Model-based Verification of a Framework for Flexible Scheduling in the Real-Time Specification for Java

Alexandros Zerzelidis
Department of Computer Science
University of York, U.K.
YO10 5DD
alex@cs.york.ac.uk

Andy Wellings
Department of Computer Science
University of York, U.K.
YO10 5DD
andy@cs.york.ac.uk

ABSTRACT
This paper describes a framework for achieving flexible scheduling in the Real-Time Specification for Java (RTSJ), and provides verification of its operation by modelling it as a system of timed automata in the UPPAAL model checker. The proposed approach is a two-level scheduling mechanism where the first level is the RTSJ priority scheduler and the second level is under application control. Minimum, backward-compatible changes to the RTSJ specification are discussed. The only assumptions made are that the RTSJ implementation supports pre-emptive priority-based dispatching of threads, with changes to priorities having immediate effect. The framework model is described and its correctness checked.

Categories and Subject Descriptors
D.4.1 [**Operating Systems**]: Process Management---*Scheduling, Synchronization;*

D.4.7 [**Operating Systems**]: Organization and Design---*Real-time systems and embedded systems*

General Terms
Algorithms, Design, Verification.

Keywords
scheduling framework, flexible scheduling, RTSJ, model checking

1. INTRODUCTION
The RTSJ ([1],[2]) provides a framework from within which real-time scheduling can be performed for single-processor systems. The intention is to support a range of schedulers, all of them conforming to the abstract `Scheduler` class. However, the current specification defines only a base scheduler, the `PriorityScheduler`. So, whilst it is clear that the RTSJ's intention is to support different (and possibly multiple) schedulers, it is far from clear that the provided framework is adequate for this purpose.

JTRES '06, October 11-13, 2006 Paris, France
Copyright 2006 ACM 1-59593-544-4/06/10...$5.00

In [3] we argued that the RTSJ should define a pre-emptive priority based dispatching model and that it should provide a framework within which multiple application-defined schedulers can be implemented. We proceeded with presenting such a framework, including a technique for sharing resources between threads controlled by different schedulers. In this paper we build on our previous work by providing verification of the framework's operation, by modelling it as a system of timed automata in UPPAAL [4].

Section 2 recapitulates the framework to give a basic understanding of its mechanism. Section 3 presents the main parts of the framework model. Section 4 provides a formal analysis of the model. Section 5 briefly touches upon related work, while Section 6 gives the conclusions.

2. FLEXIBLE SCHEDULING FOR RTSJ
Most real-time operating systems today support pre-emptive priority-based dispatching. Consequently, we have argued that the RTSJ should define this as the base scheduling mechanism. However, many modern applications require more flexible scheduling mechanisms [5], [6]. Furthermore, some applications may need to be scheduled by one policy while others may need a different policy; e.g. fixed priority for hard real-time threads and EDF for soft real-time threads. Hence, state-of-the-art real-time OSs nowadays support hierarchical scheduling [9], [10]. Therefore, we have presented in [3] a two-level scheduling scheme, with the RTSJ's priority scheduler at the top level and the second level under application control, which supports hierarchical scheduling within a fixed-priority framework.

In our approach we have tried to keep changes to the API as small as possible by keeping the scheduling mechanism invisible to applications and relying instead on the priority-based dispatching to carry out application-defined scheduling policy decisions. An application-defined scheduler is assigned four priority queues of the `PriorityScheduler`. We name these *high*, *medium*, *medium-lock*, and *low*. This set of priority queues is called a *scheduling band*. We identify a band by the value of its *low* queue, so for example band 5 is a band that extends from priority level 5 to 8 (*low*=5, *medium_lock*=6, *medium*=7, *high*=8). These priorities are used in the following manner:

- When schedulable objects are released (or become unblocked), they are released at the *high* priority level. This priority is where all scheduling decisions are carried out.
- The application-defined scheduler keeps track of the thread with the highest execution eligibility. This object has its priority set to the *medium* level.

- Queue *low* is where all the application scheduler's threads usually reside when they are not running.
- Finally, priority *medium_lock* is associated with object locking and will be discussed later in this section.

To be able to enforce scheduling decisions we need to determine the *scheduling points* in an application. We identify a scheduling point as either i) the release of thread[1], ii) the end of a thread's execution, or iii) a piece of code in a thread that is about to cause a context switch by making a potentially blocking OS or JVM routine call. We encapsulate every scheduling point with two calls to the priority scheduler. For this reason we add four new methods to `PriorityScheduler`:

```
package javax.realtime;
public class PriorityScheduler extends Scheduler {
  protected static final void prepareToSuspend(
            Schedulable sched, int reason);
  protected static final void prepareToSuspend(
            Schedulable sched, Object lock,
            MonitorControl monitor);
  protected static final void reschedule(
            Schedulable sched, Object lock,
            MonitorControl monitor, int state);
  protected static final void reschedule(
            Schedulable sched);
...
}
```

The goal is to give control to the base scheduler (`PriorityScheduler`) *just before* the schedulable object calls a potentially suspending call and *right after* it returns from such a call. We consider `prepareToSuspend()` and `reschedule()` to be called from within RTSJ library code, and therefore execute as part of the thread's execution at the priority of the thread (i.e. there is no special scheduler thread in the system and no new calls in the application). Within these methods the priority scheduler calls the application scheduler, through a one-way API defined in the `ApplicationDefinedScheduler` class, to provide its most eligible thread. The API is:

```
package javax.realtime;
public abstract class ApplicationDefinedScheduler
                extends Scheduler {
 public ApplicationDefinedScheduler(int low, int
   medium_lock, int medium, int high,
   ProcessingGroupParameters cap, int preLevels);
 protected abstract void released(
   Schedulable sched, boolean running);
 protected abstract void preempted(
   Schedulable current, Schedulable newcomer);
 protected abstract void lockedObject(
   Schedulable sched, int objectCeiling);
 protected abstract void unlockedObject(
   Schedulable sched, int objectCeiling);
 protected abstract void suspended(
   Schedulable sched, int reason);
 protected abstract Schedulable getMostEligible();
 protected abstract Schedulable
   compareEligibility(Schedulable sched1,
                      Schedulable sched2);
 protected abstract boolean setScheduler(
   Schedulable sched);
...
}
```

In order to create an application-defined scheduler we must inherit from this class and implement all abstract methods. With this approach, multiple user-defined schedulers can coexist in the system, if they are allocated non-overlapping bands in the RTSJ priority range. Hence, the proposal supports two-level scheduling. Here, we have to point out that our paper focuses on the scheduling aspect of the approach rather than on how to support feasibility analysis. For the latter we assume server-based analysis, as in [8], where each scheduler is in effect a processing group, specified by associating a `ProcessingGroupParameters` object with each application thread.

Under our framework we allow resources to be shared between threads in different bands and also regular fixed-priority threads. To enforce priority inversion avoidance we use Baker's protocol known as the Stack Resource Policy (SRP) [7]. The SRP is an extension of the Priority Ceiling Protocol (PCP) [16]. Baker uses the notion of preemption levels to introduce execution eligibility inversion control to scheduling algorithms with different notions of execution eligibility. The protocol states that a schedulable object can start execution only if it has the highest execution eligibility and its preemption level is higher than the ceiling of each locked resource. To make this check easier, the notion of the system ceiling is introduced, which is the highest ceiling amongst the locked resources. Preemption levels are ideal in helping us control resource sharing between scheduling bands. When constructing an application scheduler we assign it a number of preemption levels. From this range, we assign each thread a *relative preemption level* (*rpl*). However, to be able to compare preemption levels from different bands we need the notion of an *absolute preemption level* (*apl*). Given that each band is allocated l number of preemption levels (l>=4), a thread's absolute preemption level is given by the following **Equation 1**:

$$apl = (\left\lceil \frac{i}{4} \right\rceil - 1) \times l + i \bmod 4 + rpl - 1 \text{, when } i \bmod 4 \neq 0$$

$$apl = (\left\lceil \frac{i}{4} \right\rceil - 1) \times l + 4 + rpl - 1 \text{, when } i \bmod 4 = 0$$

For threads within a band we consider *i*=*low*, while for a fixed-priority thread τ we consider $i = p(\tau)$ and *rpl*=1. In essence, the absolute preemption level is known if we know the pair (*i*, *rpl*). Therefore, we define a resource ceiling to be a pair (*i*, *rpl*) such that the *apl* that it yields is the highest amongst the threads accessing the resource[2]. To extend the RTSJ to support preemption levels, two new classes are introduced:

```
package javax.realtime;
public class PreemptionLevelParameters
       extends PriorityParameters {...}
public class StackResourcePolicy
       extends MonitorControl {...}
```

Implementation of the priority inversion control algorithm is done at the middleware layer and is transparent to the OS. The OS can still enforce its own priority inheritance algorithm, which will be transparent to the middleware. Every application thread is

[1] In this paper we use the term thread to mean any RTSJ schedulable object or Java thread.

[2] This rule is for single unit resources; we will not deal with multi-unit resources here.

assigned an *rpl* within its band. Every resource, which is accessed by threads running under an application-defined scheduler, should be governed by a `MonitorControl` object of type `StackResourcePolicy`. In light of locking, a thread's current band (the band it is currently executing in) can be either its own (original) band or a higher band. Before a thread enters a synchronized region, `prepareToSuspend(sched, lock, monitor)` is called, raising the thread to the *high* priority of its current band, or maintaining the same priority if not in a band. Because of the SRP, the thread is guaranteed not to block, so no checking is needed. The method just sets the thread's priority to the appropriate level: if locking takes place *outside the current band* then the thread is moved to the *high* priority of the higher band or to a fixed-priority level, depending on the resource ceiling. Next the synchronized call takes place (notice that, if within a band, a thread's priority when making the synchronized call is always *high*, irrespective of the band it is in). Immediately after, `reschedule(sched, lock, monitor, LOCK)` is called, which, depending on whether or not the thread is in its own band, it takes the thread to a *medium*, *medium_lock* or normal priority level.

After unlocking we call `reschedule(sched, lock, monitor, UNLOCK)`, which raises the thread to the *high* priority of the current band (if within one, otherwise priority stays the same) and checks to see if there is a thread eligible to run in the current band (it could be the case that a thread was released while the calling thread was holding the lock, but couldn't run because of the system ceiling). At this point there are two things to consider: i) "is the current band the calling thread's own band?", and if not, ii) "is the thread returning to its own band?" If the answer to the first question is yes, then `reschedule()` calls `getMostEligible()`, calls `compareEligibility()` between the returned and the calling thread and places the most eligible at the *middle* queue. If this thread is different from the calling thread, it places the calling thread at the *low* queue. If the answer to the first question is no, then `reschedule()` calls `getMostEligible()` and places the returned thread (if eligible) at the *middle* queue of the current band, and, depending on whether the calling thread is returning to its own or to a higher-than-its-own band, it places it on the *middle* queue of its own band, or on a *middle_lock* queue, or at a fixed-priority level. Finally, if the thread was not locking within a band then `reschedule()` either keeps the thread at the same priority level, or moves it lower to another fixed-priority level, or to a *medium_lock* priority, or to its own band's *medium* priority. Note that the scheme can cope with nested locking. However, there is one condition that needs to hold: **the locking thread must never suspend itself**. In fact, the locking thread must not block under any circumstances. Effectively, every possibly suspending operation must be modelled as a shared resource. As a final note here we point out that coding practices with the SRP are the same as with the PCP, e.g. use of library code is treated as accessing a resource (in case it makes undocumented use of resources) and is assigned the highest absolute preemption level.

3. MODELLING THE FRAMEWORK

3.1 The UPPAAL tool

In this section we will give a brief introduction to the modelling tool UPPAAL. This is a model checker based on the theory of timed automata. It defines a modelling language that extends timed automata with, amongst others: **constants**; **bounded integer variables** with which we can perform arithmetic operations; **binary synchronization channels**, where a transition labelled with `c!` synchronizes with only one (out of possibly multiple) transition labelled `c?`; **broadcast channels**, where one sender `c!` can synchronize with an arbitrary number of receivers `c?`; **urgent locations**, where time is not allowed to pass when the system is in such a location; **committed locations**, where time is not allowed to pass *and* the next transition in the system must involve an outgoing transition from one of the committed locations; **arrays**. To express requirement specifications, UPPAAL has a query language that consists of **state formulae** and **path formulae**. A state formula translates to an individual state, a state being the set of the locations of all automata, all clock values and the values of all discrete variables. A path formula quantifies over paths in the model. Path formulae are classified into *reachability* ("can a particular state be reached?"), *safety* ("something will never happen") and *liveness* ("something will eventually happen"). To express path formulae we use the syntax `[A|E] ["[]"|"<>"]` φ, where φ is a state formula. `A` denotes that a given property should hold for all paths in the system. `E` denotes that there should be at least one path. `"[]"` denotes that all states in the path should satisfy the property, while `"<>"` denotes that at least one state in the path satisfies the property. So, for example, `A[]` φ means that invariantly φ should hold. UPPAAL offers the keyword `deadlock` to describe the state where no outgoing transitions are possible. The reader is referred to [4], [11] for more information on UPPAAL and timed automata in general.

3.2 Architecture Description

In order to test our framework and evaluate its behaviour we have implemented it as a collection of timed automata in UPPAAL. In building our model we have identified 5 basic automata and named them as follows: Thread, PriQueue, BaseScheduler, Dispatcher, Time. The model can be complemented with one or more application scheduler automata. To demonstrate the ability of our framework to accommodate different application level schedulers we have constructed an EDFScheduler automaton. Our model's architecture can be seen in Figure 1. In all our automata we simulate API calls (whether middleware or OS calls) with synchronization channels between automata, which can be seen as arrows in Figure 1, going from the automaton initiating the synchronization to the one receiving it. The channels on the arrows are just examples.

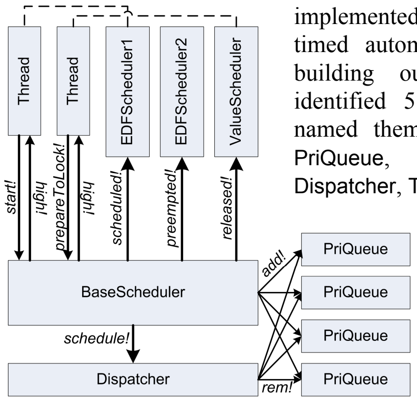

Figure 1: Model architecture

The Thread automaton represents a thread in the system. More than one instance of Thread can be specified. PriQueue models the functionality associated with an operating system FIFO priority queue. In the view of our model an operating system queue is a superset of the equivalent middleware queue in the sense that all threads on the middleware queue exist on OS queues but the inverse is not true. Naturally, there can be more than one instance of PriQueue. The BaseScheduler automaton represents both the middleware and the operating system priority scheduler. This scheduler has the operating system priority levels under its control and maps all application threads down to native threads. In essence we consider every middleware scheduler operation on a middleware priority queue to translate directly to the exact same operation on the equivalent operating system queue. That is why BaseScheduler has direct access to PriQueues. The Dispatcher automaton models the operating system dispatching mechanism. It also interacts with the PriQueues, and, in fact, is part of the scheduler but we have modelled it separately for clarity. The Time automaton increments a variable whenever a Thread is running, thus emulating the passage of absolute time. Finally, the EDFScheduler automaton we have constructed represents an application-defined EDF scheduler and includes variables and data structures (e.g. internal queues). It contains the part of the application scheduler API that is necessary to our model. The EDFScheduler automaton presents a special case since it is “pluggable”. We can replace or combine it with other scheduler automata, as long as we keep the same interface, i.e. the same synchronization channels. This is shown in Figure 1 with a ValueScheduler automaton. We can also have multiple instances of each scheduler automaton.

Every transition in the model is triggered as part of a chain of transitions that originate in the Thread automaton. In the figure we can see that the BaseScheduler plays a central role in our model. Its functionality is triggered by thread actions, e.g. start of a thread, resource locking etc. It then manages thread execution. To do so, it utilises the one-way API with the application defined schedulers (the dashed lines between threads and application schedulers demonstrating which scheduler each thread belongs to) and the Dispatcher, and sets thread priorities. The Dispatcher then sets the running thread.

In describing the automata it is useful to keep in mind that all the code pertaining to a particular edge (transition) is usually situated at some point *above* the transition line, with the *guard* expression being first, followed by the *synchronization channel* and then the *update* code. However, in this paper, due to space limitations, we will only present the 3 major automata, namely Thread, BaseScheduler and EDFScheduler. Full description of the model will be published in a technical report that can be found at: http://www.cs.york.ac.uk/ftpdir/reports/index.php

3.3 Global declarations

The global declarations section of the model contains elements that can be used by any automaton. There are three main types of declarations: constants, variables and synchronization channels.

The constants are further subdivided into two categories: constants that specify maximum values (`MAX_THREAD`, `MAX_PRIORITY`, `MAX_PRELEVEL`, `MAX_RES`, `MAX_LOCK`) and constants that are used for indexing the `thread[][]` array (`PERIOD=0`, `RDEAD=1`, `ADEAD=2`, `LOCK_CALLED=3`, `RELEASED=4`, `PRIORITY=5`, `BAND=6`, `PRELEVEL=7`, `APL=8`, `ITER=9`, `COST=10`, `LAST_LOCKED=11`). `MAX_THREAD` is the maximum possible number of threads in the system, `MAX_PRIORITY` is the maximum number of priorities, `MAX_PRELEVEL` the maximum number of preemption levels per band, `MAX_RES` the maximum number of resources in the system, and `MAX_LOCK` the maximum number resources used by a thread.

The main global variable in the model is the **`thread[][]`** array, which contains all thread parameters, e.g. row `thread[1][]` contains all parameters of thread 1 etc. Using the indexing constants we can access the different parameters for a thread. For example, to get the relative preemption level for thread 1 we access `thread[1][PRELEVEL]`. Cell `thread[][PERIOD]` is the thread’s minimum interarrival time. A thread’s relative deadline is stored in `thread[][RDEAD]`, while the absolute deadline is `thread[][ADEAD]`. To indicate that *prepareToLock!* has been called `thread[][LOCK_CALLED]` is set to true. Analogously, `thread[][RELEASED]` is a boolean indicating whether the thread has just been released. `thread[][PRIORITY]` is the thread’s priority. `thread[][BAND]` is the *low* priority of the thread’s band. `thread[][PRELEVEL]` is the thread’s relative preemption level. `thread[][APL]` is the thread’s absolute preemption level and is set by BaseScheduler during a *reschedule?* synchronization. `thread[][ITER]` is the number of times the thread will execute. `thread[][COST]` is a thread’s execution cost. `thread[][LAST_LOCKED]` is the ceiling of the latest resource the thread has locked (zero if none).

Other global variables follow. **`gclock`** is an integer variable that acts as the global system clock. It is incremented in the Time automaton. The **`PL[]`** array contains pre-calculated values of the absolute preemption level for each priority level in the system. This is calculated according to Equation 1 for fixed-priority threads, i.e. *rpl*=1. For example, for priority level *i*=5, given that the number of preemption levels per band is *l*=100, cell `PL[5]` equals 101. This makes calculations easier in the base scheduler, for example an application thread in band *low* has an absolute preemption level *apl*=`PL[`*low*`]`+*rpl*-1. The **`bands[]`** array has as many cells as are priority levels and, if a priority level has been assigned to a band, the corresponding cell contains the *low* priority of the band; otherwise it is zero. **`run_thread`** holds the thread that is currently running. It is set and unset by the Dispatcher. **`cur_thread`** is used as a parameter between the Thread and BaseScheduler automata to specify which thread performed the last synchronization call to the BaseScheduler. **`app_thread1`** and **`app_thread2`** are used between the BaseScheduler and all application scheduler automata as parameters to certain synchronization calls e.g. *compareEligibility!*. **`cur_queue`** is used for synchronization between the BaseScheduler and PriQueue to specify the target queue. **`cur_band`** is used between the BaseScheduler and the application scheduler automata to specify the target scheduler. The **`system_ceiling[][]`** array keeps a stack of the system ceiling in row 0 (`system_ceiling[0][]`), while in row 1 it keeps the thread that has locked the corresponding resource. **`mutex`** is used between the Thread and BaseScheduler automata and contains the ceiling of the resource to be locked or unlocked and the corresponding band for that ceiling, in the format *xxxy*, where *xxx* is the ceiling and *y* is the low priority of the band (this clearly only works for bands with a *low* priority of 1 to 9 but its enough for the purposes of our model). The **`res[][]`** array contains the resources available in the model. The first row `res[0][]` contains the ceilings of the resources in the same format as `mutex`. The second row `res[1][]` contains the locks of each resource, e.g. `res[1][3]`=1 means that resource 3 is locked.

The last variables are of the form **`threadx_resources[]`**, *x*=1,2… There is one such array for each thread in the system. These arrays contain the indexes of the resources (as they are allocated in the `res[][]` array) to be locked by each thread. Apart from specifying the resources, though, they also provide a schedule for locking the resources. That is, `threadx_resources[0]` will be locked first, then `threadx_resources[1]` will be locked in a nested way, and so on. The number of resources actually locked in each run is random, so we are not specifying one particular scenario through the use of this array.

Finally, there are a number of synchronization channels that are used between automata to guide the execution flow. These are divided according to the automaton which acts upon them. Unless stated otherwise, a channel is ***not*** a broadcast channel. As already mentioned, we will place our focus on the three major automata. For the rest of this paper we will refer to a synchronization channel by writing its name in *italics* followed by a `'!'` or `'?'`, depending on the way it appears in the automaton under question.

Thread: *high?*, *medium?*, *medium_lock?* and *low?* tell Thread to go to the respective location. *time!* is a broadcast channel that marks the passage of "time". The Time automaton synchronizes on this and increases `gclock`.

BaseScheduler: *start?* is equivalent to a thread `start()` method; *reschedule?* is the equivalent of our framework's `reschedule()` method; *prepareToLock?* is the equivalent of `prepareToSuspend(LOCK)`; *reschedule_unlock?* is the equivalent of `reschedule(UNLOCK)`; *reschedule_lock?* is the equivalent of `reschedule(LOCK)`; *prepareToSuspend?* is the equivalent of `prepareToSuspend(END)`.

EDFScheduler: With a *suspended?*, *scheduled?*, *released?*, or *preempted?* synchronization the scheduler is informed that `app_thread1` has been respectively suspended, scheduled, released or preempted, and takes appropriate action. *suspended?* in particular is a broadcast channel that at the same time instructs Thread to go to *Suspended*. *compareEligibility?* tells the scheduler to compare two of its threads (`app_thread1`, `app_thread2`) and returns the most eligible in `app_thread1` and the other in `app_thread2`. With *getMostEligible?* the scheduler places its current most eligible thread in `app_thread2`. *lockedObject?* and *unlockedObject?* inform the scheduler that one of its threads has respectively locked or unlocked a resource, so that it can appropriately adjust its internal queues.

3.4 Thread

Local declarations: `cost` is `thread[][COST]`; `period` is `thread[][PERIOD]`; `runtime` is the amount of CPU time consumed; `iterations` is the number of releases the thread has had; `locked` counts the number of resources currently locked by the thread and is also used as an index for the `to_lock[]` array that contains the indexes of the resources to be locked. So when `locked` is 0, meaning that no resources have been locked, it points to the first cell of the array `to_lock[0]`, which contains the index

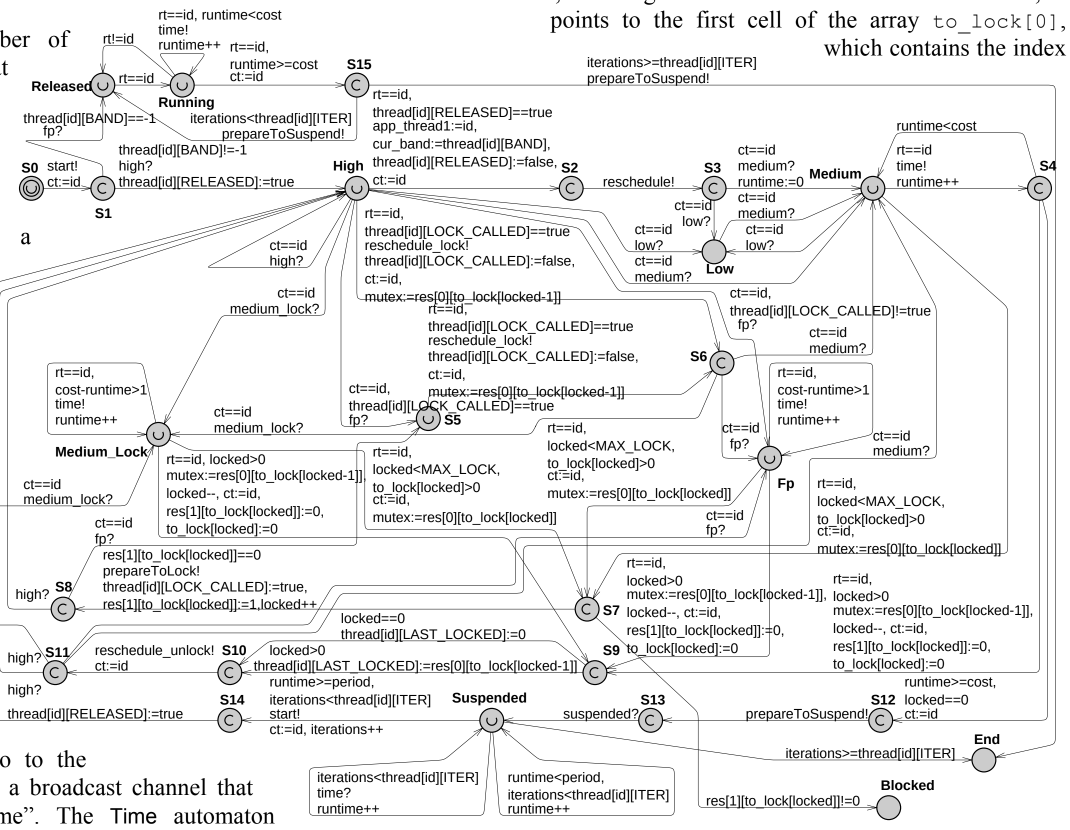

Figure 2: The Thread automaton

to the first resource to be locked. When locked is 1, it points to the second resource to be locked, `to_lock[1]` etc.

Description: The Thread automaton describes a real-time thread, both if it is under direct priority scheduler control and if it is an application scheduler thread. The former case is comprised of two main locations, *Released* and *Running*, which are self-explanatory. The latter case comprises six main locations, *High*, *Medium*, *Low*, *Medium_Lock*, *Fp* and *Suspended*. The first four correspond to the four priority levels of a scheduling band. The *Fp* location is reached when an application scheduler thread locks a resource at a fixed-priority level (i.e. the resource is also used by a fixed-priority thread). When at *Suspended* the thread lies suspended outside the priority queues, waiting for its next release – we exclude other reasons of suspension without loss of generality. Being at *High*, *Medium*, *Medium_Lock* or *Fp* means that the thread is either runnable or running. Being at *Low* means that the thread is runnable but not running. Three other locations complete the set of primary locations for a thread: *S0* is the initial location, where the thread has not been started yet; *End* is the final location in a thread's execution path, where the thread has been completely de-allocated from the system; and *Blocked*, which is where a thread would end up if it was to block on trying to lock a resource. All primary locations, except for *End* and *Blocked*, are urgent, which means that when a thread automaton is in one of them it has to take a valid outgoing transition without delay. This is to guarantee that the system will eventually progress. There are also a number of secondary locations, when moving between primary locations, which are transitory. Their importance lies with the transitions between them rather than with the locations themselves. All secondary locations are committed. The automaton can be seen in Figure 2.

The first transition $S0 \rightarrow S1$ takes place when the thread is started. At *S1* we determine the type of thread. If it is a fixed-priority thread (band=-1), it receives an *fp?* synchronization and is taken to *Released* and rotates between *Released* and *Running* until it has had a specified number of releases, at which point it goes to *End*. If it is an application thread it is taken to priority *high*. From there the thread takes transition $High \rightarrow S2 \rightarrow S3$ that calls for a system reschedule. From *S3* it is taken to either *Medium* or *Low*, depending on whether the thread is the most eligible to run or not. From *Low* the only possible transition is $Low \rightarrow Medium$ when the thread is selected as the most eligible to run. While at *Medium* three transitions are possible: i) the thread might be pre-empted by a higher thread in the band and moved to *Low*, ii) the thread might do normal execution, symbolized by transition $Medium \rightarrow S4$, during which time passes (indicated by a broadcast *time!* synchronization) and its `runtime` local variable is incremented by 1, or iii) it might try to lock a resource following the $Medium \rightarrow S7 \rightarrow S8 \rightarrow High$ path. When the thread tries to lock, it never blocks, due to the SRP. To be able to test for this property we have included the *Blocked* location, which is reached if the resource has already been locked (`res[1][to_lock[locked]]==1`). However, as we will see in the next section, this is never the case and the thread is moved all the way to *High*, in order for the locking call to take place at priority *high*. Before moving to *High*, `thread[id][LOCKED_CALLED]` is set to true in order to take the right reschedule action, namely *reschedule_lock!*. So next we have the transition $High \rightarrow S6$ or $High \rightarrow S5 \rightarrow S6$ (for the sake of clarity, we note that *S5* is the location where a thread is taken in order to call *reschedule_lock!*, when locking a resource at a fixed-priority level). From *S6* the thread is moved to either *Medium*, *Medium_Lock* or *Fp*, depending on whether locking takes place in the thread's own band, in a higher band, or outside bands.

When at *Medium_Lock*, a thread can do one of three things: i) execute, represented by the $Medium\_Lock \rightarrow Medium\_Lock$ transition, during which its `runtime` is again incremented (*time!* is again called); ii) the thread can perform a nested locking call, represented by transition $Medium\_Lock \rightarrow S7 \rightarrow S8 \rightarrow High$ (same principles apply as when locking at *Medium*); or iii) the thread can unlock the latest locked resource, following the $Medium\_Lock \rightarrow S9 \rightarrow S10 \rightarrow S11 \rightarrow High$ path. First, it sets the resource entry in the `to_lock[]` array to zero, so that it will not be locked again; in transition $S9 \rightarrow S10$ it sets the `thread[id][LOCKED]` variable to the latest locked resource; then it calls *reschedule_unlock!*, which first puts the thread to the *high* priority of the higher band it is in (thread goes to location *High*) and then the thread is moved to either *Low*, *Medium*, *Medium_Lock* or *Fp*, depending on the circumstances.

At *Fp* the thread has the same choices as at *Medium_Lock*. It can: i) execute ($Fp \rightarrow Fp$), ii) perform a nested lock call via $Fp \rightarrow S7 \rightarrow S8$ and then move to either *High* or *S5* to perform a *reschedule_lock!* (depending on whether it is next locking within or outside a band), or iii) unlock a resource ($Fp \rightarrow S9 \rightarrow S10 \rightarrow S11$), ending up in either *Medium_Lock*, *Fp* or *Medium*, depending on the circumstances.

In order to finish its release a thread must have relinquished all resources and executed for more than or equal to its cost. So from *S4* a thread can either: i) go to *Medium*, if `runtime` is still less than `cost`, ii) unlock a resource through transition $S4 \rightarrow S9 \rightarrow S10 \rightarrow S11 \rightarrow High$, if it still holds one, or iii) finish its release and suspend itself through transition $S4 \rightarrow S12 \rightarrow S13 \rightarrow Suspended$, only if it holds no locks and has executed for (or more than) its `cost`.

While suspended the thread can increase its `runtime` through $Suspended \rightarrow Suspended$ in one of two ways: either on its own or by accepting a *time?* synchronization (here we can see that `runtime` is used as a more general variable of counting time and not only for counting CPU execution time). If `runtime` equals or exceeds the thread's `period` parameter *and* if the number of performed thread iterations is less than the number of maximum thread invocations (`iterations<thread[id][ITER]`), the thread is released again. Otherwise, if the number of maximum invocations has been reached, the thread is moved to *End* and is de-allocated from the system. When all threads reach *End*, the system has finished execution.

3.5 BaseScheduler

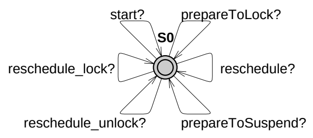

Figure 3: BaseScheduler abstract automaton

Local variables: All variables hold temporary values to assist in setting up transition guards etc. `i` is a counter; `ceil` holds the ceiling of a resource; `res` is shorthand for `thread[cur_thread][LAST_LOCKED]`; `next_band` holds the calculated value of the band the thread is next going to, after a lock or unlock.

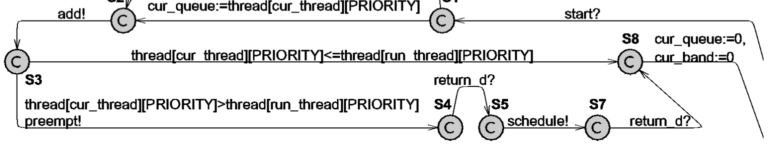
thread[cur_thread][BAND]==-1
fp!
cur_queue:=thread[cur_thread][PRIORITY]
thread[cur_thread][BAND]!=-1
high!
cur_queue:=thread[cur_thread][PRIORITY]
S2
S1
add!
start?
S8
cur_queue:=0,
cur_band:=0
thread[cur_thread][PRIORITY]<=thread[run_thread][PRIORITY]
S3
return_d?
thread[cur_thread][PRIORITY]>thread[run_thread][PRIORITY]
preempt!
S4
S5
schedule!
S7
return_d?

Figure 4: The *start!* synchronization

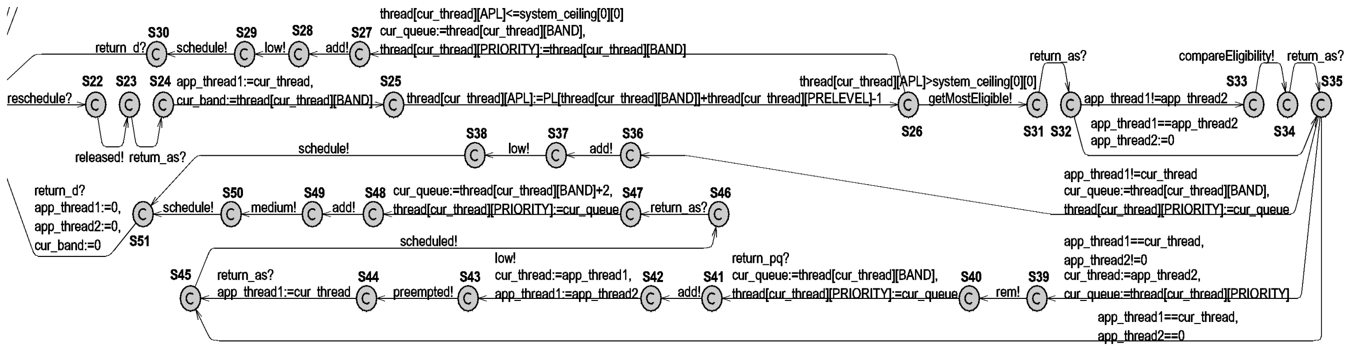

Figure 5: The *reschedule!* synchronization

Description: The `BaseScheduler` automaton waits at the initial location *S0* for a Thread to synchronize on one of six channels. This can be seen in Figure 3, which is an abstract depiction of the automaton. Due to the size of the BaseScheduler automaton we will not present all synchronizations but rather focus on the most

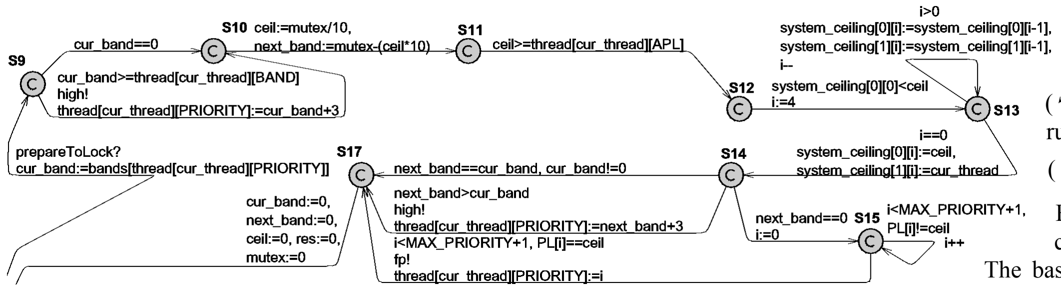
S9
cur_band==0
S10
ceil:=mutex/10,
next_band:=mutex-(ceil*10)
S11
ceil>=thread[cur_thread][APL]
cur_band>=thread[cur_thread][BAND]
high!
thread[cur_thread][PRIORITY]:=cur_band+3
i>0
system_ceiling[0][i]:=system_ceiling[0][i-1],
system_ceiling[1][i]:=system_ceiling[1][i-1],
i--
S12
system_ceiling[0][0]<ceil
i:=4
S13
i==0
system_ceiling[0][i]:=ceil,
system_ceiling[1][i]:=cur_thread
prepareToLock?
cur_band:=bands[thread[cur_thread][PRIORITY]]
S17
next_band==cur_band, cur_band!=0
S14
next_band>cur_band
high!
thread[cur_thread][PRIORITY]:=next_band+3
cur_band:=0,
next_band:=0,
ceil:=0, res:=0,
mutex:=0
next_band==0
i:=0
S15
i<MAX_PRIORITY+1,
PL[i]!=ceil
i++
i<MAX_PRIORITY+1, PL[i]==ceil
fp!
thread[cur_thread][PRIORITY]:=i

Figure 6: The *prepareToLock!* synchronization

important ones. All locations in the automaton are specified as committed so as to guarantee that the execution of the scheduler will not be interrupted. In the synchronization diagrams presented, those transitions cut at the edge of the diagram come from or lead to the initial location *S0*.

Initially, a thread calls *start!* (Figure 4). The new thread is placed on its band's *high* queue, if it is an application-scheduled thread, or on its priority's queue, if it is a normal thread ($S1 \rightarrow S2 \rightarrow S3$). Then, depending on whether its priority is greater than the running thread's priority or not, it either preempts the running thread ($S3 \rightarrow S4 \rightarrow S5$) and a reschedule is called ($S5 \rightarrow S6 \rightarrow S7$), or the running thread continues to run ($S3 \rightarrow S7$).

Following that, a thread will call *reschedule!* (Figure 5). The base scheduler first informs the application scheduler of the release ($S22 \rightarrow S23$) and then calculates the absolute preemption level for the thread ($S25 \rightarrow S26$). If the calculated level is lower than the system ceiling, the thread is put in the *low* queue ($S26 \rightarrow ... \rightarrow S30 \rightarrow S0$); otherwise the application scheduler is asked for its most eligible thread ($S26 \rightarrow S31$). If `app_thread2` is a valid thread, it is compared to the new thread

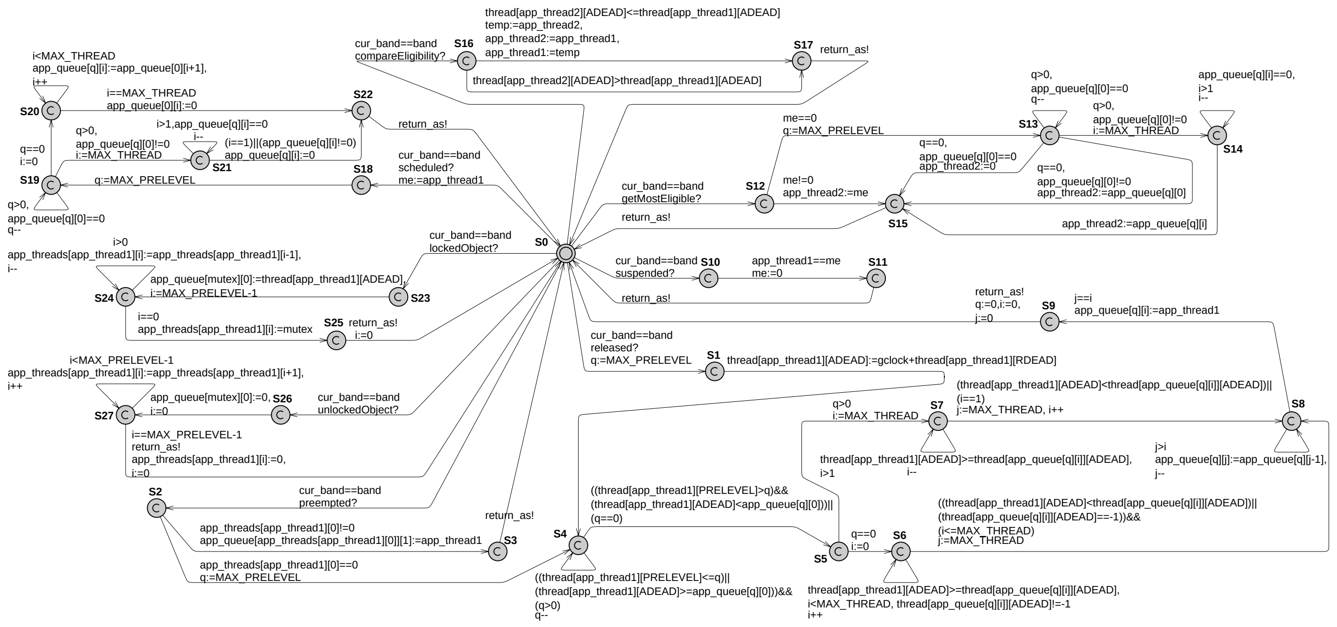
S0
cur_band==band
compareEligibility?
S16
thread[app_thread2][ADEAD]<=thread[app_thread1][ADEAD]
temp:=app_thread2,
app_thread2:=app_thread1,
app_thread1:=temp
S17
return_as!
thread[app_thread2][ADEAD]>thread[app_thread1][ADEAD]
i<MAX_THREAD
app_queue[q][i]:=app_queue[0][i+1],
i++
S20
i==MAX_THREAD
app_queue[0][i]:=0
S22
return_as!
q>0,
app_queue[q][0]!=0
i:=MAX_THREAD
i>1,app_queue[q][i]==0
i--
(i==1)||(app_queue[q][i]!=0)
app_queue[q][i]:=0
S21
q==0
i:=0
S19
q:=MAX_PRELEVEL
S18
cur_band==band
scheduled?
me:=app_thread1
q>0,
app_queue[q][0]==0
q--
i>0
app_threads[app_thread1][i]:=app_threads[app_thread1][i-1],
i--
cur_band==band
lockedObject?
app_queue[mutex][0]:=thread[app_thread1][ADEAD],
i:=MAX_PRELEVEL-1
S23
S24
i==0
app_threads[app_thread1][i]:=mutex
S25
return_as!
i:=0
i<MAX_PRELEVEL-1
app_threads[app_thread1][i]:=app_threads[app_thread1][i+1],
i++
S27
app_queue[mutex][0]:=0,
i:=0
S26
cur_band==band
unlockedObject?
i==MAX_PRELEVEL-1
return_as!
app_threads[app_thread1][i]:=0,
i:=0
S2
cur_band==band
preempted?
return_as!
app_threads[app_thread1][0]!=0
app_queue[app_threads[app_thread1][0]][1]:=app_thread1
S3
app_threads[app_thread1][0]==0
q:=MAX_PRELEVEL
me==0
q:=MAX_PRELEVEL
S13
q>0,
app_queue[q][0]==0
q--
q>0,
app_queue[q][0]!=0
i:=MAX_THREAD
app_queue[q][i]==0,
i>1
i--
S14
q==0,
app_queue[q][0]==0
app_thread2:=0
q==0,
app_queue[q][0]!=0
app_thread2:=app_queue[q][0]
cur_band==band
getMostEligible?
S12
me!=0
app_thread2:=me
S15
app_thread2:=app_queue[q][i]
return_as!
cur_band==band
suspended?
S10
app_thread1==me
me:=0
S11
return_as!
return_as!
q:=0,i:=0,
j:=0
S9
j==i
app_queue[q][i]:=app_thread1
cur_band==band
released?
q:=MAX_PRELEVEL
S1
thread[app_thread1][ADEAD]:=gclock+thread[app_thread1][RDEAD]
(thread[app_thread1][ADEAD]<thread[app_queue[q][i]][ADEAD])||
(i==1)
j:=MAX_THREAD, i++
q>0
i:=MAX_THREAD
S7
S8
thread[app_thread1][ADEAD]>=thread[app_queue[q][i]][ADEAD],
i>1
i--
j>i
app_queue[q][j]:=app_queue[q][j-1],
j--
((thread[app_thread1][PRELEVEL]>q)&&
(thread[app_thread1][ADEAD]<app_queue[q][0]))||
(q==0)
S4
((thread[app_thread1][PRELEVEL]<=q)||
(thread[app_thread1][ADEAD]>=app_queue[q][0]))&&
(q>0)
q--
q==0
i:=0
S5
S6
((thread[app_thread1][ADEAD]<thread[app_queue[q][i]][ADEAD])||
(thread[app_queue[q][i]][ADEAD]==-1))&&
(i<=MAX_THREAD)
j:=MAX_THREAD
thread[app_thread1][ADEAD]>=thread[app_queue[q][i]][ADEAD],
i<MAX_THREAD, thread[app_queue[q][i]][ADEAD]!=-1
i++

Figure 7: The EDFScheduler automaton

($S33 \rightarrow S34$), and the most eligible of the two is placed in the *medium* queue while the other in the *low* queue (transition $S35 \rightarrow S39 \rightarrow ... \rightarrow S45$ if the new thread is more eligible, transition $S35 \rightarrow ... \rightarrow S38 \rightarrow S51$ if the previous one is). Essentially the current most eligible thread (if any), is the previously running thread, which the new thread preempts. So if it is more eligible we do not have to do anything about it, since it already is in the *medium* queue.

When a thread is running, it can synchronize on *prepareToLock?*. This synchronization can be seen in Figure 6. At first, the thread, if running within a band, is placed on its *high* queue ($S9 \rightarrow S10$). Once the resource ceiling has been calculated, we make sure it is greater or equal to the locking thread's absolute preemption level and also greater than the system ceiling ($S10 \rightarrow ... \rightarrow S13$). We then update the system ceiling stack, saving also which thread performed the locking ($S13 \rightarrow S14$). Then the base scheduler moves the thread to the appropriate priority before returning control to the thread. Depending on the resource ceiling, the thread may remain at the same priority, or go to the *high* priority of a higher band ($S14 \rightarrow S17$). If locking outside a band, the appropriate priority level is calculated with the help of the `PL[]` array ($S14 \rightarrow S15 \rightarrow S17$).

After locking takes place a thread synchronizes on *reschedule_lock?*. The base scheduler moves the thread to the appropriate priority and informs its application scheduler of the locking through *lockedObject!*. The thread is taken to a *medium_lock* priority, if the thread is at a higher band than its own, or its *medium* priority, if the thread is locking in its own band. If the thread is locking outside a band then its priority remains unaltered. At the end the Dispatcher is called to schedule the system.

When unlocking a thread synchronizes on *reschedule_unlock?*. This puts the thread on the correct priority queue and checks whether another more eligible thread became available while the current was locking the resource, but was unable to preempt due to the system ceiling. The most important part of this is synchronizing on *getMostEligible!* with the current band, if the thread is in one. If the returned thread is more eligible, it is placed on its *middle* queue. At the end, the unlocking thread is appropriately placed on a *medium_lock*, *medium*, *low*, or fixed-priority queue, and a system schedule is called.

The final part of the BaseScheduler automaton is the *prepareToSuspend?* synchronization. If the suspended thread is a simple fixed-priority thread, then the base scheduler just tells the Dispatcher to *schedule!*. If the thread belongs to a band, the base scheduler initially informs the thread's scheduler, through *suspended!,* that its thread has been suspended. Then that scheduler is asked, through *getMostEligible!*, to provide its next most eligible thread. If that thread is null, or if it is a thread with a preemption level lower than the system ceiling, then again the base scheduler just calls for a system schedule. Finally, if the application scheduler returns a thread with higher preemption level than the system ceiling, then that thread is scheduled, moved to *medium* priority, and a system schedule is called.

3.6 EDFScheduler

Local variables: The `app_queue[][]` array is the scheduler's internal EDF queues, which follow the model found in [12]; the `app_threads[][]` array holds each EDF thread's locking history; `me` is the current most eligible thread; `i`, `j`, `q` and `temp` are helper variables.

Description: The EDFScheduler automaton (Figure 7) is the only one in the model that is pluggable. That is, we can substitute it with a different scheduler automaton. We can also have many instances of scheduler automata, each one constituting a different scheduling band. As we have previously seen, when a thread is released, its scheduler is informed via the *released?* channel. This causes the scheduler to calculate the thread's absolute deadline ($S1 \rightarrow S4$) and insert it into its scheduling queues (`app_queue[][]`), according to the rules found in [12] ($S4 \rightarrow ... \rightarrow S9$). When a thread is preempted while holding a lock, the scheduler puts it directly in the appropriate queue ($S2 \rightarrow S3$), according to the resource ceiling that is stored in the `app_threads[][]` array. If not holding a lock when preempted, the exact same steps as *released?* are taken to place the thread on the queues. Conversely, if a thread is *scheduled?* it is removed from the queues and it is set as the band's most eligible thread (`me`) ($S18 \rightarrow ... \rightarrow S22$). If the scheduler is asked for its most eligible thread through *getMostEligible?*, it returns `me`, if `me!=0` (i.e. the thread that was running in the band), or the first thread on the `app_queue`, if `me==0` ($S12 \rightarrow ... \rightarrow S15$). If the scheduler is synchronized on *compareEligibility?* to compare `app_thread1` and `app_thread2`, it places the one with the shorter absolute deadline in `app_thread1` and the other in `app_thread2` ($S16 \rightarrow S17$). Finally, if a thread is suspended the scheduler just resets the `me` variable to 0 ($S10 \rightarrow S11$).

4. FORMAL ANALYSIS OF THE MODEL

In this section we specify certain properties both to evaluate the correctness of our model and to explore its behaviour. To test these properties we have defined a system with 10 priority levels (i.e. 10 PriQueue automata) and two EDF bands with $low_{EDF1} = 1$ and $low_{EDF2} = 6$ (Figure 8). It follows that priority levels 5 and 10 are left under direct priority scheduler control. In order to keep our system analysable we have only 3 threads (3 Thread automata). Based on this we have specified a number of scenarios to test against, which can be seen in Table 1.

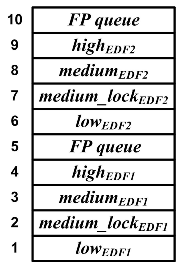

Figure 8: The test system

Table 1: Number of threads per band

Test case	EDF1	Priority 5	EDF2	Priority 10
A	1	1	1	0
B	1	1	0	1
C	1	0	1	1
D	2	1	0	0
E	0	1	2	0
F	2	0	1	0

We also specify 5 resources. The rule is that an application thread in any scenario uses one resource in its own band and one in each band/priority above its band that has a thread, e.g. in case A the thread in EDF1 locks one resource within EDF1, one at priority level 5 and one in EDF2.

Model Consistency First we specify three properties to check the correctness of our model.

Property 1: *Throughout its execution a thread will only be placed on either its own band's low, medium or high queue, or on a higher band's medium_lock or high queue, or on a normal priority queue above its band.*
We specify a safety property for each thread. For the above-mentioned thread in case A the property is written: A[] ((thread[3][PRIORITY]!=2) && (thread[3][PRIORITY]!=6) && (thread[3][PRIORITY]!=8) && (thread[3][PRIORITY]!=10) &&(thread[3][PRIORITY]<11)&&(thread[3][PRIORITY]>=0)). As we have seen, `thread[][5]` is a thread's priority, so we check that it never takes an invalid value. The property is satisfied for all threads.

Property 2: *No thread can block due to locking after it starts.*
We specify the safety property A[] !((Thread2.Blocked) || (Thread3.Blocked) || (Thread4.Blocked)), which checks that under no circumstances does a Thread go to the *Blocked* location. This property is also satisfied.

Property 3: *The system will always select a thread to run with higher absolute preemption level (apl) than the system ceiling, unless the selected thread is currently locking a resource with higher ceiling than its apl, or is a thread that has just been released.*
This property is expressed as follows: A[] (Dispatcher.S10 && run_thread>0 && thread[run_thread][APL]>0 && thread[run_thread][RELEASED]!=true) imply ((thread[run_thread][APL]>system_ceiling[0][0]) || ((thread[run_thread][APL]<=system_ceiling[0][0]) && (system_ceiling[1][0]==run_thread)))

Dispatcher.S10 is the location where the dispatcher has picked the next thread to run and assigned it to `run_thread`. So at that point, if we have a `run_thread` and if its *apl* has been calculated and if it hasn't just been released, then it will either have an *apl* greater than the system ceiling or it will be holding the lock to the resource that set the ceiling. This property is satisfied.

Exploring the behaviour The next two properties guarantee the unhindered progress of the system.

Property 4: *The system is livelock free.*
This property checks to see whether all threads reach the end of their execution. It is written as a liveness property: A<> ((Thread2.End)&&(Thread3.End)&&(Thread4.End)). This is satisfied.

Property 5: *The system can never deadlock.*
This safety property can be expressed as A[] (not deadlock), using the UPPAAL built-in keyword *deadlock*. This property is not satisfied. However, if we ask UPPAAL for the shortest trace to a deadlock, we get the state where all threads are at the End location. This state is the final state after all thread executions. Since the End location has no outgoing transitions, it follows that this is the only deadlock in the system. This essentially means that the system is deadlock-free. This can also be demonstrated with a positive example, if we add a loop transition $End \rightarrow End$. With this loop the "not deadlock" property is satisfied.

Properties 2, 3 and 5 prove the following important property.

Corollary 1: *The system implements the SRP correctly and enforces its rules whenever resource locking takes place.*

It is useful to note here that priority threads not belonging to a band can still use other priority inheritance protocols when they do not share resources with scheduling band threads. This is easy to understand if we consider bands to be "black boxes" to any thread outside them. A thread would have its priority elevated by the priority inheritance algorithm, bypassing any bands situated between its base and active priorities. The priority scheduler would then, as normal, pick the next thread to run.

5. RELATED WORK

There are three approaches to achieve flexible scheduling (see [3]):

Pluggable schedulers – in this approach the system provides a framework into which different schedulers can be plugged. The CORBA Dynamic Scheduling [17] specification is an example of this approach. Kernel loadable schedulers also fall into this category, such as that used within the SHaRK kernel [18].

Application-defined schedulers – in this approach, the system notifies the application every time an event occurs that requires a scheduling decision to be taken. The application then informs the system which thread should execute next. The proposed extensions to real-time POSIX support this approach [19].

Implementation-defined schedulers – in this approach, an implementation is allowed to define alternative schedulers. Typically this would require the underlying operating system (virtual machine, in the case of Java) to be modified. The Ada 95 language allows this approach.

Currently, the RTSJ adopts the implementation-defined scheduler approach. Unfortunately, this is the least portable approach, as an application cannot rely on any particular implementation-defined scheduler being supported. The only scheduler an application can rely on being present is the `PriorityScheduler`. The work reported in this paper only assumes the presence of the priority scheduler and that priority changes have an immediate effect. An attempt has been made [13] to support a utility accrual scheduler in the RTSJ but this required a non standard interface and was not generalized. Similarly, although JTime supports multiple schedulers, this has been achieved in an ad hoc manner [1]. The use of dynamic priority changes to support alternative scheduling policies is well established. The approach adopted here is based on [14]. Li et al [15] have recently taken this approach and provided a formalized POSIX framework, although they do not support resource sharing between different schedulers.

6. CONCLUSIONS

In [3] we provided a backward compatible hierarchical approach to introducing application-defined schedulers in the RTSJ. In this paper we have presented a model of the proposed framework and have proven its main properties, through the use of timed automata in the UPPAAL modelling tool.

As future work it is our intention to extend our approach to cater for the problem where a thread does suspend while holding a lock.

This is allowed in the RTSJ and therefore it is desirable to account for such a scenario.

Acknowledgments
The authors would like to thank Osmar Marchi dos Santos for his insightful comments on this work.

7. REFERENCES
[1] Dibble, P. and Wellings, A.J. (2004), "The Real-Time Specification for Java: Current Status and Future Direction", 7th International Conference on Object-Oriented Real-Time Distributed Computing (ISORC), pp. 71--77.

[2] Dibble, P. (Ed) (2005), "The Real-Time Specification for Java", Version 1.0.1, www.rtsj.org

[3] Zerzelidis, A. and Wellings, A.J. (2006), "Getting More Flexible Scheduling in the RTSJ", 9th International Symposium on Object-oriented and distributed Real-time Computing (ISORC), pp. 3--10.

[4] Behrmann, G., David, A. and Larsen, K.G. (2004), "A Tutorial on UPPAAL", 4th International School on Formal Methods (SFM-RT 2004), LNCS 3185, pp. 200--236.

[5] Brandt et al (2003), "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes", pp.396, 24th IEEE RTSS.

[6] Regehr, J., Jones, M. B., and Stankovic, J. A. (2000), "Operating System Support for Multimedia: The Programming Model Matters", Technical Report MSR-TR-2000-89, http://research.microsoft.com/~mbj/papers/tr-2000-89.pdf

[7] Baker, T.P (1991) , "Stack-Based Scheduling of Real-Time Processes", Real-Time Systems, 3(1), pp. 57-99.

[8] Davis, R.I. and Burns, A. (2005), "Hierarchical Fixed Priority Pre-Emptive Scheduling", 26th IEEE International Real-Time Systems Symposium, pp. 389-398.

[9] Stankovic, John A. and Rajkumar, R. (2004), "Real-Time Operating Systems", Real-Time Systems, Volume 28, Issue 2 - 3, Nov 2004, Pages 237 - 253.

[10] Aldea, M. et al. (2006), "FSF: A Real-Time Scheduling Architecture Framework", 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 113-124.

[11] Bengtsson, J. and Yi, W. (2004), "Timed Automata: Semantics, Algorithms and Tools", in Lecture Notes on Concurrency and Petri Nets, LNCS 3098, Springer-Verlag.

[12] Burns, A., Wellings, A.J. and Taft, T. (2004), "Supporting Deadlines and EDF Scheduling in Ada", LNCS 3063, Springer-Verlag, Pages 156 – 165.

[13] Feizabadi et al (2003), "Utility Accrual Scheduling with Real-Time Java", JTRES 03, pp. 550-563, Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Vol. 2889/2003.

[14] Burns, A., and Wellings, A.J. (1995), "Concurrency in Ada, 2nd Edition", Cambridge University Press.

[15] Li et al (2004), "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems", IEEE Transactions on Software Engineering, 30(9), pp. 613-629.

[16] Sha, L., Rajkumar, R., and Lehoczky, J. P. (1990), "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, 39(9) (Sep. 1990), 1175-1185. DOI= http://dx.doi.org/10.1109/12.57058

[17] OMG (2003), "Real-time Corba Version 2.0", OMG Document formal/03-11-01, http://www.omg.org/docs/formal/03-11-01.pdf

[18] Gai et al (2001), "A New Kernel Approach for Modular Real-Time Systems Development", p. 199, Proceedings of the 13th Euromicro Conference on Real-Time Systems.

[19] Aldea Rivas, M., and González Harbour, M. (2002), "POSIX-Compatible Application-Defined Scheduling in MaRTE OS", 14th Euromicro Conference on Real-Time Systems, IEEE Computer Society Press, pp. 67–75.