Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java

Andy Wellings, Alan Burns, Osmar Marchi dos Santos∗
University of York – UK
{andy, burns, osantos}@cs.york.ac.uk

Benjamin M. Brosgol
AdaCore – USA
brosgol@adacore.com

Abstract
Priority inversion and priority inheritance protocols for bounding blocking time are well-understood topics in real-time systems research. The two most commonly used priority inheritance protocols are basic priority inheritance and priority ceiling emulation. Although both are supported in POSIX, Ada and the Real-Time Specification for Java (RTSJ), little has been written about the consequences of using both protocols concurrently in the same program. The assumption is usually that only one is in force at any particular time. For large real-time systems, this assumption may not be valid. This paper provides motivation for why a mixture of the two can occur and illustrates that this can result in the raising of unwanted asynchronous exception. This has led the Technical Interpretation Committee for the RTSJ to propose a new version of the priority ceiling emulation protocol that will enable it to work in harmony with basic priority inheritance. The protocol is described and modelled using the UPPAAL tool, where formal properties are explored using model checking.

1 Introduction
In priority-based systems, unbounded blocking time and deadlock situations may rise from having lower priority threads executing (with mutually exclusive) shared resources whilst higher priority threads are trying to gain access to the same resources. This is a well-known problem, called priority inversion [13], and several protocols for boosting the priority of lower priority threads, when accessing shared resources, have been proposed to bound the blocking. For uniprocessor systems, the most popular protocols are [12]: basic priority inheritance and priority ceiling emulation. Whilst use of the individual protocols is well understood [11], little work has been done on the properties of systems that contain a mixture of inheritance protocols. Although the POSIX standard allows both priority inheritance and priority ceiling emulation (called the priority protect protocol), nothing is specified about the interaction between the two. The Ada language supports priority ceiling emulation (called immediate priority ceiling inheritance) and a weak form of priority inheritance. However, as the priority inheritance support is so limited, it is not possible for interactions to occur. Most Real-Time Operating Systems (RTOSs) support just priority inheritance.

The Real-Time Specification for Java (RTSJ) augments the semantics of the Java programming language and virtual machine in order to make it suitable for real-time computing [3]. The initial version of the RTSJ [4] followed POSIX’s and Ada’s lead and provided support for the two main priority inheritance algorithms. A real-time Java virtual machine must support priority-ordered queues and perform basic priority inheritance1 whenever high

∗This author is partially sponsored by CAPES-Brazil.

1From now on we will use the term priority inheritance rather than basic priority inheritance.
priority threads are blocked by low priority real-time threads. There are many locks that are held by the virtual
machine (for example, in order to implement real-time garbage collection) that are invisible to the application. Pri-
ority inheritance automatically occurs when using these locks. Furthermore, the RTSJ allows the Java’s application
locks (associated with any object that has a synchronized method or that is used in a synchronized statement) to
be supported by priority inheritance or priority ceiling emulation. The default is priority inheritance. The result
is that an application can consist of threads that use nested locks that might be controlled by a mixture of the two
inheritance protocols. As with POSIX, little attention was paid to the semantics of any interactions.

This paper is organized as follows. In Section 2 we introduce the changes that the RTSJ has made to Java syn-
chronization semantics and present the details of the approach to controlling priority inversions. We illustrate some
of the problems that will be encountered when the priority inheritance protocol is used in conjunction with prior-
ity ceiling emulation. The main problem is that a thread executing inside a synchronized method (or statement)
protected by priority ceiling emulation can suddenly find itself executing above the ceiling priority as a result of
priority inheritance. Under these circumstances, the priority ceiling emulation protocol requires an asynchronous
exception to be raised. We define a new version of the priority ceiling emulation protocol that removes this prob-
lem. This protocol forms the basis of the most recent versions of the RTSJ’s PriorityCeilingEmulation monitor
control policy. In Section 3 we present a formal model of the synchronisation protocol. Section 4 then analyzes
certain formal properties of the protocol. Conclusions and final remarks are presented in Section 5.

2 The RTSJ Synchronisation Protocol

The RTSJ enhances several areas of the Java programming language and virtual machine [14]. For our purposes,
the changes to the scheduling model are the most important. Both real-time threads and asynchronous event
handlers are defined (collectively called schedulable objects\(^3\)). All implementations must support priority-based
scheduling. Each schedulable object has a base and an active priority. The base priority is the priority allocated by
the programmer. The active priority is the priority that the scheduler uses to order the run queue. As mentioned
before, the real-time Java Virtual Machine (JVM) must support priority-ordered queues and perform priority inher-
itance whenever high priority threads are blocked by low priority ones. The active priority of a thread is, therefore,
the maximum of its base priority and the priority it has inherited.

In order to allow the programmer to specify an appropriate priority inheritance algorithm for its application-
defined locks (those associated with any objects that have synchronized methods or that are used in a synchronized
statement), three classes are defined. The abstract class MonitorControl defines a static method that allows the
default priority inheritance algorithm to be set along with a static method that allows a particular object to have the
default overridden. Two subclasses of MonitorControl are provided: PriorityInheritance and PriorityCeilingEm-
ulation, which allow the programmer to specify the priority inheritance and the priority ceiling emulation algorithm
respectively.

The Technical Interpretation Committee for the RTSJ was set up in 2001 to respond to questions about the
specification. It was clear that whilst the initial designers’ had done a good job in addressing the weaknesses of
Java, Version 1.0 was under specified, and the designers intentions were not clear in many places. Consequently,
it was necessary to undertake a major rewrite in order to tighten up the semantics. During this process, it became
clear that we did not fully understand the implications of allowing applications to have both inheritance protocols
in operation at the same time – let alone what would happen when the programmer dynamically changed priorities

\(^2\)Wellings and Brosgol are members of the RTSJ Technical Interpretation Committee. The research reported in this paper is a result of that
committee rewriting the Version 1.0 specification to be more rigorous.

\(^3\)For the remainder of this paper we will use the term thread to include both types of schedulable objects.
and even protocols. Furthermore, when we reviewed the literature we were unable to find any help. One of the goals of the RTSJ is to support large real-time systems with a mixture of hard, soft and non-real-time threads. We had to assume that such applications would make full use to the bountiful pre-written Java libraries. This software will inevitably obtain locks, as will the underlying JVM. Some of the JVM locks may actually be RTOS locks, which will usually be priority inheritance locks. Consequently, we either had to remove our support for priority ceiling emulation (to the detriment of the hard real-time threads), or we had to accept that applications may have the two locking protocols executing concurrently and that nested locks governed by different protocols is possible. We adopted the latter position.

2.1 Mixing Priority Inheritance and Priority Ceiling Emulation

Priority inheritance (PI) is an appropriate synchronization protocol in large real-time systems where it is often difficult to determine the pattern of indirect synchronization between threads. In PI, a thread holding a lock inherits the highest priority of all threads attempting to acquire the lock. Its main advantages are that it is widely supported by RTOSs, priority changes only occur when needed (there is no cost in the common case when the lock is not in use). Its main disadvantages are that a thread may be blocked separately for each lock that it needs (and, therefore, deadlock can occur), “chained blocking” may occur when threads are waiting for locks that are held by other threads that are waiting for locks held by different threads, and implementation may be expensive because of nested (recursive) inheritance and the fact that a thread’s priority is changed by an action external to the thread.

The priority ceiling emulation (PCE) protocol allocates each lock a ceiling priority. This ceiling is set to the maximum active priority that a thread requesting the lock can acquire. When a thread acquires the lock, its active priority is immediately raised to (if it is not already at) the ceiling. If the thread’s current active priority is already greater than the ceiling, a run-time exception is thrown. The protocol’s main advantages are: if no thread can block while holding the lock then a queue is not needed for that lock (the processor is the lock), “nested monitor” deadlock is prevented, a thread can be blocked at most once during each release by some lower priority thread holding the lock. The disadvantages are: computation of ceilings needs careful analysis, especially if thread priorities and ceiling values can change dynamically; it requires a check and priority change at each call\(^4\) (used to prevent unbounded priority inversion); there is overhead even if an object is not locked.

When PI or PCE locking protocols are mixed, asynchronous exceptions are possible. Consider the following scenario. Thread \(T_B\) (priority medium) shares PCE lock \(L_X\) with thread \(T_A\) (priority low). The ceiling priority of \(L_X\) is, therefore, set to medium. \(T_B\) executes and acquires \(L_X\) (there is no change to the active priority because the ceiling of \(L_X\) is medium), it then performs some action that requires the JVM to acquire an internal PI lock \(L_Y\). Thread \(T_B\) is now preempted by a high priority thread \(T_C\) that executes a JVM operation that requires lock \(L_Y\). This is a PI lock and, consequently, priority inheritance occurs. Thread \(T_B\) has the lock so its priority is asynchronously boosted to the high priority. \(T_B\) is now executing within lock \(L_X\) at an active priority greater than the ceiling priority of \(L_X\). According to the PCE protocol this is an error condition and an asynchronous exception is raised in thread \(T_B\). If fact, as thread \(T_B\) will release lock \(L_Y\) before it tries to acquire another lock, no problems would have been encountered by the application. Further examples are given in [6].

In the general case, if an application allows both PI and PCE locks, problems will occur if the locks are nested. The chain blocking that occurs with PI means that it is very difficult to analyse a large program to determine the correct ceiling if a PCE lock is acquired by a thread that already holds a PI lock. Further complications occur when threads can dynamically change priorities, ceilings and protocols. To circumvent the above problems, a new

\(^4\)Although lazy priority changing is possible, where the JVM keeps track of the ceilings but only performs the change if contention occurs. This is particularly effective if the priority change requires an RTOS call.
version of the priority ceiling emulation protocol is proposed. The following summarizes the approach.

Objects (and their associated locks) that are governed by PCE have ceilings. However, the ceiling of an object $O$ is set to the maximum of: (1) the highest base priority for any thread that can lock $O$, and (2) the highest ceiling of any object already locked by a thread that is attempting to lock $O$. Now instead of using a thread’s active priority for a ceiling check, the maximum of its base priority and the ceiling of already held PCE locks is used.

For instance, consider threads $T_A$ and $T_B$ with, respectively, low and medium priorities. Both threads share two PCE locks $L_X$ and $L_Y$, with ceiling priorities set to medium. $T_A$ starts executing and acquires $L_X$, having its active priority boosted to the ceiling (medium) priority, eventually acquiring $L_Y$. During entrance in $L_X$, only the base priority of $T_A$ (low) is checked. Then, when entering in $L_Y$, both base priority of $T_A$ (low) and its previous entered lock $L_X$ (medium ceiling) are checked.

2.2 The Full Priority Inheritance Model

Every thread has a base and active priority. The base priority for thread $t$ is set by the programmer and is the priority it is created with, but it can be changed dynamically\(^5\). If $t$ does not hold any locks, then $t$’s base priority equals $t$’s active priority. However, when $t$ holds one or more locks, it is said that $t$ has a set of priority sources. Indeed, the active priority for $t$ (at any time) is the maximum of the priorities associated with all $t$’s priority sources.

The rules for defining the active priority for $t$, based on its priority sources, are defined for when $t$ enters a lock. If the priority sources consists of:

1. Only the thread $t$ itself: the active priority for $t$ is its base priority;

2. Each object locked by $t$ (and governed by a PCE policy): the active priority for $t$ is the maximum value of the ceiling priorities of the locked objects. If $t$ already holds other PCE locks, the ceiling value of the previous lock has to be always lower or equal to the ceiling of the current lock, otherwise a CeilingViolationException is thrown. This exception also is thrown when the base priority of $t$ is greater than the ceiling value of the current lock;

3. Each thread attempting to synchronise on an object locked by $t$ (and governed by a PI policy): the active priority for $t$ is the maximum active priority of all such threads;

4. Each thread attempting to synchronise on an object locked by $t$ (and governed by a PCE-based policy): the active priority for $t$ is the maximum active priority of all such threads.

Rule 1. presents the case where no locks have been acquired and, therefore, no change of priority is needed. In rule 2. we have the modified definition of the PCE protocol, whereas in rule 3. defines the normal PI protocol. However, in order to cope with possible implementations where both PCE and PI protocols might be interacting, rule 4. is defined. Indeed, this rule makes the PCE policy to work as a PI policy. This is defined to avoid the priority inversions that could otherwise occur in the presence of nested synchronization involving a mixture of PCE and PI policies.

The RTSJ also defines specific rules, regarding the addition and removal of priority sources, for the priority-based scheduler when both PI and PCE policies are supported:

\(^5\)As a result, changing the priority of $t$ immediately removes $t$ from the current execution queue and places $t$ at the tail of its new priority.
1. Addition of a priority source: either increases or leaves unchanged \( t \)'s priority. If increased, \( t \) is placed at the tail of its new priority queue;

2. Removal of a priority source: either decreases or leaves unchanged \( t \)'s priority. If decreased, \( t \) is placed at the head of its new priority queue.

An implementation of the RTSJ must perform the following checks when a thread \( t \) attempts to synchronize on an object governed by a PCE policy with ceiling \( \text{ceil} \):

- Thread \( t \)'s base priority does not exceed \( \text{ceil} \);

- The highest ceiling priority of already locked objects by \( t \) (if \( t \) is holding any other PCE locks) does not exceed \( \text{ceil} \).

More formally, if a thread \( t \) whose base priority is \( p_1 \) attempts to synchronize on an object governed by a PCE policy with ceiling \( p_2 \), where \( p_1 > p_2 \), then a CeilingViolationException is thrown in \( t \). A CeilingViolationException is likewise thrown in \( t \) if \( t \) is holding a PCE lock that has a ceiling priority exceeding \( p_2 \). Changes to base priority and changes between the PI and PCE policy (via the method setMonitorControl()) occur immediately. However, to change the policy requires the caller to have acquired the lock.

It is a consequence of the above rules that, when a thread \( t \) attempts to synchronize on an object \( \text{obj} \) governed by a PriorityCeilingEmulation policy with ceiling \( \text{ceil} \), \( t \)'s active priority may exceed \( \text{ceil} \) but \( t \)'s base priority must not. In contrast, once \( t \) has successfully synchronized on \( \text{obj} \) then \( t \)'s base priority may also exceed \( \text{obj} \)'s monitor control policy’s ceiling. Finally it should be noted that when PCE is combined with PI then the overall system will exhibit the same characteristic as a PI system.

Consider again the example used in Section 2.1. Thread \( T_B \) (priority medium) shares PCE lock \( L_X \) with thread \( T_A \) (priority low). The ceiling priority of \( L_X \) is, therefore, set to medium. \( T_B \) executes and acquires lock \( L_X \) (without changing priority because of the ceiling of \( L_X \)), it then performs some action that requires the JVM to acquire an internal PI lock \( L_Y \). \( T_B \) is now preempted by a high priority thread \( T_C \) that executes a JVM operation that requires \( L_Y \). Priority inheritance occurs and thread \( T_B \) has its priority asynchronously boosted to the high priority. Thread \( T_B \) is now executing within lock \( L_X \) at an active priority greater than the ceiling of \( L_X \). According to the new PCE protocol this is NOT an error condition as the base priority of thread \( T_B \) is lower than the ceiling priority of \( L_X \).

### 3 Formal Model

In Section 2, we presented a reformulation of the PCE protocol to enable it to work seamlessly with the PI protocol. In this section we formally model the new protocol, using this model in the next section to analyse the protocol. To correctly formalise the semantics of the new protocol, we propose a modelling architecture using the extended Timed Automata (TA) formalism of the UPPAAL tool\(^7\). We can identify the following basic components for defining the modelling architecture:

- **Thread**: necessary for the definition of verification scenarios, where it enters and exits Locks;

---

\(^6\)This changes the ceiling protocol check to a precondition check instead of an invariant check, when accessing synchronized methods/statements.

\(^7\)The reader can find more information about model checking and the UPPAAL tool in Appendix A.
- **Lock**: defines a shared resource, i.e. synchronized statements or methods in a Java program. The access to these resources is regulated by the synchronisation protocol. A Lock can have a PI or PCE policy;

- **Protocol**: models the synchronisation protocol of the RTSJ, following the rules for including/removing *priority sources*, as described in this paper;

- **Scheduler**: because the synchronisation protocol includes possible movements between priority queues, this component represents a priority-based scheduler.

Further to the components cited so far, we also define another one used to generate verification scenarios for the interaction between threads using locks. This is necessary since we are using model checking as our analysis method. Therefore, using this component for generating scenarios, we can ensure that for a finite amount of threads and shared resources all possible interactions are verified.

In Figure 1 we present the modelling architecture showing the possible interactions between the components. Interactions occur through the use of (global) variables and channels. Specifically, the arrows showing the interactions are labelled with the name of the channel used to activate the desired behaviour in the target automaton, and the necessary parameters (global variables). For instance, when a thread becomes eligible-for-execution, it invokes the Scheduler by synchronising with the *efe* channel and passing its identification number (*tid* variable) as a parameter. As the boxes for Lock and Thread components suggest, it is possible to have one or more (*I..N*) Lock and/or Thread automata in a given verification scenario, whereas there is only one Scenario Generator, Protocol and Scheduler automata.

![Figure 1: Modelling architecture.](image)

### 3.1 Scenario Generator

This is used to generate combinations of verification scenarios having as input a certain number of threads and locks. One automaton of such Scenario Generator is shown in Figure 2 (a). The automaton presented is used to define scenarios for an arbitrary (but finite) number of threads (defined by *NT*) and three different locks. It starts
by non-deterministically assigning (state $S_1$) to the array $nl$ (using as index the id of each thread) a certain number of locks that each thread will be entering in a nested manner. After that, it non-deterministically defines the first lock that each thread will be entering. This is done by assigning values to the array $sl$ (state $S_2$), again using as index the id of each thread. After leaving this state, the automaton signals to all thread automata that they can start (via the broadcast channel $start$) the execution of the generated scenario.

The automaton represents the scenario depicted in Figure 2 (b). All locks are of PI type, the base priority of the threads are defined as $T_0 = 0$, $T_1 = 2$ and $T_2 = 4$ according to the figure. Moreover, the arrows are used to show what nested executions for each thread can occur. In this sense, all of the threads can start entering in either one of the three locks, entering in up to a maximum of two other locks (e.g. possible executions for any of the threads could be $L_2 \rightarrow L_0 \rightarrow L_1$ or $L_0 \rightarrow L_1 \rightarrow L_2$, and so on). For instance, if there was no arrow to the right side of the lock $L_2$ in the figure, the possible executions for the affected thread would be restricted in such way that it would be not possible to do a nested locking from $L_2 \rightarrow L_0$. As explicitly stated in the definition of the scenario, it does not allow threads to sleep inside locks. In Section 4 we define scenarios that allow threads to sleep inside locks (an assumption valid in RTSJ programs) and use the syntax describe in Figure 2 for defining scenarios.

![Figure 2: Scenario generator.](image)

3.2 Thread

In order to evaluate the synchronisation protocol, it is necessary to have execution units entering and exiting locks. Indeed, we provide a generic model (presented in Figure 3), where a thread enters in a certain number of locks (in a nested fashion). The $C$ array includes a clock for each thread that is used to monitor the CPU execution of non-critical and critical (inside a lock) code. Each thread automaton is also composed of another clock, called $s$. This is used to model the start time and the sleeping time (when this behaviour is activated using the variable $cansleep$). After receiving a signal to start (transition $S0 \rightarrow S1$), we have the following behaviour defined for the Thread automaton (see Figure 3):

- $S1 \rightarrow S2$: signal to the scheduler that it has become eligible-for-execution (channel $efe$). This is done non-deterministically in the interval ($s \leq NT$). Because in our model we have a static number of threads executing, the constant $NT$ represents the total number of threads in the model. This way, using the values in this interval as the start time for each thread, we guarantee that in certain executions low priority threads can start before higher priority threads, introducing non-determinism. Also, the lock that will be acquired is set to the variable $cl$;

- $S2 \rightarrow S3$: after one-time unit it tries to acquire the lock (channel $try\_seq\_lock[cl]$);
• **S3 → S2**: a synchronisation with the channel `no_acq_lock[cl]` happens if it cannot acquire the lock. Eventually, if the thread gets scheduled to run in the system ($R == id$), the thread tries to acquire the lock again – transition $S2 → S3$;

• **S3 → S6**: when the lock is acquired, a synchronisation with the channel `acq_lock[cl]` occurs and it executes for one-time unit inside the critical section;

• **S3 → S4 → S5 → S6**: if the sleeping behaviour is active, it could potentially sleep (channel `sleep`) for one-time unit ($s == 1$). Awaking (channel `wake`) and trying to execute the other one-time unit of code inside the critical section;

• **S6 → S7 → S3**: if the thread has not entered in the pre-defined number of locks it was supposed to enter in a nested manner (variable `nol[id]`), this transition is taken and the thread tries to acquire the next lock;

• **S6 → S6**: when the thread acquires more than one lock, it starts exiting the locks (channel `leave_lock[cl]`);

• **S6 → S8**: this transition is taken to exit the first lock that the thread had entered;

• **S8 → S9**: after exiting all locks, the thread executes for one time-unit outside the critical section and stops its execution contacting the scheduler via channel `stop`. It also decreases by one the variable `end`. This variable is used to make sure that all threads have stopped their execution, therefore safely ending the execution of the system – transition $S9 → S9$.

### Figure 3: Thread automaton.

#### 3.3 Lock

The Lock automaton Figure 4 provides basic functions for a thread to enter and exit their critical sections. When the thread tries to acquire a lock (channel `try_acq_lock[cl]`), it can: (i) atomically acquire the lock if no other thread is using the lock (transition $S0 → S1 → S2 → S3 → S0$) and probably have its priority increased depending...
on the type of the lock and the priority sources for the thread (channel `enter_up`); or (ii) become a priority source (transition $S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_0$) for the thread that is using the lock (channel `add_ps`). In either case, the Lock automaton synchronizes with the Protocol automaton (see Figure 5) in order to correctly set the priorities in the system – channels `enter_up` and `add_ps`.

![Figure 4: Lock automaton.](image)

After finishing the necessary actions for changing the priority of the thread or adding priority sources, the Protocol enables the Lock to signal the Thread that it acquired, or not, the lock (channel `continue`). When a thread exits a lock (channel `leave_lock`, transition $S_0 \rightarrow S_7 \rightarrow S_0$), it synchronizes with the Protocol automaton (channel `leave_us`) in order to update the priority sources for the threads in the system affected by that thread and lock.

### 3.4 Protocol

The Protocol automaton (Figure 5) is the main component of the model. It is responsible for defining the behaviour of the protocol proposed in this paper and is composed of four main functions, which are triggered by the Lock automaton using the following synchronisation channels:

- **`enter_up`** called when a thread acquires a lock. At the end of the possible transitions, the Protocol synchronises with the Lock (channel `continue`) thereby enabling the Lock to signal the thread that it has acquired the resource. The possible behaviours are:

  (i) priority is not changed: if the Lock is of PI type (transition $S_0 \rightarrow S_1 \rightarrow S_0$) or PCE type but the thread has one or more priority sources that are threads (transitions $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_0$ or $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_0$). In the latter, where two different transitions are possible, the PCE Lock works as a PI lock because other threads are acting as priority sources. In the transition $S_4 \rightarrow S_0$, the PCE lock is added to the queue of PCE lock sources for the thread (array $SP$);

  (ii) exception is thrown: if the thread has already acquired a PCE lock and the ceiling of this previous lock is greater than the current lock, the transition $S_3 \rightarrow EXCEPTION$ takes place. When the base priority of the thread is greater than the ceiling of the current lock, an exception also occurs (transition $S_6 \rightarrow EXCEPTION$). These are synchronous exceptions;
(iii) priority can be changed: if the thread has already acquired a PCE lock and the ceiling of the current lock is greater than the previous PCE lock ($S_3 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_0$) or this is the first lock and also its ceiling is greater than the current priority of the thread ($S_2 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_0$). However, if the current active priority of the thread is equal to the ceiling, no priority change is carried out (transition $S_7 \rightarrow S_9 \rightarrow S_0$);

- **add_ps**: a thread synchronises with this channel when it tries to enter in a shared resource that is already in use. If the active priority of the thread trying to acquire the lock is lower or equal than the current thread using the lock, the result of this function is that the thread will be blocked and become a priority source for the thread using the lock (channel block in transition $S_0 \rightarrow S_10 \rightarrow S_11 \rightarrow S_14 \rightarrow S_15 \rightarrow S_0$). However, if the active priority of the thread trying to acquire the lock is greater than the current thread using the lock, as well as becoming a priority source, the priority of the thread using the lock is boosted and, that thread is a priority source to other threads, the priorities of these threads are boosted as well (transition $S_{12} \rightarrow S_{13}$). This loop ends when a thread that is not a priority source is reached or the priority of the thread is greater or equal to the new priority – transition $S_{12} \rightarrow S_{14}$;

- **leave_ps**: called when the thread leaves the lock. It is used to: update the priority sources related to the thread and the lock, reset the queue of the lock to zero and trigger a new scheduling. The new scheduling is triggered to ensure that the thread with highest priority (especially if there are threads in the lock queue) will be selected for execution. Specifically, three initial paths are possible depending on the size and type of the lock object in question: (i) there are no threads in the queue and the lock is of PI type (transition $S_{16} \rightarrow S_{22}$); (ii) there are no threads in the queue and the lock is of PCE type, in which case the lock is removed from the lock sources for the thread (transition $S_{16} \rightarrow S_{22}$); (iii) there is one or more threads in the queue of the lock (transition $S_{16} \rightarrow S_{17}$). When the last transition is taken (iii), the following behaviour occurs. If the lock is of PCE type, it is removed from the thread’s PCE type lock sources (transition $S_{17} \rightarrow S_{18}$). In the transition $S_{18} \rightarrow S_{19} \rightarrow S_{20} \rightarrow S_{21}$ the Protocol removes the threads in the lock queue from the list of priority sources of the thread that was using the lock. Then, it unblocks all the threads related to the lock (transition $S_{19} \rightarrow S_{22}$). The next transitions also apply for the paths (i) and (ii) presented before and are used to define the new priority of the thread leaving the lock. If the thread has no priority sources (either other threads or PCE locks) the transition $S_{22} \rightarrow S_{25}$ is taken. However, if the thread has as priority sources other locks but no threads the transition $S_{22} \rightarrow S_{23} \rightarrow S_{25}$ are used to find the highest priority of the PCE lock sources. Finally, if the thread has as priority sources other threads the transitions $S_{22} \rightarrow S_{24} \rightarrow S_{25}$ are used to find the highest priority of the thread sources. In the end (transition $S_{25} \rightarrow S_{0}$), a new scheduling occurs.

### 3.5 Scheduler

As previously stated, we need to model a priority-based scheduler in order to analyse the behaviour of the synchronisation protocol. The proposed scheduler defined in this section only provides the necessary functions for checking the correctness of the synchronisation protocol. The automaton for modelling the scheduler, called Scheduler automaton, is shown in Figure 6.

In order to correctly define the behaviour of a priority-based scheduler, the automaton uses a queue (array $Q$) for representing the priority queues for the threads that are waiting/running or blocked in the system. Every entry in this array is composed by a pair: the id of the thread in the priority queue and either the value 1 (meaning that
Figure 5: Protocol automaton.

the thread is not blocked) or 0 (the thread is currently blocked). The Scheduler automaton stays continuously in state $S_0$ awaiting synchronisations that activated different behaviours:

- **New scheduling (channel schedule, transitions $S_0 \rightarrow S_3 \rightarrow S_0$):** the priority queues are searched in order to find the id of the thread with the highest eligibility in the system;

- **Schedule a thread (channel efc, transitions $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_0$):** adds the thread id ($tid$) to its current priority queue and reschedules;

- **Awake a thread that was sleeping (channel wake, transitions $S_0 \rightarrow S_{19} \rightarrow S_3 \rightarrow S_0$):** removes the thread ($tid$) from the blocked state and makes a new scheduling:
- **Deschedule the thread** (channel `stop`, transitions `S0 → S4 → S5 → S3 → S0`): removes the thread id (`tid`) from the priority queue and makes a new scheduling;

- **Put the thread to sleep** (channel `sleep`, transitions `S18 → S3 → S0`): puts the thread (`tid`) in the blocked state\(^8\) and makes a new scheduling;

- **Add the thread to the head of a given priority queue** (channel `add_head`, transitions `S0 → S6 → S7 → S8 → S9 → S10 → S0`): adds the thread id (`tid`) to the head of its new priority queue (`tnp`) and removes the thread id from its previous queue;

- **Add the thread to the tail of a given priority queue** (channel `add_tail`, transitions `S0 → S11 → S12 → S13 → S9 → S10 → S0`): adds the thread (`tid`) to the tail of its new priority queue (`tnp`) and removes the thread id from its previous queue;

- **Block a thread** (channel `block`, transitions `S0 → S14 → S0`): blocks the corresponding thread (`tid`);

\(^8\)In Java [9], a call to the `sleep()` method temporarily ceases the execution of the thread for the specified duration. The thread does not lose ownership of any monitor locks.

Figure 6: Scheduler automaton.
• Unblock one or more threads (channel `unblock`, transitions $S_0 \rightarrow S_{15} \rightarrow S_6 \rightarrow S_{17} \rightarrow S_0$): unblocks thread(s) that were in the queue for the correspondent lock (`lid`).

When a scheduling happens, the id of the thread selected for running the CPU is stored in the $R$ variable. This enables the selected thread to execute its next action, according to the behaviour defined in the Thread automaton (see Section 3.2).

## 4 Model Analysis

In this section we formally explore the behaviour of the synchronisation protocol proposed in this paper. Since we are using model checking as our analysis method, we need to define different scenarios that provide the executions we are interested in analysing. From an analysis point of view, we classify our properties into two different types: consistency and behaviour. Consistency properties are used to ensure that we have corrected modelled the protocol. Behaviour properties are used to explore in more detail different behaviours associated with the protocol.

### Verification scenarios:

The generation of all possible scenarios for the protocol using model checking is not feasible during the verification process. Even if it could, such an approach would generate many scenarios that would never occur in a real system. Therefore, we model some scenarios aiming to capture the major behaviours we are interested in analysing. These scenarios are depicted in Figure 7, where we propose seven different scenarios for our formal analysis. The syntax for the definition of the scenarios follows the one explained in Section 3.1. Moreover, when a PCE lock is defined (Figure 7), the number in its right side is the ceiling priority for that lock.

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1} L_2)$</td>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1} L_3)$</td>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1})$</td>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1} L_2)$</td>
</tr>
<tr>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
</tr>
<tr>
<td>$T_2 = (L_3)$</td>
<td>$T_2 = (L_3)$</td>
<td>$T_2 = (L_3)$</td>
<td>$T_2 = (L_3)$</td>
</tr>
<tr>
<td>$T_3 = 0$</td>
<td>$T_3 = 0$</td>
<td>$T_3 = 0$</td>
<td>$T_3 = 0$</td>
</tr>
<tr>
<td>$T_4 = 2$</td>
<td>$T_4 = 2$</td>
<td>$T_4 = 2$</td>
<td>$T_4 = 2$</td>
</tr>
<tr>
<td>Sleep behaviour is deactivated.</td>
<td>Sleep behaviour is deactivated.</td>
<td>Sleep behaviour is deactivated.</td>
<td>Sleep behaviour is deactivated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario 5</th>
<th>Scenario 6</th>
<th>Scenario 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1} L_2)$</td>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1})$</td>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1} L_2)$</td>
</tr>
<tr>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
<td>$T_1 = (L_2 \xrightarrow{0})$</td>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
</tr>
<tr>
<td>$T_2 = (L_3)$</td>
<td>$T_2 = (L_3)$</td>
<td>$T_2 = (L_3)$</td>
</tr>
<tr>
<td>$T_3 = 0$</td>
<td>$T_3 = 0$</td>
<td>$T_3 = 0$</td>
</tr>
<tr>
<td>$T_4 = 2$</td>
<td>$T_4 = 2$</td>
<td>$T_4 = 2$</td>
</tr>
<tr>
<td>Sleep behaviour is deactivated.</td>
<td>Sleep behaviour is activated.</td>
<td>Sleep behaviour is deactivated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario 8</th>
<th>Scenario 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1})$</td>
<td>$T_0 = (L_0 \xrightarrow{0} L_1 \xrightarrow{1})$</td>
</tr>
<tr>
<td>$T_1 = (L_2 \xrightarrow{0})$</td>
<td>$T_1 = (L_2 \xrightarrow{0} L_3)$</td>
</tr>
<tr>
<td>$T_2 = (L_3)$</td>
<td>$T_2 = (L_3)$</td>
</tr>
<tr>
<td>$T_3 = 0$</td>
<td>$T_3 = 0$</td>
</tr>
<tr>
<td>$T_4 = 2$</td>
<td>$T_4 = 2$</td>
</tr>
<tr>
<td>Sleep behaviour is deactivated.</td>
<td>Sleep behaviour is deactivated.</td>
</tr>
</tbody>
</table>

Figure 7: Scenarios for the analysis.

In both Scenarios 1, 2, 5 and 7 we expect not to have deadlocks. For Scenario 3 this assumption is not true, due to the possible combinations of entrance and exits of PI locks. Using Scenario 4 we depict the occurrence of exceptions for PCE locks. Scenario 6 is used to illustrate problems with the use of the sleep method found in Java.

---

9We used only two locks in Scenario 6 (three locks generated state explosion problems) because the complexity added to the analysis when the sleep behaviour is activated.
The last Scenario 7 is used to explore the behaviour of using both PI and PCE policies in the same system.

**Consistency analysis:** Consistency analysis is used to ensure that the model presented in the last section correctly incorporates the definitions of the protocol, with respect to both PI and PCE lock behaviours. We start analysing the change of priorities that can occur with the interaction of threads entering and exiting locks. According to the protocol, the behaviour for such property is defined via the addition of priority sources.

**Property 1.** When a thread ($T_{id}$) acquires a lock. If the lock was of PI type and higher priority threads try to enter this lock, new thread priority sources are added to $T_{id}$ and its active priority becomes greater than its base priority. Otherwise, if the lock was of PCE type, a new PCE lock source is added to $T_{id}$ and its active priority is changed if $T_{id}$ active priority is lower than the lock’s priority ceiling.

In the context of PI locks we are dealing with the addition of thread sources (array $STN[T_{id}]$ in the model). For PCE locks we are looking at the addition of PCE locks as priority sources (array $SPN[T_{id}]$ in the model). To specify this property we use the pattern ($\Phi \rightarrow \phi$). The left side of the implication specifies that the thread has either threads or PCE locks as priority sources ($STN[T_{id}] > 0 \lor (SPN[T_{id}] > 0)$). At the right side we make sure that the active priority of the thread is greater than its base priority ($T[T_{id}][TAP] > T[T_{id}][TBP]$).

In order to check this property, we substitute $T_{id}$ for each thread in the scenarios, except for Scenarios 3 and 4. Executing different verification runs and having a true result in all of them confirms the properties. For Scenarios 3 and 4 we only execute this property with the substitution of $T_{id}$ for the lowest priority thread in the scenario (zero), reaching a true result. Specifically, for the other threads of the scenarios we do not verify this property for the following reason. In Scenario 3 it is possible that a low priority thread becomes source of a higher one (in executions that can potentially lead to a deadlock situation, see Property 4). Because the active priority of the higher thread is not changed, the right-side of the implication is false. In Scenario 4 we have the case where a thread enters in a PCE lock (satisfying the left-side of the implication) and, because its base priority is greater than the ceiling priority of the lock, an exception happens (see Property 3) and the priority is not changed (right-side of the implication is false). The next property focuses on ensuring that mutual exclusion is guaranteed by the model.

**Property 2.** Between the period that a thread enters and exits a lock, no other thread can enter in the lock.

We specify this safety property by inserting a global array ($threadin[NL]$) in the model that has the size of all locks in the given scenario (we use the id of the locks ($L_{id}$) as index to this array). We start with all values of $threadin[NL]$ set to zero, incrementing it when a thread enters in a lock (transition $S0 \rightarrow S1$ for Lock automaton) and decrementing when a thread exits a lock (transition $S0 \rightarrow S7$ for Lock automaton). This way we define the property: $A[]$ ($threadin[L_{id}] >= 0$) && ($threadin[L_{id}] <= 1$). The property is verified for all locks ($L_{id}$) in the scenarios. Having those properties verified to a true result (see Table 1), now we focus on analysing some behaviours of the protocol.

**Behaviour analysis:** Behaviour analysis is used to explore certain behaviours of the protocol. In the first property, we start analysing that it is not possible to have exceptions being raised.

**Property 3.** An exception is never generated.

To specify this property we define the safety formula: $A[]$ (not Protocol.EXCEPTION) – making sure that the EXCEPTION state is never reached. This property mainly concerns PCE locks and, when evaluated in the scenarios, had a false result only for Scenario 3 due to the different ways threads can nest locks and that ceiling priorities for the locks are defined. The exception generated refers to a synchronous exception, rather than an asynchronous exception. One trivial counter-example that shows the generation of the synchronous exception is shown in Figure 8.

Another behaviour property looks at the important issue of not having deadlock situations.
Property 4. No deadlock situations can occur in the defined scenarios.

Using the deadlock keyword available in the UPPAAL tool, we can easily specify this formula, leading to the specification $A[\text{not deadlock}]$. During verification, this property was true for Scenarios 1, 2, 5 and 7. Specifically, with the true result in the verification of Scenario 5 (where ceilings are correctly defined), we ensure that the basic properties of the PCE protocol, deadlock freedom and single blocking, is maintained in the modified protocol presented in this paper. We ensure single blocking because in Scenario 5, without having single blocking, certain executions would inevitably lead to deadlock situations, which do not occur. For Scenarios 3, 4 and 6 the property was false. In Scenario 3 the use of PI locks leads to a deadlock situation. The generation of an exception is the reason for the deadlock in Scenario 4. Finally, in Scenario 6 the use of PCE locks when threads can sleep inside locks leads to the deadlock – a situation prone to happen in such scenarios [12]. In Figure 9 we show the counter-example for the property using Scenario 6.

In the counter-example, thread $T_2$ with highest priority starts executing and enters lock $L_0$, increasing its priority to 5. Instead of continuing the execution, $T_2$ sleeps and enables thread $T_1$ to start executing and entering in lock $L_1$. When $T_2$ wakes, it tries to enter in lock $L_1$ that has been locked by $T_1$. The deadlock situation has been formed due to the sleep semantics found in the Java language.

Property 5. It is not possible for thread $T_0$ to have its active priority raised above priority 4.

The property is defined as $(A[] (T[T_0][TAP] < 4))$. This ensures that in all states the active priority of thread $T_0$ is lower than or equal to four. Looking at the definition of the scenario and its interactions, at first sight the maximum active priority of thread $T_0$ should be 4. Nevertheless, this is not true and we show in Figure 10 the
counter-example that illustrates the main feature of the protocol defined in this paper.

Initially, thread $T_0$ enters in PI-type lock $L_1$. After entering in the lock, thread $T_1$ starts executing and enter in lock $L_0$. It is then preempted by thread $T_2$ which tries to enter in lock $L_1$, increasing the priority of $T_0$ which then enters in lock $L_2$. Finally, the highest priority thread $T_3$ tries to enter in lock $L_0$ and increases the priority of $T_1$. $T_1$ now tries to enter in PCE lock $L_2$, already locked by $T_0$. According to the protocol definition, the active priority of $T_0$ is boosted (lock $L_2$ act as a PI lock) and the system can finish. Such interaction without this protocol would not have made possible the correct execution of the system and an asynchronous exception would have been generated as soon as thread $T_1$ tried to enter in the lock $L_2$ with its active priority greater than the ceiling priority.

**Verification results:** In Table 1 we condensed the verification results for both consistency and behaviour properties. The syntax used to describe the results is: (Result (T for true or F for false), Time (approximated in sec:msec) and Memory space (approximated in MB)). In the verifications we used an Intel Pentium 4 1.9 GHz machine running the Slackware Linux 8 Operating System with 1 Gb of RAM. UPPAAL tool version 3.4.11 was used with its aggressive state space reduction option set.

![Figure 10: PCE lock acting as a PI lock.](image)

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
<th>Scenario 5</th>
<th>Scenario 6</th>
<th>Scenario 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T, (\approx 00:52, \approx 28MB))</td>
<td>(T, (\approx 00:62, \approx 37MB))</td>
<td>(T, (\approx 23:81, \approx 1089MB))</td>
<td>(T, (\approx 08:22, \approx 267MB))</td>
<td>(T, (\approx 18:13, \approx 560MB))</td>
<td>(T, (\approx 01:35, \approx 64MB))</td>
<td>(F, (\approx 01:19, \approx 55MB))</td>
</tr>
<tr>
<td>2</td>
<td>(T, (\approx 00:42, \approx 23MB))</td>
<td>(T, (\approx 00:63, \approx 24MB))</td>
<td>(T, (\approx 21:84, \approx 753MB))</td>
<td>(T, (\approx 00:42, \approx 17MB))</td>
<td>(T, (\approx 00:52, \approx 23MB))</td>
<td>(T, (\approx 20:87, \approx 753MB))</td>
<td>(T, (\approx 01:29, \approx 24MB))</td>
</tr>
<tr>
<td>3</td>
<td>(T, (\approx 00:42, \approx 17MB))</td>
<td>(T, (\approx 00:52, \approx 23MB))</td>
<td>(T, (\approx 21:84, \approx 753MB))</td>
<td>(T, (\approx 00:97, \approx 24MB))</td>
<td>(T, (\approx 01:29, \approx 24MB))</td>
<td>(F, (\approx 37:66, \approx 567MB))</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(T, (\approx 00:97, \approx 24MB))</td>
<td>(T, (\approx 01:29, \approx 24MB))</td>
<td>(T, (\approx 37:66, \approx 567MB))</td>
<td>(T, (\approx 01:35, \approx 31MB))</td>
<td>(T, (\approx 01:35, \approx 31MB))</td>
<td>(T, (\approx 01:35, \approx 64MB))</td>
<td>(T, (\approx 03:43, \approx 65MB))</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(F, (\approx 01:19, \approx 55MB))</td>
</tr>
</tbody>
</table>

5 Conclusions and Final Remarks

Whilst there is literature abound on priority inheritance protocols (a good review is given by Liu [11]), all of it focuses on the properties of a particular approach. There is none that addresses the combined use of multiple
protocols. The main international standards that address the use of more than one type of priority inheritance protocol are POSIX and Ada. Similar to the RTSJ, both allow the use of PI and PCE protocols in programs (although with a different level of support). Unfortunately, they do not consider the use of a mixture of PI and PCE locks in the same system. The unwritten assumption seems to be that an application will use one approach or the other.

This paper has illustrated why multiple approaches are likely to be needed in large real-time systems, and we have shown that asynchronous exceptions can be generated under normal operational circumstances. To solve this problem, we proposed a new version of the PCE protocol that allows a more harmonious integration with the PI protocol. We have used model checking technology to analyse the behaviour of the protocol. During the analysis we have showed that for all modelled scenarios, the protocol avoids unwanted asynchronous exceptions. We also reconfirm the danger of self suspension whilst holding a lock (for example by calling the sleep() method) in RTSJ programs. This can lead to deadlock situations even when only PCE locks are used. This undermines one of the main advantages of using PCE.

In addition, the proposed formal priority inheritance model can be modified to explore the behaviour of other priority inheritance protocols in the literature. The model is available on our web site (http://www.cs.york.ac.uk/rtos/osantos) for this purpose. With respect to the maximum blocking time, the protocol is equivalent to the PI protocol when: (i) only PI locks are used or; (ii) there is interaction between PI and PCE locks. When only the PCE lock is used, the maximum blocking time equals that of the original PCE protocol.

Currently, we do not model any kind of dynamic changes that may occur in the system due to program behaviour, including the change of thread priorities and the change of lock protocols. This adds considerable complexity to the formal model and can easily generate state explosion problems. As future work, we are consider different ways to manage this complexity and hence extend the model.

Acknowledgements

The authors gratefully acknowledge the contributions of David Holmes, Peter Dibble, Rudy Belliardi and Doug Locke to some of the work presented in this paper.

Appendix A Model Checking and the UPPAAL Tool

Model checking comprises a technique for the automatic verification of concurrent systems. The basic idea is to build a finite model of the system and exhaustively check that a desired property holds for that model [7]. Normally, if a property does not hold for a model, a counter-example is shown – an execution path generated from the model showing where the property was evaluated to false. The exhaustive check of the model is carried out by a model checking algorithm, therefore, model checking is a verification technique that requires tool support.

In general, the process of applying model checking to a system consists of three different phases [8]: (i) system modelling (build a model of the system using a modelling language); (ii) property specification (define properties of the system to be checked against its model, typically using temporal logic); and (iii) formal verification (the process of exhaustive verification itself). In the real-time area, the main modelling language, which includes tool support for model checking, is the Timed Automata (TA) formalism [1]. Among the range of tools in the literature we can find reachability analysis checkers, like UPPAAL [10], and full temporal-logic model checkers, like KRONOS [5]. Due to its easy of use, continuous implementation and support, and efficient verification techniques, in this paper we use UPPAAL for the formal analysis.
Here we only give a brief overview of the UPPAAL features – the reader is referred to [2] for a more detailed explanation. The modelling language of UPPAAL extends TA with features like [2]: **bounded integer variables** that can be read, written and are subject to arithmetic operations; **constants**, which cannot be modified and are used to assign a name to an integer value; **synchronization channels**, where a sender automaton (c!) can non-deterministic synchronize with a receiver automaton (c?); and **committed locations**, where if a transition occurs and the destination locations is committed, the next transition must occur atomically without delay and without interleaving with other transitions in the model.

For the specification of properties, UPPAAL provides a simplified version of the Timed Computation Tree Logic (TCTL) – increasing the efficiency of the verification, but as the down side decreasing the expressability of its property specifications. Basically, property specifications in UPPAAL consist of a path and a state formula. The path formula is used to reason about possible execution paths of the model, and the state formula refers to specific states. Properties are classified as state, reachability, safety and liveness [2]. A state property refers to an expression (φ) evaluated at a specific state. The **deadlock** keyword is provided and defines a state where no possible transitions can occur in the model. Reachability properties are used to check if a given state is possibly reachable (E <> φ) – we use the syntax found in [2] for the specification of properties. A safety property is used to guarantee that “something bad will never happen”, i.e. the property is valid in every state of all execution paths (A[] φ) or at least one or more execution paths (E[] φ). Finally, liveness properties are used to guarantee that “something good will eventually happen”. They can be specified using A <> φ, or the pattern Φ ⇝ φ (which means that whenever Φ is satisfied, φ will be satisfied).

**References**