some parameter changes take place immediately (for example, priority and cost changes), others do not have an impact on a schedulable object's executions *until its next release (for example, changes to deadline or period). In all cases, the scheduler's feasibility set is updated.* Of course, an infeasible system may still meet all its deadlines if the worst-case loading is not experienced (perhaps the worst-case phasing between the threads does not occur, or threads do not run to the worst-case execution time).

The only scheduler that the RTSJ fully defines is a priority scheduler, which can be summarized as follows.

Scheduling policy. The PriorityScheduler

- supports the notion of **base** and **active** priority – with at least 28 unique real-time priorities;
- orders the execution of schedulable objects on a single processor according to the active priority;
- allows the programmer to assign the base priorities at run time;
- supports priority inheritance or the priority ceiling emulation protocol for synchronized objects;
- assigns the active priority of a schedulable object to be the higher of its base priority and any priority it has inherited.

Scheduling mechanism. The PriorityScheduler

- supports pre-emptive priority-based dispatching of schedulable objects;
- does not define where in the run queue (associated with the priority level) a pre-empted object is placed; the RTSJ recommends that it be placed at the front of the queue;
- places a blocked schedulable object that becomes runnable, or has its base priority changed, at the back of the run queue associated with its (new) priority;
- does not define whether schedulable objects of the same priority are scheduled in FIFO, round-robin order or any other order.

Schedulability analysis. The PriorityScheduler requires no particular analysis to be supported.

3 Cost Enforcement and Deadline Monitoring in Version 1.0.1 of RTSJ

As mentioned in Section 2, every schedulable object has a release profile (periodic, sporadic or aperiodic) and for each release it is given a CPU budget of "cost" and a deadline. However, the details of the cost enforcement and deadline monitoring models are not well defined in Version 1.0 of the RTSJ. This section clarifies the model. Section 4 proposes extensions to allow the model to be consistently applied across all schedulable objects.

In Version 1.0.1, the cost enforcement and deadline monitoring model applies to periodic real-time threads whose deadlines are less than their period and to asynchronous event handlers. In both cases, event handlers may be released when cost overruns and deadline misses occur. However, the response of the system when handlers are not released is different.

3.1 Periodic Real-Time Threads

Each thread can define (but need not do so) deadline-miss and cost-overrun asynchronous event handlers. An implementation must support deadline monitoring, and it is required that any deadline-miss handler be released at the point its associated real-time thread misses its deadline (note, however, the real-time thread remains executable). Cost overrun detection is optional. If cost monitoring is supported, the RTSJ requires that the priority scheduler gives a schedulable object a CPU budget of no more than its cost value on each release. Hence, if a periodic real-time thread overruns its cost budget, it is automatically suspended. *It will not be resumed until either its next release occurs (in which case its budget is replenished) or its associated cost value is increased.*

The `RealtimeThread` class has the following methods to support the overall model:

- `waitForNextPeriod` – when called by a periodic real-time thread (and in the absence of any deadline miss), the thread is blocked until its next release, when the method returns true;
- `deschedulePeriodic` – when called, the real-time thread is descheduled when *it finishes its current release* (i.e. calls `waitForNextPeriod`);
- `schedulePeriodic` – when called, if the real-time thread is currently descheduled in `waitForNext-Period`, it is re-scheduled when its next release occur (if not already descheduled, `schedulePeriodic` cancels any outstanding `deschedulePeriodic` requests).

If the programmer has set up an appropriate deadline-miss handler, the RTSJ assumes that the handler will take some corrective action on a deadline miss and then (if appropriate) reschedule the real-time thread by calling the `schedulePeriodic` method. If no call to the `schedulePeriodic` method has been made in between the deadline miss and the call to `waitForNextPeriod`, the method automatically deschedules the real-time thread (a call to `waitForNextPeriod` signals the end of the current release). It remains descheduled until a call to `schedulePeriodic` is made. In this situation, the `waitForNextPeriod` method returns at the next release time *following* the call to `schedulePeriodic`. All releases in between are lost. A descheduled thread, by definition, cannot miss any further deadlines.

If the programmer has not set up the appropriate handler, the `waitForNextPeriod` method assumes that the real-time thread itself will undertake some corrective action and then call `waitForNextPeriod` again. Hence, the method returns false immediately indicating that the current release is still active and the real-time thread should respond to the deadline-miss condition.

The behavior of a real-time thread with periodic release parameters can be defined in terms of four *private* variables added to the real-time thread state, which are not accessible to the programmer (here it is assumed that no changes to the release parameters occurs):

- `boolean lastReturn` — the last value returned by `waitForNextPeriod`,
- `integer missCount` — the number of deadlines that have been missed and that have not been communicated to the application (by releasing a deadline-miss handler)
- `boolean descheduled` — when true, results in the thread being descheduled at the end of its current release (it will not receive any further release events and no deadlines can be missed)
- `integer pendingReleases` — indicates the number of outstanding release events (periods) that haven't been acted upon.

The semantics of these values are detailed below. The approach is to consider each significant event in the execution of an RTSJ program that manipulates them.

1. **On each deadline miss**: if the associated real-time thread has a deadline-miss handler, the value of `descheduled` is set to true, the deadline miss handler is released and the `missCount` is set to zero[1]. If the real-time thread does not have a deadline-miss handler, one is added to its `missCount` value.

[1] Any outstanding deadline misses not acted upon by the program will be passed to the asynchronous event handler (AEH) at this time (via the AEH's `fireCount` – see section 3.2), hence `missCount` is set to zero.

2. **On each cost overrun**: the execution of the real-time thread is suspended (its new state is blocked-for-cost-replenishment) - any overrun handler is released.
3. **At the start of each period**: If the thread is waiting for its next release (that is, it is blocked-for-reschedule in `waitForNextPeriod`) and it is descheduled, no action is taken. Otherwise, the thread's `pendingReleases` value is incremented. If the thread is waiting for its next release, it is notified. If the thread is not eligible for execution because it is blocked-for-cost-replenishment, its cost budget is replenished and it is made eligible for execution.
4. **When the thread's `schedulePeriodic` method is invoked**: the value of `descheduled` is set to false. If the thread is blocked-for-reschedule in `waitForNextPeriod`, the value of `pendingReleases` is set to zero and the thread is notified.
5. **When the thread's `deschedulePeriodic` method is invoked**: the value of `descheduled` is set to true.
6. **When the thread's cost parameter changes**: if the change causes the thread's cost budget to be depleted and the thread is currently eligible for execution, a cost overrun is triggered for the thread (see 2. above), otherwise if the change causes the cost budget not to be depleted and the thread is currently blocked for cost replenishment, the thread is made eligible for execution.
7. **When the `waitForNextPeriod` method is called**: there are two possible behaviors depending on the state of `missCount` and `descheduled`:
   - If `missCount` is greater than zero: the `missCount` value is decremented. If the `lastReturn` value is false, `pending-Releases` is decremented and false is returned. If the `lastReturn` value is true, it is set to false and false is returned.
   - Otherwise, the method waits while the `descheduled` value is true or `pendingReleases` is zero, then `pending-Releases` is decremented and the `lastReturn` value to set to true and the method returns true.

A return of false from the `waitForNextPeriod` method indicates that the current release has missed its deadline. A second return of false indicates that not only did the current release miss its deadline, but also the next release has already occurred and the deadline for that release has been missed.

3.2 Asynchronous Event Handlers

The RTSJ views asynchronous events as data-less occurrences that are either fired by the program or associated with the triggering of interrupts (or signals or other asynchronous events) in the environment. One or more handlers can be associated with a single event, and a single handler can be associated with one or more events. The association between handlers and events is dynamic. Each handler has a count (called `fireCount`) of the number of outstanding occurrences. When an event occurs, the count is atomically incremented. The attached handlers are then released for execution. Recall from section 2, a schedulable object can be released even though it has not completed the execution associated with its previous releases. The fire count caters for this situation, allowing the implementation to start the execution associated with the new release immediately the old releases have finished.

The release of each asynchronous event handler (AEH) has an associated deadline and cost budget. As with real-time threads, other event handlers can be released if deadline misses or cost overruns occurs. Cost overruns result in the errant AEH being automatically suspended until either its cost parameter is increased, or it is released again.

To support optimizations of event handlers, the RTSJ allows an AEH to manipulate its `fireCount` value. These methods do not have an impact on either the current deadline or the current cost budget

4 Generalising the RTSJ Model

Version 1.0.1 of the RTSJ provides a much more detailed explanation of its cost enforcement and deadline monitoring model. However there are two weaknesses in the approach that require more significant changes to the specification. These changes go beyond the removal of ambiguity and errors that are normally associated with a minor re-release of the specification. These weaknesses are:

- the support for sporadic and aperiodic real-time threads is not adequate to allow multiple releases and the detection of a deadline miss or a cost overrun; furthermore, sporadic and aperiodic threads cannot be descheduled ;
- the support for asynchronous events does not allow a handler to recover from a deadline miss other than by releasing another asynchronous event handler — this is not consistent with the model provided for periodic real-time threads; furthermore, asynchronous event handlers cannot be descheduled.

This section considers how the facilities supported by version 1.0.1 of the RTSJ can be generalized to provide consistent support for all schedulable objects and whose deadlines can be less than, equal to, or greater than the time between releases. It should be stressed that this

proposal is not sanctioned or supported by any Java JSR. The goal is to help stimulate discussion in the community on the functionality of future RTSJ releases.

In order to generalize the facilities provided by the RTSJ and provide a consistent set of mechanisms for all schedulable objects it is necessary to augment:

- the `Schedulable` interface with facilities to allow a schedulable object to be included/excluded from the set of schedulable objects currently being considered for execution;
- the `RealtimeThread` class with a general release mechanism;
- the `AsyncEventHandler` class with a mechanism to handle deadline misses when no deadline miss handler has been specified.

4.1 An Extended `Schedulable` Interface

Currently the RTSJ provides a mechanism that allows a periodic real-time thread to be removed from the current group of objects that are eligible for scheduling (`deschedulePeriodic`). The "suspension" is not like the normal asynchronous suspend operation that has been deprecated in standard Java. It is a much safer real-time equivalent, as the "suspension" occurs at the end of the current release. The corresponding "resume" mechanism is provided by `schedulePeriodic`. This paper proposes that these mechanisms should be generalized and made available to all schedulable objects and, consequently, should be provided in the `Schedulable` interface:

- `deschedule` – when called, the schedulable object is descheduled (made not eligible for release) when it finishes its current release;
- `schedule` – when called, if the schedulable object is currently descheduled, it becomes eligible for release; it next executes when its next release occurs.

All releases that occur when a schedulable object is descheduled are lost and, by definition, no deadlines are missed.

4.2 An Extended `RealtimeThread` Class

As well as providing methods to implement `deschedule` and `schedule` (the new methods in the `Schedulable` interface), the `RealtimeThread` class should provide:

- `waitForNextRelease` – when called by a real-time thread, in the absence of any deadline-miss condition, the thread is blocked until its next release occurs;
- `release` – when called, this indicates that the real-time thread should be released.

With this proposal, the current `waitForNextPeriod` method would be synonymous with `waitForNextRelease`; `deschedulePeriodic` synonymous with `deschedule` and `schedule-Periodic` synonymous with `schedule`. The scheduler will usually be responsible for releasing a real-time thread with periodic release parameters. However, the application can force a release by calling the `release` method explicitly. Such a facility might be useful during mode changes; though care must be taken not to undermine any feasibility analysis.

4.3 An Extended `AsyncEventHandler` Class

As well as providing methods to implement `deschedule` and `schedule`, this extended class should provide a `deadlineMissCondition` — this method would be called by the system (*at the end of the current release*) if an event handler missed a deadline and has no deadline miss handler specified. This would allow an event handler to provide a response in a similar manner to the "false return" from the `waitForNextRelease` method in the `RealtimeThread` class.

4.4 A Consistent Model of Deadline Monitoring and Cost Enforcement for all Schedulable Objects

Given the above mechanisms, it is now possible to define a consistent integrated model for deadline monitoring and cost enforcement.

For an arbitrary release, *j*, of a schedulable object, the CPU budget is automatically fully replenished to `cost` units when

- release j-1 has completed (or release j-1 has been suspended as a result of a cost overrun), *and*
- release j has occurred.

Hence,

- if the completion time (or the suspension time) is less than or equal to the time between releases, the CPU budget is replenished at the time of release j;
- if the completion time (or suspension time) is greater than the time between releases, the CPU budget is replenished at the time of completion (or suspension) of release j-1.

The completion point for a real-time thread is when it calls the `waitForNextRelease` method. For an asynchronous event handler, the completion point is when it returns from the `handleAsyncEvent` method.

A cost overrun results in the schedulable object being *immediately* automatically suspended - made not eligible for execution - and any cost overrun handler released (assuming cost enforcement is supported). The schedulable object will not be resumed until either

- its next release occurs (or has occurred, in which case its CPU budget is automatically replenished with the `cost` value) or
- its associated `cost` value is increased.

Note, that the release of a schedulable object with sporadic release parameters may be subject to a delay in order to satisfy any minimum inter-arrival time restrictions.

If the schedulable object misses its deadline, it remains eligible for execution (unless it has also suffered a cost overrun). Any associated deadline-miss handler is released at the point the deadline expires. If there is no associated handler, a count is kept of the number of missed deadlines.

When a schedulable object has missed its deadline and indicates that it has completed its current release (called the `waitForNextRelease` method for a real-time thread, and returned from the `handleAsyncEvent` method for an event handler), the system undertakes the following:

- if a deadline has been missed and there was an associated deadline-miss handler, the schedulable object is descheduled *unless* the `schedule` method has been called since the last deadline miss; if it is descheduled, it will remained descheduled until `schedule` is called;
- if there is no deadline miss handler, the `deadlineMiss` count is decremented, the system then requests that the schedulable object provide a recovery mechanism; for a real-time thread, it does this by `waitForNextRelease` returning false immediately — subsequent calls to `waitForNextRelease` return false if the count is greater than 0 and there has been no subsequent call to `schedule` since the last deadline miss. For an event handler, the `deadlineMissCondition` method is called immediately the `handleAsyncEvent` returns — the method is called again if the count remains above zero and no call to `schedule` has occurred since the last deadline miss.

5 Handling Aperiodic Activities

In any system where it is required to give guarantees, aperiodic schedulable objects present a problem. As they have no well-defined release characteristics, they can impose an unbounded demand on the processor's time.

To support aperiodic activities, a **server** can be employed. Servers protect the processing resources needed by periodic and sporadic schedulable objects but otherwise allow aperiodic schedulable objects to run as soon as possible. The real time community has defined several types of servers. The one that is most relevant to the RTSJ is a **deferrable server** [7]. With the deferrable server, an analysis is undertaken that enables a new thread to be introduced at a particular priority level. This thread, the server, has a period and a capacity. These values can be chosen so that all the periodic and sporadic schedulable objects in the system remain schedulable even if the server executes periodically and consumes its capacity. At run-time, whenever an aperiodic thread is released, and there is capacity available, it starts executing at the server's priority level until either it finishes or the capacity is exhausted. In the latter case, the aperiodic thread is suspended (or transferred to a background priority). With the deferrable server model, the capacity is replenished every period.

The RTSJ provides support for aperiodic server technologies via processing group parameters. When processing group parameters are assigned to one or more schedulable object, a server is effectively created. The server's start time, cost (capacity) and period is defined by the particular instance of the parameters. These collectively define the points in time when the server's capacity is replenished.

Any aperiodic schedulable object that belongs to a processing group is executed at the schedulable object's defined priority. However, it only executes if the server still has capacity (and it has not overrun its own individual CPU budget). As it executes, each unit of CPU time consumed is subtracted from the server's capacity (as well as its own). When capacity is exhausted, the aperiodic threads are not allowed to execute until the start of the next replenishment period. If the application only assigns aperiodic schedulable objects of the same priority level to a single `ProcessingGroupParameters` object, then the functionality of a deferrable server can be obtained [4].

6 Related Work

Over the last 10-15 years, there has been a gradual migration from sequential real-time systems based on cyclic scheduling to concurrent real-time systems based on priority-based scheduling. In cyclic executives, overrun conditions are automatically caught if one minor cycle is still executing when the next is due. However, industrial-strength modern languages and operating systems have been slow to realize the importance of monitoring the amount of CPU time consumed by a concurrent entity. Any support that has been provided has been in the context of profiling an application looking for hot spots.

The problems are further compounded by the lack of support provided for release profiles for concurrent entities. For example Ada 95 [1], CHILL [5] and the real-time extensions to POSIX [10] do not support the explicit specification of periodic or sporadic processes with deadlines; rather a delay primitive, timer and so on, must be used within a looping construct (although POSIX does allow a periodic timer to be set). Any deadline overrun detection mechanisms must similarly be programmed with low-level mechanisms such as the "select then abort" statement in Ada or via watchdog timers in real-time POSIX.

The notable exception to this "low level mechanism" approach occurs in the German industrial control language PEARL [9] which provides explicit timing information concerning the start, frequency and termination of processes. Some research-oriented languages have also taken this approach (for example, Real-Time Euclid [6]).

The situation with CPU budgets and cost enforcement nowadays is more encouraging. The POSIX community has led the way by supporting execution time monitoring by extending its clock and timer facilities to include CPU-time clocks for processes and threads. Each process/thread has an associated execution-time clock whose current value can be queried or set. The standard POSIX timers can be used to create timers that generate process signals when the execution time set has expired. However, it has no facilities other than setjmp and longjmp to provide controlled termination of the errant thread.

In the current revision process of Ada (for Ada 2005) it is likely that facilities similar to those provided by POSIX will be added. In particular, a new clock is proposed which measures CPU execution time. A timer can be constructed via an Ada protected type. This allows the mechanisms to be linked into other Ada facilities, such as the “select then abort” statement, to provide various models [8]. However, scheduling and CPU monitoring has not been fully integrated in the way that it is with the cyclic executive approach and with the RTSJ.

7 Conclusions

This paper has clarified the intended support for cost enforcement and deadline monitoring in the RTSJ. However, there is not a consistent model for the integration of scheduling, cost enforcement and deadline monitoring for all schedulable objects, irrespective of their release profiles. Modifications to the following interfaces and classes have been proposed to rectify this problem:

- `Schedulable` interface — the addition of methods to `deschedule` and `schedule` a schedulable object at the end of its current release and at the start of its next release respectively.
- `RealtimeThread` class — methods to implement the new functionality in the `Schedulable` interface, a method that allows a real-time thread to wait for its next release and a method that allows the application to indicate that a real-time thread should be released.
- `AsyncEventHandler` class — methods to implement the new functionality in the `Schedulable` interface, a method that is called by the implementation when an asynchronous event handler misses its deadlines and has not specified a handler for this condition.

With these modifications, a model has been proposed which will ensure that schedulable object execution will remain within the cost boundaries the programmer has specified (and has been assumed in the feasibility analysis). They can be used in conjunction with the asynchronous transfer of control facilities to provide structured support for immediately informing a schedulable object that its deadline has been missed or its cost overrun. They are fully integrated within the RTSJ scheduling model.

References

[1] ARM, 1995, "Ada 95 Reference Manual", ANSI/ISO/IEC-8625:1993.

[2] Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., and Turnbull, M., “The Real-Time Specification for Java”, version 0.9, Addison Wesley, 2000 (version 1.0 is available from www.rtj.org).

[3] Burns, A., and Wellings, A.J, "Real-Time Systems and Programming Languages", Addison Wesley, 2001.

[4] Burns, A., and Wellings, A.J., “Processing Group Parameters in the Real-Time Specification for Java”, Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2003.

[5] CHILL, “CHILL – The ITU-T Programming Language”, International Telecommunications Union, ITU-T Recommendation Z.2000, 1999.

[6] Kligerman, E. and Stoeyenko, A., "Real-Time Euclid: a Language for Reliable Real-Time Systems", IEEE TOSE, SE-12(9), pp 941-949, 1986.

[7] Lehoczky, J.P., Sha, L. and J. K. Strosnider, J.K., "Enhanced Aperiodic Responsiveness in a Hard Real-Time Environment", Proceedings of the IEEE RTSS, pp 261-270, 1987.

[8] Miranda Gonzalez, J. and Gonzalez Harbour, M., Ada Issue 95-00307/03, 2002.

[9] PEARL 90, “PEARL 90 Language Report, Version 2.2”, GI-Working Group 4.4.2., 1998.

[10] POSIX.1d, 1999, IEEE std 1003.d-1000, "Portable Operating Sysrem Interface)POSIX) Part 1: Systems Application Programmer Interface (API) Amendment: Additional Realtime Extensions [C Language]", IEEE.

[11] Wellings, A.J., "Concurrent and Real-Time Programming in Java", Wiley, 2004.