A Verified Implementation of Priority Monitors in Java

Ángel Herranz and Julio Mariño[*]

Babel research group
Universidad Politécnica de Madrid

Abstract. Java monitors as implemented in the `java.util.concurrent.locks` package provide *no-priority nonblocking* monitors. That is, threads signalled after blocking on a *condition* queue do not proceed immediately, but they have to wait until both the signalling thread and possibly some of the others which have requested the lock release it. This can be a source of errors (if threads that get in the middle leave the monitor in a state incompatible with the signalled thread re-entry) or inefficiency (if repeated evaluation of preconditions is used to ensure safe re-entry). A concise implementation of priority nonblocking monitors in Java is presented. Curiously, our monitors are implemented on top of the standard no-priority implementation. In order to verify the correctness of our solution, a formal transition model (that includes a formalisation of Java locks and conditions) has been defined and checked using Uppaal. This model has been adapted to PlusCal in order to obtain a formal proof in TLA independent of the number of threads.

Keywords: Monitors, Java, model checking, priority, nonblocking, TLA, *PlusCal*.

1 Introduction

A *model-driven* approach to the development of concurrent software [7] advocates the use of high-level, language-independent entities that can be subject to formal analysis (e.g., to early detect risks due to concurrency in reactive, critical systems) and that are later translated into a concrete programming language by means of safe transformations. This, rather than the unrestricted use of the concurrency mechanisms in the target language, aims at improving portability (always a must in embedded systems) and preventing hazards due to the use of error-prone or poorly documented primitives. We have used this approach for teaching concurrency for more than fifteen years at our university, first using Ada95 [3], and now Java, which is certainly lacking in many aspects when compared to the former.

[*] Authors partly funded by grants DESAFIOS10 (TIN2009-14599-C03-00) from the Spanish Ministry of Science and Innovation, PROMETIDOS (P2009/TIC-1465) from the Madrid Regional Government and COST Action IC0701 on Formal Verification of Object Oriented Software.

CADT Counter
ACTION Inc: $C\_Type[io]$
ACTION Dec: $C\_Type[io]$

SEMANTICS
TYPE: $C\_Type = \mathbb{Z}$
INVARIANT: $\forall\, c \in C\_Type.c \geq 0$
INITIAL(c)**:** $c = 0$

CPRE: *True*
Inc(c)
POST: $c^{\text{out}} = c^{\text{in}} + 1$

CPRE: $c > 0$
Dec(c)
POST: $c^{\text{out}} = c^{\text{in}} - 1$

```java
class Counter {
    final Lock mutex =
        new ReentrantLock(true);
    final Condition strictlyPos =
        mutex.newCondition();
    private int c = 0;
    public void inc() {
        mutex.lock();
        c++;
        // signal some pending dec
        strictlyPos.signal();
        mutex.unlock();
    }
    public void dec() {
        mutex.lock();
        // check the CPRE now
        if (c == 0) {
            try {strictlyPos.await();}
            catch(Exception e) {}
        }
        // if we are here, that means
        // that c > 0
        c--;
        // it CAN be proved that
        // no more signals are needed
        mutex.unlock();
    }
}
```

Fig. 1. A minimal example showing the risks of Java's no-priority locks and conditions.

Indeed, the early mechanisms for thread synchronization provided by Java were so limited that one of the pioneers of concurrent programming, Brinch Hansen, wrote a famous essay [6] alerting on the risks of their usage. As a response to these adverse reactions, more powerful concurrency mechanisms were introduced in later versions of Java, although without being explicitly presented as realizations of the *classical* ones.

In this work we will focus on the *locks & conditions* library (`java.util.concurrent.locks`), that can be seen as an attempt to implement *monitors* (actually, Brinch Hansen's contribution) in Java. Figure 1 shows an (apparently correct) implementation of a shared resource, specified in a formal notation, using the aforementioned library. On the left side of the figure, a shared counter is specified as a kind of abstract data type with implicit mutually exclusive access and an interface that specifies the only operations (**Inc** and **Dec**) that can be used to modify its state. Moreover, these *actions* are pure and transactional. In this case, we also state the invariant that the counter cannot be negative at any time, and specify the conditional synchronization (CPRE, *concurrency precondition*) for **Dec**, that ensures the invariant after the execution of any of the two actions.

To the right, we see an implementation of the resource specification as a Java class using locks and conditions. Class `Counter` encapsulates an integer variable `c` to represent the counter. Also, a lock object `mutex` is used to ensure execution of the actions in mutual exclusion. Finally, a condition object `strictlyPos` is used to implement the conditional synchronization of threads invoking `dec()`, i.e., to put them to sleep when the action is invoked with the counter set to 0.

The code has been written following the principle that only 0 or 1 `await()` calls are executed during an action, and the same applies to `signal()`. This coding guideline really pays off when applied to concurrency problems of greater significance. Basically, it reduces the context switching points inside the code for an action to (at most) one (the execution of `await()`), dividing it into two sequential fragments, which facilitates formal reasoning. It is the responsibility of the signalling thread (in this case, any thread executing `Inc`) to ensure that the signalled thread wakes up in a safe state. In this case, there is nothing to check, as `strictlyPos.signal()` is placed right after incrementing the counter and thus, if the signalled thread resumes execution right after the signalling one executes `mutex.unlock()` it is safe to decrement the counter. That justifies the comment above the line containing "`c--;`".

Unfortunately, this code is unsafe. Why? The reason is that Java's implementation is an example of *no-priority* signalling [2]. That means that threads signalled after blocking on a *condition* queue do not proceed immediately, but they have to wait until both the signalling thread and also other threads that might have requested the lock release it. In other words, when the decrementer thread resumes execution it could be the case that other decrementer thread was already queued on `mutex` and set the counter to 0 right before. Moreover, given that the execution of the `inc()` and `dec()` methods takes very little time, the probability that this scenario happens is relatively low, which makes this the kind of error that can go unnoticed for thousands of runs.

The official documentation for the `Condition` class leaves considerable room for the operational interpretation of the `await()` and `signal()` methods in future implementations and, in fact, the no-priority behaviour is not really implied by the natural language specification of the API. There is, however, a generic recommendation to enclose the calls to `await()` inside invariant-checking loops, motivated by the possibility of *spurious* wakeups. Although the use of such loops can be seen as a defensive coding technique that can avoid hazards in general, we think that forcing the application programmer to use them can be quite unsatisfactory in many situations as testing the concurrency precondition repeatedly can increase contention of certain concurrent algorithms intolerably.

The use of the 0-1 coding scheme allows for low-contention, deterministic implementations of shared resources. Also, in [2] the reader can find a classification and comparison of many monitor variations, concluding that, in general, priority implementations are less error-prone. With this in mind, we decided to reuse as much from the existing reference implementation of condition objects and turn them into a concise, priority implementation of monitors. Our proposal is described in the following section.

```
package es.upm.babel.cclib;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class Monitor {
   private Lock mutex = new ReentrantLock(true);
   private Condition purgatory = mutex.newCondition();
   private int inPurgatory = 0;
   private int pendingSignals = 0;

   public Monitor() {
   }

   public void enter() {
      mutex.lock();
      if (pendingSignals > 0 || inPurgatory > 0) {
         inPurgatory++;
         try { purgatory.await(); }
         catch (InterruptedException e) { }
         inPurgatory--;
      }
   }

   public void leave() {
      if (pendingSignals == 0 && inPurgatory > 0) {
         purgatory.signal();
      }
      mutex.unlock();
   }

   public Cond newCond() {
      return new Cond();
   }

   inner class Cond goes here.
}
```

Fig. 2. Java source for priority monitors (class `Monitor`).

2 Implementing Priority Monitors with Java's No-priority Monitors

Figures 2 and 3 show a stripped-down version of the source code for `Monitor.java`, our implementation of priority monitors. Due to space limitations, comments and exception managers have been omitted. A full version, including comments and a couple of extra operations can be found at http://babel.ls.fi.upm.es/software/cclib.

In addition to the `Monitor` class, `Monitor.java` defines the `Cond` class. Both classes are intended to be a replacement for the original `Lock` and `Condition` classes. Figure 3 contains the code for class `Cond`, and the rest of `Monitor.java` is

```
public class Cond {
   private Condition condition;
   private int waiting;
   private Cond() {
      condition = mutex.newCondition();
      waiting = 0;
   }

   public void await() {
      waiting++;
      if (pendingSignals == 0 && inPurgatory > 0 ) {
         purgatory.signal();
      }
      try { condition.await(); }
      catch (InterruptedException e) { }
      pendingSignals--;
   }

   public void signal() {
      if (waiting > 0) {
         pendingSignals++;
         waiting--;
         condition.signal();
      }
   }
}
```

Fig. 3. Java source for priority condition variables (class `Cond`).

shown in Figure 2. However, this separation is a mere convenience for displaying the code: the implementation of class `Cond` requires access to the state variables in class `Monitor`.

The functionality provided by the class is similar to that of locks and conditions. Method `enter()` provides exclusive access to the monitor, like method `lock()` in class `Lock`. If one thread got exclusive access to a monitor object m, susbsequent calls to m`.enter()` from other threads will force these to wait in m's queue. The monitor is released by invoking m`.leave()` (analogous to `unlock()`). If there are threads waiting at m's entry queue, executing m`.leave()` will give control to the first thread in the queue.

Condition queues associated with m are created by invoking m`.newCond()` rather than calling the constructor of class `Cond`. This is similar to the behaviour of `newCondition()` in class `Lock`. Instances $c, c' \ldots$ of class `Cond` associated with a given monitor object m are managed like condition variables. A thread that invokes c`.await()` after m`.enter()` is blocked unconditionally and put to wait in a queue associated with c. Also, executing c`.await()` releases m, so that other threads can get exclusive access to m by executing m`.enter()`. As in the case of `leave()`, if there are threads waiting at m's entry queue, executing m`.await()` will give control to the first thread in queue.

A thread that executes c`.signal()` after getting exclusive access to m will continue executing (signal-and-continue policy) but it is guaranteed that if there is a thread waiting on c, it will be awakened *before* any process waiting on m's entry queue. This is where the behavior differs from that of signals in the standard locks and conditions package, where signalled threads are requeued at the tail of the lock's entry queue.

The implementation is done on top of the existing implementation of locks and conditions, without reimplementing their functionality say, by using low-level concurrency mechanisms such as semaphores, nor accessing the Java runtime in order to "magically move" threads from one queue to another.

The technique used to simulate the effect of moving signalled threads from the head of the condition queue to the head (rather than the tail) of the monitor's entry queue is to *flush* the contents of the latter until the signalled thread becomes the first. This is achieved by letting some threads get in the monitor, but only to make them await in a special condition queue devised for that purpose, and which we call "the purgatory". Of course, these threads will have to be eventually awakened and allowed to get access to the monitor in the right sequence. As they have been moved to a condition variable, signalling them will bring them back to the lock's entry queue, so a little care is needed to ensure that the whole system progresses appropriately. Two global counters are responsible for this.

Variable `inPurgatory` counts the number of threads that have been sent to the `purgatory` condition queue and have not been given access to the monitor yet – even if they have already been signalled. Variable `pendingSignals` counts how many threads that have been signalled (in regular condition variables, not `purgatory`) are yet waiting for re-entry.

Method `enter()` starts executing `lock()` on the monitor's internal lock object, called `mutex` (line 16). However, if `pendingSignals` is not zero, this thread should give way to some signalled thread that was waiting for monitor re-entry later in the queue associated with `mutex`, and thus has to execute a `purgatory.await()`, updating counter `inPurgatory` before and after (lines 17–22). The same is done if other threads have been sent to the purgatory earlier and have not re-entered yet (`inPurgatory` $>$ `0`).

The implementation of method `leave()` is quite symmetric. It ends by invoking `mutex.unlock()` (line 29), but before, a chance to re-gain access to the monitor must be given to threads moved to the purgatory (line 27), only if there are no signalled threads queued for monitor re-entry (line 26).

Method `newCond` is implemented just by invoking the standard constructor of class `Cond` (line 33). Association to a given monitor object is just implicit in the visibility of the monitor's state variables that the `Cond` object has. A `Cond` object is basically a pair of a `Condition` object associated with `mutex` and a counter `waiting` that keeps track of the number of threads waiting on it (lines 38–43).

The core of method `await()` is the corresponding call on its condition object (line 50) but, before, counter `waiting` is incremented (line 46) and the right to execute inside the monitor is given to threads in the purgatory only if there are no pending signals (lines 47–49). Finally, if the thread that has invoked `await()` reaches line 52, the whole signalling procedure has been successful including monitor re-entry, so `pendingSignals` must be decremented accordingly.

Finally, method `signal()` checks whether the queue size is nonzero (line 56) and if so increments `pendingSignals` (line 57), as monitor re-entry for the signalled thread will be incomplete, decrements `waiting` (line 58) and executes `signal()` on the condition object (line 59). Executing `signal()` on an empty queue has no effect.

Although the implementation of the `Monitor` and `Cond` classes is quite concise, the interplay between the different threads and synchronization barriers, governed by the different counters is, admittedly, complex and hard to follow. After testing the class on a number of applications specifically devised to force errors related to incoming threads getting in the way of signalled ones, we still felt that a more convincing proof of the correctness of our implementation was necessary.

3 A Formal Model of Java Locks and Conditions

As a first step towards a formal argumentation of the correctness of our implementation of priority monitors, we decided to define a transition model for the underlying implementation of locks and condition variables.

The model has been defined as a network of (parametric) automata in Uppaal [1]: one automaton for modelling the concurrency mechanism, i.e., the lock object and its associated conditions, and one automaton per thread that makes use of it. While we give a complete formalization of the concurrency mechanism, unrelated thread code is abstracted away by just showing the interaction with the lock and the conditions.

Each thread has its own thread identifier, *pid*, and so do condition variables (*cid*). All interaction between the different automata takes place via communication channels:

- lock[*pid*] is the channel the thread *pid* uses when it needs to lock the lock.
- await[*cid*][*pid*] is the channel the thread *pid* uses when it needs to wait in condition *cid*.
- signal[*cid*][*pid*] is the channel a thread uses to signal condition *cid*.
- lock_granted[*pid*] is the channel the concurrency mechanism uses to grant a lock to the thread *pid*.
- lock[*pid*] is the channel the thread *pid* uses to unlock the lock.

Let's start with the abstract model for the threads, since, in our view, it is the most didactic way to present the whole model.

3.1 Modelling the Threads

Each thread has its own thread identifier, *pid*, and the model of a thread is the automaton (actually an Uppaal template parametrised by thread identifiers) shown in Figure 4.

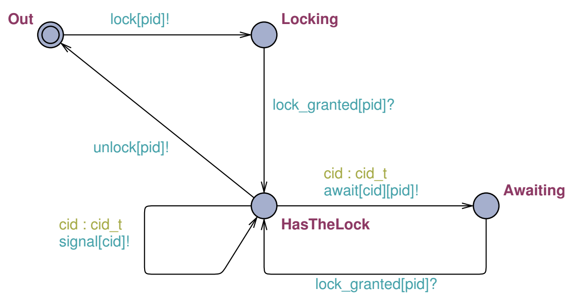

Fig. 4. State model for a Java thread invoking operations on locks.

The automaton has four locations that represent the state of a thread with respect to a lock:

– Out is the location that represents that the thread neither has got the lock nor is waiting in a condition. It is an abstract representation of any program point of any arbitrary thread *outside* of the monitor.
– HasTheLock is the location that represents that the thread has the lock. It is another abstract representation, this time, of any program point of any arbitrary thread *in* the monitor.
– Locking is the location that represents that the thread is blocked while waiting for the lock to be granted to it. From the monitor perspective, it is trying to enter the monitor.
– Awaiting is the location that represents that the thread is blocked in a condition. From the monitor perspective, it is waiting for a signal that allows it to re-enter the monitor.

Edges are extremely simple. The idea behind this simplicity is trying to *contaminate* the thread models as little as possible. They do not involve access to variables, neither global nor local so adapting the model to any particular thread is immediate. Let us see the intended meaning of the actions at the edges:

– Out–Locking with action lock[pid]!, a send-action that indicates that the thread needs the lock. As we will see, the co-action (lock[pid]?) is continuously enabled in the model of the mechanism.
– Locking–HasTheLock with action lock_granted[pid]?, a receive-action that indicates the thread is allowed to get the lock. The co-action will occur when the thread *pid* has the *top* priority to lock the monitor.
– HasTheLock–Awaiting with action await[cid][pid]!, a send-action that indicates that the thread waits in the condition variable *cid*.[1]
– Awaiting–HasTheLock with action lock_granted[pid]?, a receive-action that indicates the thread is allowed to re-take the lock (re-enter the monitor).

[1] In Uppaal, expressions such as cid : cid_t non-deterministically bind the identifier *cid* to a value in the range cid_t. It can be understood as an edge *replicator*.

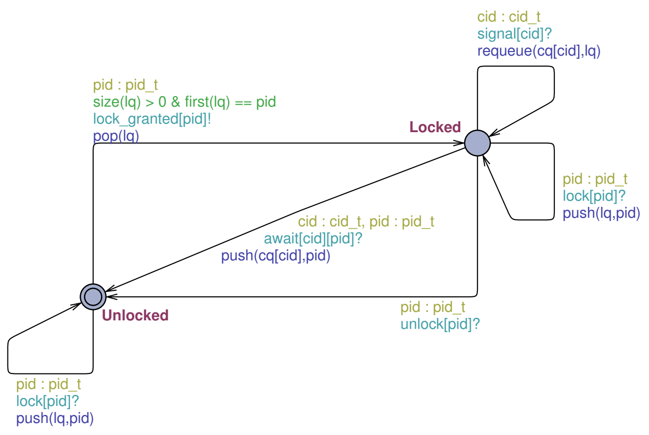

Fig. 5. State model for a Java lock.

- HasTheLock–Out with action unlock[*pid*]!, a send-action that indicates that the thread releases the lock (leave the monitor).
- HasTheLock–HasTheLock with action signal[*cid*]!, a send-action that indicates that the thread signals a thread on the condition variable *cid*.

3.2 Modelling Java Locks and Conditions

The model of a lock object and its associated condition variables is shown in Figure 5. The main task of the automaton is to keep track of threads by *listening* for their actions (on lock, await and signal channels) and responding appropriately (on channel lock_granted).

To keep track of the threads, the automaton defines two queues:

- A bounded queue of thread identifiers, lq (*lock queue*), that represent the queue for getting the lock.
- A bounded queue per condition *cid* of thread identifiers, cq[*cid*] (*condition queue of cid*), that represent the condition variable queues.

The automaton has two locations, Locked and Unlocked, that represent that the lock is locked by a thread or unlocked, respectively.

Most logic is encoded in the edges. To explain this logic we will explore the actions that fire them. We have to take into account that edges are *indexed* by thread identifiers (*pid*) and condition variable identifiers (*cid*).

- Edges with the receive-action lock[*pid*]? do not change locations, are always enabled and their assignments just push the thread identifier index *pid* in the access queue lq.
- The edge Locked–Unlocked with the receive-action await[*cid*][*pid*]? is always enabled and the assignment just pushes the thread identifier index *pid* in the condition variable queue cq[*cid*].

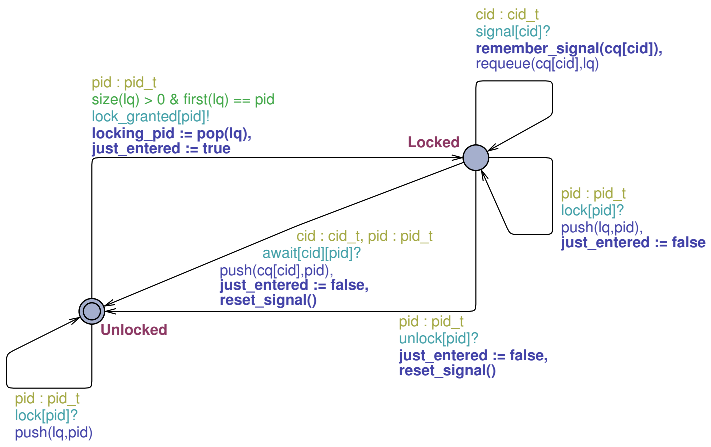

Fig. 6. An instrumented version of the Lock automaton.

- The edge Locked–Locked with the receive-action signal[cid][pid]? is always enabled and the assignment just moves the first thread identifier index pid from the condition queue cq[cid] to the access queue lq.
- The edge Unlocked–Locked with index pid is just enabled when the access queue lq is not empty and its first thread identifier is pid. The send-action of this edge is lock_granted[pid]!, granting the access to the thread with identifier pid.
- The edge Locked–Unlocked with the receive-action unlock[pid]? is always enabled and does not modify any queue.

The full code for the declarations and auxiliary method definitions in this model can be found under http://babel.ls.fi.upm.es/software/cclib.

The natural step after defining this model is to check some properties on it. The first one is the correctness property: mutual exclusion. Encoding mutual exclusion in the Uppaal subset of CTL, the Computational Tree Logic [5], is easy:

$$
\begin{aligned}
&\mathsf{A}\Box\ (\forall\ (\mathsf{pid1} : \mathsf{pid\_t})\ (\\
&\qquad \forall\ (\mathsf{pid2} : \mathsf{pid\_t})\ (\\
&\qquad\quad (\mathsf{Thread(pid1).HasTheLock} \wedge \mathsf{Thread(pid2).HasTheLock}) \Rightarrow \mathsf{pid1} = \mathsf{pid2})))
\end{aligned}
$$

That is, whether for all paths (A path quantifier), invariantly ($\Box$ modality), if *two* threads got the lock it is because they are the same thread. The tool finds the property correct for models of various sizes.

3.3 Instrumenting the Model

Nevertheless, for us, the most relevant property (correctness aside) is whether signalled threads resume their execution inside the monitor immediately after the signalling thread leaves. We will call this the *characteristic* property. As a sanity check, we woud like to check that this property does *not* hold for the model just presented.

However, stating this property in the query language supported by Uppaal is not possible. Basically, the query language is a subset of CTL that only allows temporal modalities as the topmost symbol of formulae, but a query representing the characteristic property for our model would require to nest modal operators.

A workaround for this limitation consists in instrumenting the model so that part of its recent history is somehow *remembered*. This way, we can encode certain temporal properties as state predicates, thus reducing the number of temporal operators required to express the characteristic property.

Again, we avoid contaminating the thread model and Figure 6 shows the resulting lock automaton (changes bold-faced). Basically, the system has new global variables to store:

- whether the nth thread that got access to the lock did execute an effective signal on some condition, and
- the identifier for the thread that received the signal (if any).

The first is done thanks to a new variable just_signalled, and the second with variable thread_just_awakened. Both of them are set by operation remember_signal while operation reset_signal resets just_signalled once the thread leaving the monitor coincides with thread_just_awakened. Also, the automata for threads are slightly constrained so that at most one signal is allowed per lock.

With these changes, the characteristic property is violated iff the $(n + 1)$th thread in gaining access to the lock is different from the thread signalled by the nth thread. In order to avoid having generation counters in the model, we have added yet another boolean variable just_entered to represent that the last transition to take place in the lock automaton is precisely the one that follows a lock_granted message. Violation of the characteristic property can then be encoded in the Uppaal query

$$\mathsf{E}\Diamond\ (\mathsf{just\_entered} \wedge \mathsf{just\_signalled} \wedge \mathsf{locking\_pid} \neq \mathsf{thread\_just\_awakened}).$$

That is, whether there is some path (E path quantifier) that eventually leads ($\Diamond$ modality) to a state in which the aforementioned proposition holds. The tool finds an example for 3 threads almost immediately, as expected.

4 Verifying our Implementation

Since our implementation of priority monitors is based on the existing lock and conditions model:

- The lock `mutex` is represented by the lock model in Figure 5.
- The condition `purgatory` is represented by the condition identifier purgacid (a symbolic name for referring to condition identifier 0).
- Integers `inPurgatory` and `pendingSignals` are represented by **int** Uppaal variables with the same name (inPurgatory and pendingSignals).

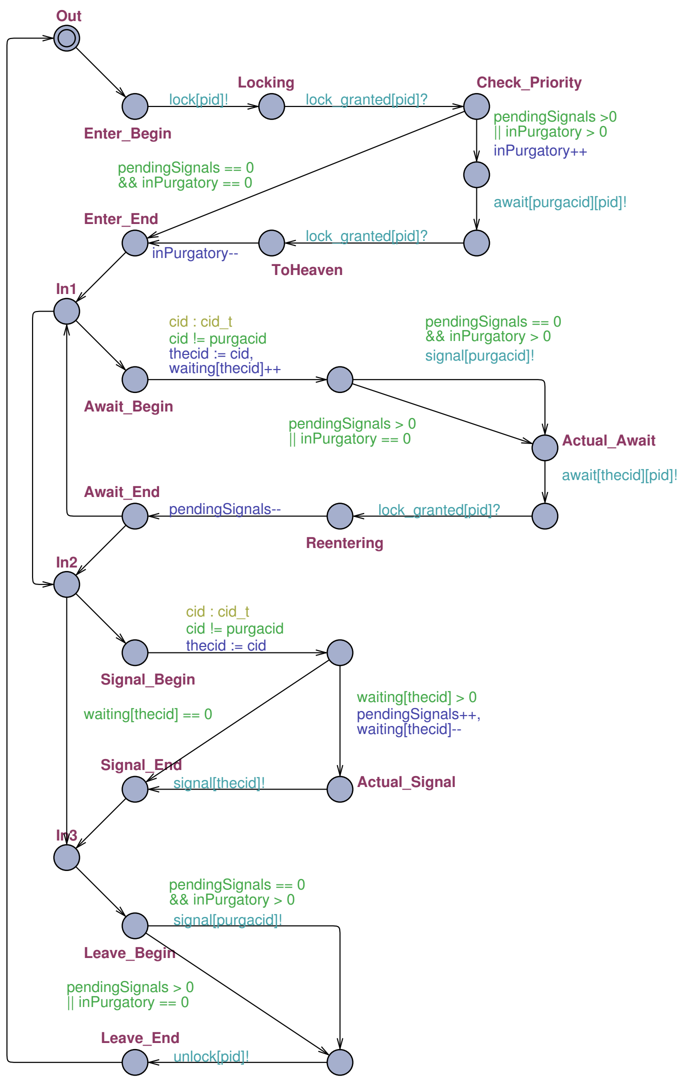

Fig. 7. State model for a thread using the Monitor class operations.

– The implementation of the methods `enter`, `leave`, `await`, and `signal` have been *inlined* in the thread model (Figure 7).

The automaton in Figure 7 models a very general thread that enters the monitor, then executes several `await` calls (possibly 0), then executes one, at the most, `signal` call, and finally leaves the monitor. To make the thread model easier to follow, most of the locations have been named according to fragments of the source code in Monitor.java. For example, execution of method `enter()` starts at location Enter_Begin and finishes at location Enter_End and includes the

	2	3	4	5	6
1:	0.01	0.02	0.18	6.38	274.03
2:	0.01	0.03	1.14	87.62	
3:	0.01	0.06	3.90		
4:	0.01	0.10	9.78		

locks & conditions

	2	3	4	5	6
1:	0.00	0.03	0.50	17.06	442.26
2:	0.01	0.08	3.35	214.99	
3:	0.02	0.18	12.06		
4:	0.02	0.39	35.70		

priority monitors

Table 1. Times spent in checking the mutual exclusion property.

	3	4	5	6
1:	0.00	0.02	0.08	0.40
2:	0.01	0.04	0.16	0.82
3:	0.01	0.08	0.32	1.48
4:	0.03	0.11	0.55	2.40

locks & conditions
(property violated)

	3	4	5	6
1:	0.03	0.36	11.18	450.73
2:	0.07	2.43	152.99	
3:	0.16	9.42		
4:	0.30	24.95		

priority monitors

Table 2. Times spent in checking the characteristic property.

possibility of a short excursion to the purgatory. The same scheme has been used to represent all the methods.

4.1 Experimental Results

The experimental results shown in this section have run on the instrumented models of the Java lock and conditions and of our priority monitor implementation. The model of the threads are those presented in previous sections except for the elimination of the main loop[2] (otherwise the state explosion makes the tool useless).

We have checked the correctness property for both models (lock and conditions and monitor) with the expected result: the models satisfy mutual exclusion. Table 1 shows execution times (in seconds) given different numbers of client threads (indicated at the top) and condition queues (indicated on the left).[3]

The state model for priority monitors has been instrumented following the ideas in Sec. 3.3. Table 2 shows execution times for checking the characteristic property for both instrumented models.

In addition to the characteristic property, we have also checked the property that threads sent to the purgatory eventually enter the monitor, as an implementation that would keep them blocked forever would satisfy the characteristic property trivially. The results are shown in Table 3.

[2] Represented by edge Leave_End-Out in Figure 7.

[3] Figures obtained on a Dual-Core AMD Opteron(tm) Processor 2218 @ 2.6GHz, RAM 8GB, running the Academic version of Uppaal 4.1.7 (rev. 4934) running a Debian distro with kernel Linux 2.6.39-bpo.2-amd64 SMP.

	2	3	4	5
1:	0.01	0.02	0.42	14.04
2:	0.01	0.07	2.53	169.37
3:	0.01	0.15	9.49	
4:	0.01	0.30	27.52	
5:	0.02	0.58	64.18	

Table 3. Times spent in checking that the stay in the purgatory is transitory.

5 Towards a Mathematical Proof

In order to prove our implementation of monitors correct for an unbounded number of threads we have encoded it in TLA, the temporal logic of actions [8], a logic for specifying and reasoning about concurrent systems. This has been done in two steps:

– A straightforward, manual, transformation of the Java code to *PlusCal* [10], an algorithm description language specially suited for concurrency.
– An automatic transformation of the PlusCal descriptions into TLA+ [9], a complete specification language based on TLA, using The TLA Toolbox [12], which allow both model-checking finite systems[4] but also symbolic proof.

The PlusCal code is shown in figures 8–10. Figure 8 contains the specification of the original locks and conditions library as a set of four procedures which can be invoked by concurrent processes. Analogously to the Uppaal model, locks and conditions are modelled as FIFO queues ("lq" and "cq[*cid*]"). The **await** statements (used at labels *java_lock_blocked* and *java_await_blocked*) ensure the synchronization behaviour. Labels in PlusCal specifications are not only an aid to the reader, but also help structuring the resulting TLA spec, and will be used for reasoning about the system behaviour.

The direct translation of our monitors is shown in Figure 9. The only relevant change is that part of the code (see labels *cclib_leave_resetlastsignal* and *cclib_signal_setlastsignal*) manages an extra variable (*lastSignal*) used to ease stating and proving the characteristic property, similarly to the instrumentation in the Uppaal model. Variable *lastSignal* is a tuple with two process identifiers. Initially, and when no signal is in progress, *lastSignal* $= \langle 0, 0\rangle$ (0 is a non-valid process identifier). Otherwise, it records the pids of the signaler and signaled processes.

Finally, a system of several processes to reason about the behaviour of our algorithm is defined in Figure 10. These clients enter the monitor (*proc_enter*), execute 0 or 1 calls to await (*proc_await*), execute 0 or 1 signal (*proc_signal*), and then leave the monitor (*proc_leave*).

Theorem 1 (Characteristic property). *When a process performs a signal on a nonempty condition (label cclib_signal_setlastsignal in Figure 9), the next*

[4] All theorems in this section have been model-checked with TLC, the TLA+ model-checker, prior to the symbolic proof.

```
                          procedure java_lock() {
java_lock_begin:            lq := Append(lq,self);
java_lock_blocked:          await self = Head(lq);
java_lock_end:              return;
                          }
                          procedure java_unlock() {
java_unlock_begin:          lq := Tail(lq);
java_unlock_end:            return;
                          }
                          procedure java_await(wcid) {
java_await_begin:           cq[wcid] := Append(cq[wcid],self);
java_await_unlocking:       lq := Tail(lq);
java_await_blocked:         await Len(lq) > 0 ∧ self = Head(lq);
java_await_end:             return;
                          }
                          procedure java_signal(scid) {
java_signal_begin:          if (Len(cq[scid]) > 0) {
java_signal_requeue:          lq := Append(lq,Head(cq[scid]));
                              cq[scid] := Tail(cq[scid]);
                            };
java_signal_end:            return;
                          }
```

Fig. 8. PlusCal specification of Java locks and conditions.

process to enter the monitor (label proc_in in Figure 10) is the signaled one, and not an "opportunist" process:

$$\forall\ pid \in ProcSet : pc[pid] = \text{"proc\_in"} \Rightarrow lastSignal[2] \in \{0,\ pid\}$$

Theorem 2 (Purgatory is transitory). *Every process sent to the purgatory (cq[0]) eventually enters the monitor (label proc_in in Figure 10):*

$$\forall\ pid \in ProcSet : (InPurgatory(pid) \leadsto pc[pid] = \text{"proc\_in"})$$

(The temporal formula $\phi \leadsto \psi$ means that every state satisfying ϕ will eventually progress to another in which ψ holds.)

Theorem 3 (Enter call order is preserved). *No process (pid2) entering the monitor (label java_lock_blocked in Figure 8) can overtake*[5] *any process (pid1) sent to the purgatory:*

$$\begin{aligned}\forall\ pid1, pid2 \in ProcSet{:}&(Member(cq[0],\ pid1) \land pc[pid2] = \text{"java\_lock\_blocked"})\\ &\leadsto (\ \land Member(order,\ pid1) \land Member(order,\ pid2)\\ &\qquad \land Index(order,\ pid1) < Index(order,\ pid2))\end{aligned}$$

The proof of the first theorem is lengthy due to the number of cases, but relatively straightforward, as is an invariant maintained by all transitions, and

[5] Now the use of the instrumental queue *order* becomes apparent.

```
                                    procedure cclib_enter() {
cclib_enter_begin:                    call java_lock();
cclib_enter_check_priority:           if (pendingSignals $>$ 0 $\vee$ inPurgatory $>$ 0) {
                                        inPurgatory := inPurgatory + 1;
                                        call java_await(0);
cclib_enter_toheaven:                   inPurgatory := inPurgatory - 1;
                                      };
cclib_enter_end:                      return;
                                    }
                                    procedure cclib_leave() {
cclib_leave_begin:                    if (pendingSignals = 0 $\wedge$ inPurgatory $>$ 0) {
cclib_leave_saveit:                     call java_signal(0);
                                      };
cclib_leave_resetlastsignal:          if (lastSignal[1] $\neq$ self) { lastSignal := $\langle 0, 0 \rangle$; };
cclib_leave_unlocking:                call java_unlock();
cclib_leave_end:                      return;
                                    }
                                    procedure cclib_await(cclwcid) {
cclib_await_begin:                    waiting[cclwcid] := waiting[cclwcid] + 1;
                                      if (pendingSignals = 0 $\wedge$ inPurgatory $>$ 0) {
cclib_await_saveit                      call java_signal(0);
                                      };
cclib_await_actualawait:              call java_await(cclwcid);
cclib_await_reentering:               pendingSignals := pendingSignals - 1;
cclib_await_end:                      return;
                                    }
                                    procedure cclib_signal(cclscid) {
cclib_signal_begin:                   if (waiting[cclscid] $>$ 0) {
cclib_signal_actualsignal:              pendingSignals := pendingSignals + 1;
                                        waiting[cclscid] := waiting[cclscid] - 1;
cclib_signal_setlastsignal:             lastSignal := $\langle$self, Head(cq[cclscid])$\rangle$;
cclib_signal_actualsignal:              call java_signal(cclscid);
                                      };
cclib_signal_end:                     return;
                                    }
```

Fig. 9. Priority monitors encoded in PlusCal.

independent of the number of processes. The second property is proved by induction on the number of awaiting processes. The third property is not strictly necessary for the safety requirements of the Monitor class, but is nice anyway. The full TLA theory and an appendix detailing the formal proof (done by hand) of the first two theorems can be found with the rest of the *cclib* software, at http://babel.ls.fi.upm.es/software/cclib/doc.

```
              process (procid ∈ ProcId) {
proc_enter:     call cclib_enter();
proc_in:        order := Append(order,self); \* to keep the order to gain access
proc_await:     with (cid ∈ CId) { if (cid # 0) { call cclib_await(cid); }; };
proc_signal:    with (cid ∈ CId) { if (cid # 0) { call cclib_signal(cid); }; };
proc_leave:     call cclib_leave();
              }
```

Fig. 10. Process specification.

6 Conclusion

We have presented an implementation of nonblocking (signal-and-continue) priority monitors in Java, implemented on top of the existing nonblocking, no-priority implementation in the standard locks & conditions package. Moreover, we have provided a state model for our solution (extending a model of the existing Java mechanism) that gives *formal evidence* that the priority mechanism is actually implemented by our algorithm.

Our `Monitor` class encourages the use of certain coding styles that cannot be used with the standard Java implementation (i.e., the 0-1 await/signal coding idiom) and that result in cleaner, safer code, and we are actually using it in the classroom as a replacement of standard locks and conditions.

To our knowledge, there are just two other publicly available implementations of priority monitors in Java. The first one, by Chiao, Wu and Yuan [4], is really a language extension devised to overcome the limitations of synchronized methods and *per-object* waitsets – the paper is from 1999 and locks and conditions had not been added to Java yet. The implementation is based on a preprocessor which translates extended Java programs into standard Java code that invokes the methods of some EMonitor class in the right sequence. In a more recent work, T.S. Norvell [11] already considers the limitations of Java monitors in the light of the provided locks and conditions library. However, his approach is different from ours, reimplementing the monitor functionality on top of lower level concurrency mechanisms (semaphores) and using explicit queues of threads. The fact that our code is conciser and based on higher-level methods has facilitated the use of model-checking as a validation tool, while his implementation is not verified.[6]

Some details of the reference implementation of Java locks have been omitted in our model, namely the possibility of spurious wakeups and reentrant locking. Regarding the first, it is fairly reasonable to assume the implementation of class ReentrantLock to be free of spurious wakeups as, in the most liberal interpretation of the API specification, they would make most attempts at formal reasoning useless. Regarding the second, it can be seen as a benefit of the model-driven approach: we do not consider features useless for the intended idioms.

[6] To be fair, Norvell's implementation is longer also because he extends the functionality of Java monitors in other ways.

Methodologically speaking, the use of PlusCal/TLA as an intermediate stage between model checking/Uppaal and a deductive proof using some program verification tool seems an adequate compromise. On one hand, the PlusCal model is closer to the actual Java code while containing all the information in the Uppaal one. Also, now that we have crafted a TLA proof of the key properties of our system by hand, we have a clearer idea of the techniques that should be supported by a program verification system in order to facilitate the task that remains future work.

Acknowledgments. We are very grateful to Bart Jacobs and the anonymous reviewers for their comments on an earlier version of this paper.

References

1. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004.
2. Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27:63–107, 1995.
3. Manuel Carro, Julio Mariño, Ángel Herranz, and Juan José Moreno-Navarro. Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In C.N. Dean and R.T. Boute, editors, *Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium*, volume 3294 of *LNCS*, pages 85–106. Springer, 2004. ISBN 3-540-23611-2.
4. Hsin-Ta Chiao, Chi-Houng Wu, and Shyan-Ming Yuan. A more expressive monitor for concurrent Java programming. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1053–1060. Springer Berlin / Heidelberg, 2000.
5. E. Allen Emerson and Joseph Y. Halpern. ̈sometimesänd ̈not never ̈revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.
6. Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34:38–45, 1999.
7. Ángel Herranz, Julio Mariño, Manuel Carro, and Juan José Moreno-Navarro. Modeling concurrent systems with shared resources. In *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, pages 102–116, 2009.
8. Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.
9. Leslie Lamport. *Specifying Systems.* Addison Wesley, 2004.
10. Leslie Lamport. The PlusCal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing (ICTAC2009)*, number 5684 in LNCS, pages 36–60. Springer Verlag, 2009.
11. Theodore S. Norvell. Better monitors for Java. *Javaworld*, October 2007. http://www.javaworld.com/javaworld/jw-10-2007/jw-10-monitors.html.
12. TLA+. The Way to Specify. http://www.tlaplus.net/.

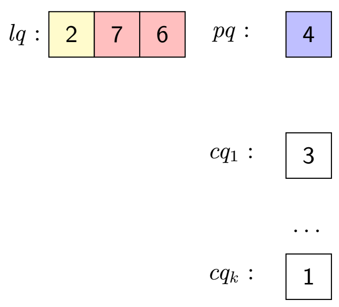

Fig. 11. State of the process queues in the PlusCal simulation.

A Proofs

A proof of the three properties stated in Section 5 follows. Section A.5 contains the whole specification of the algorithm in TLA+.

A.1 Facts of the Model

The model of our monitor implementation is based on the following facts that will be used in the proofs:

– For a process to gain access to the monitor, from *cclib_enter* or *cclib_await*(*cid*), its process id must be the first in the "locking queue" *lq* (and just one process can be at the head).
– Processes ids are inserted in this queue in three situations:
  1. When the process execute *java_lock*: label *java_lock_begin* in Figure 8.
  2. When the process is signalled by other process: the signaller moves the signalled process id from a condition queue (*cq*(*cid*)) to *lq*: label *java_signal_requeue* in Figure 8. This requeue can be done from two different points:
     (a) When the signalled process was in the purgatory: labels *cclib_leave_saveit* and *java_await_saveit* in Figure 9.
     (b) When the signalled process was in a user condition: label *cclib_actual_signal* in Figure 9.
– The only process that can progress is the one at the head of *lq*, as this is precisely the guard that controls the execution of processes after performing a *java_lock* (label *java_lock_blocked* in Figure 8) or a *java_await* (label *java_await_blocked* in Figure 8).

Several lemmata are assumed for all of them, which are stated first. Also, a graphical representation of a state of the PlusCal execution is shown in Figure 11.

Queue *lq* is the "lock" or "enter" queue. Process identifiers are *always* enqueued in *lq* when trying to execute an *enter*. Moreover, a pid can be inserted in this queue in two more situations: right after receiving a *signal* and after being sent to the *purgatory*. It is important to note that apart from the processes that try to execute *enter* – which simply enqueue their pids in *lq* and get blocked – the only process that can progress is the one at the head of *lq*, as this is precisely the guard that controls the execution of processes after performing an *enter*, thus ensuring mutual exclusion inside the monitor.

Queue $pq = cq_0$ is the purgatory queue. A process trying to enter "moves" its own pid from the head of *lq* to the rear of *pq* when there are pending signals – to give way to the signalled process – or when there are already some pid's annotated in the purgatory – so as to prevent overtaking other processes in a similar situation.

Queues $cq_1 \dots cq_k$ are the "condition" queues. A process that executes *await* on condition i moves its pid from the head of *lq* to the rear of cq_i. If, later, another process executes a signal on condition i, it will move the pid at the front of cq_i to the rear of *lq*, and *pendingSignals* is incremented. This is the second "state" that a process annotated in *lq* can be in.

As $pq = cq_0$, i.e., it is one distinguished condition queue, processes blocked on it are eventually signalled when no better option exists. That means that those pids will also be moved from the front of *pq* to the rear of *lq*. As this happens when the process is blocked at a different code point than a normal *signal*, we regard this as a third state. The colors in Figure 11 are used to distinguish the point of the code at which the process whose pid is at a given cell of *lq* blocked: *red* for processes visiting *lq* for the first time, *blue* for processes which come from the purgatory and *yellow* for processes signalled after being blocked on a (normal) condition.[7]

The aforementioned lemmata follow. Lemmas only needed for proving one of the main properties can be found in the corresponding sections.

Lemma 1 (Weak fairness). *We assume that all processes have finite progress.*

Proof. By definition, the system specification *Spec* in TLA+ directly specifies finite progress for every process: weak fairness is require for the *Next* action and the *Next* action means "every process is eligible for execution" (by a explicit enumeration of their program points):

$$\begin{aligned}
&\mathit{Spec} \triangleq \land\ \mathit{Init} \land \Box[\mathit{Next}]_{vars} \land \mathrm{WF}_{\mathbf{vars}}(\mathbf{Next})\\
&\mathbf{Next} \triangleq (\exists\ \mathit{self} \in \mathit{ProcSet} :\ \lor \mathit{java\_lock}(\mathit{self}) \lor \mathit{java\_unlock}(\mathit{self})\\
&\qquad\qquad\qquad\qquad\qquad\quad \lor \mathit{java\_await}(\mathit{self}) \lor \mathit{java\_signal}(\mathit{self})\\
&\qquad\qquad\qquad\qquad\qquad\quad \lor \mathit{cclib\_enter}(\mathit{self}) \lor \mathit{cclib\_leave}(\mathit{self})\\
&\qquad\qquad\qquad\qquad\qquad\quad \lor \mathit{cclib\_await}(\mathit{self}) \lor \mathit{cclib\_signal}(\mathit{self}))\\
&\qquad\qquad\qquad\qquad\qquad\quad \lor \mathit{proc\_enter}(\mathit{self}) \lor \mathit{proc\_in}(\mathit{self})\\
&\qquad\qquad\qquad\qquad\qquad\quad \lor \mathit{proc\_await}(\mathit{self}) \lor \mathit{proc\_signal}(\mathit{self})\\
&\qquad\qquad\qquad\qquad\qquad\quad \lor \mathit{proc\_leave}(\mathit{self}))
\end{aligned}$$

Lemma 2 (Mutual exclusion). *The only process that can progress* inside *the monitor is the one whose pid is annotated at the front of lq.*

[7] Abusing a bit, we have displayed the cell for process 4 in blue, state that will be reached when it is moved back to *lq* and no yellow cells exist.

Proof. Only one process id can be the first element of lq.

Lemma 3. *All queues have bounded size.*

Lemma 4 (At most one pending signal). *If processes are not allowed to call* signal *twice before they execute* leave*, the value of* pendingSignals *is at most one.*

A.2 Proof of Theorem 1

The characteristic property was expressed as an invariant of a slightly instrumented model in which a pair of two auxiliary variables were used to recall the pids of the signaler and signaled processes after performing a signal:

$$\forall\ pid \in ProcSet : (InPurgatory(pid) \leadsto pc[pid] = \text{“}\mathsf{proc\_in}\text{”})$$

Lemma 5. *There can only be one yellow pid in lq at a given time.*

We will consider several cases, depending on the execution state of the process whose pid is at the front of lq, which will be named p:

Case 1: a signaled process. Then p is a *yellow* cell. Initially, when p reaches the front of lq, we know that $pc[p]$ is not in any of the points required for the antecedent of the invariant. Then, the state evolves to a state in which the antecedent is true and (due to the previous lemma) the consequent is also true – $lastSignal[1] = p$ – and later, p performs a *leave*, which resets the pair (that makes the invariant hold) and finally removes p from the front of lq (which takes us to a different case).

Case 2: a locker*.* There are also several subcases here. If p is the signaler, an argument similar to the previous one shows that the system evolves only through states in which the invariant holds until p performs a *leave* – in this case $lastSignal[0] = p$ right before the *leave*.

If p is a (*red*) locker, and there is no pending signal, then $lastSignal = \langle 0, 0\rangle$, and the invariant holds when $pc[p] = \mathsf{locker\_in}$. Otherwise, if there is one signal pending, then p blocks in the purgatory before removing itself from lq so the invariant holds in all the states in between.

XM Note: Necesitamos un lema para esto?

The reasoning for *blue* lockers is analogous.

A.3 Proof of Theorem 2

The heart of this proof lies in showing the “liveness” of pq, i.e. that it decreases eventually. This is so because pids cannot be in the purgatory an arbitrary number of times.

Lemma 6. *A pid cannot be in lq more than three times.*

Lemma 7 (Liveness of pq). *The pid at the front of pq will be eventually removed.*

XM Note: esta es la parte que tiene seis casos.

A.4 Proof of Theorem 3

A.5 CCLib Monitor Specification in TLA+

```
──────────────────── MODULE Monitor ────────────────────
EXTENDS Naturals, Sequences, TLC, FiniteSets, TLAPS

────────────────────────────────────────────────────────
  Auxiliary definitions

  General definitions
PositiveInteger ≜ Nat \ {0}

  Operations on
Last(q) ≜ q[Len(q)]
Front(q) ≜ [i ∈ 1 .. (Len(q) − 1) ↦ q[i]]
  not used: Insert_as_2nd(q, x) ≜ ⟨x⟩ ∘ q
Member(q, e) ≜ ∃ i ∈ 1 .. Len(q) : q[i] = e
Index(q, e) ≜ CHOOSE i ∈ 1 .. Len(q) : q[i] = e

────────────────────────────────────────────────────────
  Process identifiers
CONSTANTS N_Procs
ASSUME AtLeastOne ≜  ∧ N_Procs ∈ PositiveInteger

First_ProcId ≜ 1
Last_ProcId ≜ First_ProcId + N_Procs − 1
ProcId ≜ First_ProcId .. Last_ProcId

────────────────────────────────────────────────────────
  Condition identifiers (condition number 0 will be implemented as purgatory)
CONSTANT N_Conditions
ASSUME N_Conditions ∈ PositiveInteger
CId ≜ 0 .. N_Conditions

────────────────────────────────────────────────────────
–algorithm TestMonitor
{

 variables
   \ * Problem variables
   cq = [cid ∈ CId ↦ ⟨⟩];
   lq = ⟨⟩;
   pendingSignals = 0;
   inPurgatory = 0;
   waiting = [cid ∈ CId ↦ 0];

   \ * Instrumentation
   lastSignal = ⟨0, 0⟩;  \ * ⟨signaller, signalled⟩
   entryOrder = ⟨⟩;
```

```
\ ******************************************************************
\ * Specification of the Java implementations of Lock and Conditions
procedure java_lock()
{
 java_lock_begin :
   $lq := Append(lq, self)$;
 java_lock_blocked :
   await $self = Head(lq)$;
 java_lock_end :
   return;
}
procedure java_unlock()
{
 java_unlock_begin :
   $lq := Tail(lq)$;
 java_unlock_end :
   return;
}
procedure java_await(wcid)
{
 java_await_begin :
   $cq[wcid] := Append(cq[wcid], self)$;
 java_await_unlocking :
   $lq := Tail(lq)$;
 java_await_blocked :
   await $Len(lq) > 0 \land self = Head(lq)$;
 java_await_end :
   return;
}
procedure java_signal(scid)
{
 java_signal_begin :
   if$(Len(cq[scid]) > 0)${
     $lq := Append(lq, Head(cq[scid]))$;
     $cq[scid] := Tail(cq[scid])$;
   };
 java_signal_end :
   return;
}
\ ******************************************************************
\ * Specification of the CCLib implementations of Monitors
procedure cclib_enter()
{
 cclib_enter_begin :
   call java_lock();
 cclib_enter_check_priority :
   if$(pendingSignals > 0 \lor inPurgatory > 0)${
     $inPurgatory := inPurgatory + 1$;
```

```
  $cclib\_enter\_to\_purgatory$ :
      $call\ java\_await(0);$
  $cclib\_enter\_toheaven$ :
      $inPurgatory := inPurgatory - 1;$
    $\};$
  $cclib\_enter\_end$ :
    $return;$
$\}$
$procedure\ \ cclib\_leave()$
$\{$
  $cclib\_leave\_begin$ :
    $if(pendingSignals = 0 \wedge inPurgatory > 0)\{$
      $call\ java\_signal(0);$
    $\};$
  $cclib\_leave\_resetlastsignal$ :
    $if(lastSignal[1] \neq self)\{$
      $lastSignal := \langle 0,\ 0\rangle;$
    $\};$
  $cclib\_leave\_unlocking$ :
    $call\ java\_unlock();$
  $cclib\_leave\_end$ :
    $return;$
$\}$
$procedure\ \ cclib\_await(cclwcid)$
$\{$
  $cclib\_await\_begin$ :
    $waiting[cclwcid] := waiting[cclwcid] + 1;$
    $if(pendingSignals = 0 \wedge inPurgatory > 0)\{$
      $call\ java\_signal(0);$
    $\};$
  $cclib\_await\_actualawait$ :
    $call\ java\_await(cclwcid);$
  $cclib\_await\_reentering$ :
    $pendingSignals := pendingSignals - 1;$
  $cclib\_await\_end$ :
    $return;$
$\}$
$procedure\ \ cclib\_signal(cclscid)$
$\{$
  $cclib\_signal\_begin$ :
    $if(waiting[cclscid] > 0)\{$
  $cclib\_signal\_actualsignal$ :
      $pendingSignals := pendingSignals + 1;$
      $waiting[cclscid] := waiting[cclscid] - 1;$
  $cclib\_signal\_setlastsignal$ :
      $lastSignal := \langle self,\ Head(cq[cclscid])\rangle;$
      $call\ java\_signal(cclscid);$
    $\};$
  $cclib\_signal\_end$ :
    $return$;
 }
 \ ******************************************************************
 \ * Process in the system
 $process(procid \in ProcId)$
 {
  $proc\_enter$ :
    $call\ cclib\_enter()$;

  $proc\_in$ :
    $entryOrder := Append(entryOrder, self)$;

  $proc\_await$ :
    $with(cid \in CId)\{$
      $if(cid \neq 0)\{$
        $call\ cclib\_await(cid)$;
      };
    };



  $proc\_signal$ :
    $with(cid \in CId)\{$
      $if(cid \neq 0)\{$
        $call\ cclib\_signal(cid)$;
      };
    };



  $proc\_leave$ :
    $call\ cclib\_leave()$;
 }
}
 BEGIN TRANSLATION
CONSTANT $defaultInitValue$
VARIABLES $cq, lq, pendingSignals, inPurgatory, waiting, lastSignal,$
          $entryOrder, pc, stack, wcid, scid, cclwcid, cclscid$

$vars \triangleq \langle cq, lq, pendingSignals, inPurgatory, waiting, lastSignal,$
         $entryOrder, pc, stack, wcid, scid, cclwcid, cclscid\rangle$

$ProcSet \triangleq (ProcId)$

$Init \triangleq$  Global variables
         $\land\ cq = [cid \in CId \mapsto \langle\rangle]$
         $\land\ lq = \langle\rangle$
         $\land\ pendingSignals = 0$
         $\land\ inPurgatory = 0$
```

$$\begin{aligned}
&\land\ \mathit{waiting} = [\mathit{cid} \in \mathit{CId} \mapsto 0]\\
&\land\ \mathit{lastSignal}\ \ = \langle 0,\ 0\rangle\\
&\land\ \mathit{entryOrder} = \langle\rangle\\
&\ \text{Procedure } \mathit{java\_await}\\
&\land\ \mathit{wcid} = [\mathit{self} \in \mathit{ProcSet} \mapsto \mathit{defaultInitValue}]\\
&\ \text{Procedure } \mathit{java\_signal}\\
&\land\ \mathit{scid} = [\mathit{self} \in \mathit{ProcSet} \mapsto \mathit{defaultInitValue}]\\
&\ \text{Procedure } \mathit{cclib\_await}\\
&\land\ \mathit{cclwcid} = [\mathit{self} \in \mathit{ProcSet} \mapsto \mathit{defaultInitValue}]\\
&\ \text{Procedure } \mathit{cclib\_signal}\\
&\land\ \mathit{cclscid} = [\mathit{self} \in \mathit{ProcSet} \mapsto \mathit{defaultInitValue}]\\
&\land\ \mathit{stack} = [\mathit{self} \in \mathit{ProcSet} \mapsto \langle\rangle]\\
&\land\ \mathit{pc} = [\mathit{self} \in \mathit{ProcSet} \mapsto \text{CASE } \mathit{self} \in \mathit{ProcId} \rightarrow \text{“proc\_enter”}]
\end{aligned}$$

$$\begin{aligned}
\mathit{java\_lock\_begin}(\mathit{self}) \triangleq\ &\land\ \mathit{pc}[\mathit{self}] = \text{“java\_lock\_begin”}\\
&\land\ \mathit{lq}' = \mathit{Append}(\mathit{lq},\ \mathit{self})\\
&\land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“java\_lock\_blocked”}]\\
&\land\ \text{UNCHANGED } \langle \mathit{cq},\ \mathit{pendingSignals},\ \mathit{inPurgatory},\\
&\qquad\qquad \mathit{waiting},\ \mathit{lastSignal},\ \mathit{entryOrder},\\
&\qquad\qquad \mathit{stack},\ \mathit{wcid},\ \mathit{scid},\ \mathit{cclwcid},\\
&\qquad\qquad \mathit{cclscid}\rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{java\_lock\_blocked}(\mathit{self}) \triangleq\ &\land\ \mathit{pc}[\mathit{self}] = \text{“java\_lock\_blocked”}\\
&\land\ \mathit{self} = \mathit{Head}(\mathit{lq})\\
&\land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“java\_lock\_end”}]\\
&\land\ \text{UNCHANGED } \langle \mathit{cq},\ \mathit{lq},\ \mathit{pendingSignals},\\
&\qquad\qquad \mathit{inPurgatory},\ \mathit{waiting},\\
&\qquad\qquad \mathit{lastSignal},\ \mathit{entryOrder},\ \mathit{stack},\\
&\qquad\qquad \mathit{wcid},\ \mathit{scid},\ \mathit{cclwcid},\ \mathit{cclscid}\rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{java\_lock\_end}(\mathit{self}) \triangleq\ &\land\ \mathit{pc}[\mathit{self}] = \text{“java\_lock\_end”}\\
&\land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \mathit{Head}(\mathit{stack}[\mathit{self}]).\mathit{pc}]\\
&\land\ \mathit{stack}' = [\mathit{stack} \text{ EXCEPT } ![\mathit{self}] = \mathit{Tail}(\mathit{stack}[\mathit{self}])]\\
&\land\ \text{UNCHANGED } \langle \mathit{cq},\ \mathit{lq},\ \mathit{pendingSignals},\\
&\qquad\qquad \mathit{inPurgatory},\ \mathit{waiting},\ \mathit{lastSignal},\\
&\qquad\qquad \mathit{entryOrder},\ \mathit{wcid},\ \mathit{scid},\ \mathit{cclwcid},\\
&\qquad\qquad \mathit{cclscid}\rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{java\_lock}(\mathit{self}) \triangleq\ &\mathit{java\_lock\_begin}(\mathit{self}) \lor \mathit{java\_lock\_blocked}(\mathit{self})\\
&\quad \lor \mathit{java\_lock\_end}(\mathit{self})
\end{aligned}$$

$$\begin{aligned}
\mathit{java\_unlock\_begin}(\mathit{self}) \triangleq\ &\land\ \mathit{pc}[\mathit{self}] = \text{“java\_unlock\_begin”}\\
&\land\ \mathit{lq}' = \mathit{Tail}(\mathit{lq})\\
&\land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“java\_unlock\_end”}]\\
&\land\ \text{UNCHANGED } \langle \mathit{cq},\ \mathit{pendingSignals},\\
&\qquad\qquad \mathit{inPurgatory},\ \mathit{waiting},\\
&\qquad\qquad \mathit{lastSignal},\ \mathit{entryOrder},\ \mathit{stack},
\end{aligned}$$

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad wcid, scid, cclwcid, cclscid\rangle \end{array}$$

$$\begin{array}{ll} java\_unlock\_end(self) \triangleq & \land\ pc[self] = \text{“java\_unlock\_end”} \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = Head(stack[self]).pc] \\ & \land\ stack' = [stack \text{ EXCEPT } ![self] = Tail(stack[self])] \\ & \land\ \text{UNCHANGED } \langle cq, lq, pendingSignals, \\ & \qquad\qquad\qquad inPurgatory, waiting, \\ & \qquad\qquad\qquad lastSignal, entryOrder, wcid, \\ & \qquad\qquad\qquad scid, cclwcid, cclscid\rangle \end{array}$$

$$\begin{array}{ll} java\_unlock(self) \triangleq & java\_unlock\_begin(self) \\ & \lor\ java\_unlock\_end(self) \end{array}$$

$$\begin{array}{ll} java\_await\_begin(self) \triangleq & \land\ pc[self] = \text{“java\_await\_begin”} \\ & \land\ cq' = [cq \text{ EXCEPT } ![wcid[self]] = Append(cq[wcid[self]], self)] \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“java\_await\_unlocking”}] \\ & \land\ \text{UNCHANGED } \langle lq, pendingSignals, \\ & \qquad\qquad\qquad inPurgatory, waiting, \\ & \qquad\qquad\qquad lastSignal, entryOrder, stack, \\ & \qquad\qquad\qquad wcid, scid, cclwcid, cclscid\rangle \end{array}$$

$$\begin{array}{ll} java\_await\_unlocking(self) \triangleq & \land\ pc[self] = \text{“java\_await\_unlocking”} \\ & \land\ lq' = Tail(lq) \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“java\_await\_blocked”}] \\ & \land\ \text{UNCHANGED } \langle cq, pendingSignals, \\ & \qquad\qquad\qquad inPurgatory, waiting, \\ & \qquad\qquad\qquad lastSignal, entryOrder, \\ & \qquad\qquad\qquad stack, wcid, scid, cclwcid, \\ & \qquad\qquad\qquad cclscid\rangle \end{array}$$

$$\begin{array}{ll} java\_await\_blocked(self) \triangleq & \land\ pc[self] = \text{“java\_await\_blocked”} \\ & \land\ Len(lq) > 0 \land self = Head(lq) \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“java\_await\_end”}] \\ & \land\ \text{UNCHANGED } \langle cq, lq, pendingSignals, \\ & \qquad\qquad\qquad inPurgatory, waiting, \\ & \qquad\qquad\qquad lastSignal, entryOrder, \\ & \qquad\qquad\qquad stack, wcid, scid, cclwcid, \\ & \qquad\qquad\qquad cclscid\rangle \end{array}$$

$$\begin{array}{ll} java\_await\_end(self) \triangleq & \land\ pc[self] = \text{“java\_await\_end”} \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = Head(stack[self]).pc] \\ & \land\ wcid' = [wcid \text{ EXCEPT } ![self] = Head(stack[self]).wcid] \\ & \land\ stack' = [stack \text{ EXCEPT } ![self] = Tail(stack[self])] \\ & \land\ \text{UNCHANGED } \langle cq, lq, pendingSignals, \\ & \qquad\qquad\qquad inPurgatory, waiting, lastSignal, \\ & \qquad\qquad\qquad entryOrder, scid, cclwcid, \\ & \qquad\qquad\qquad cclscid\rangle \end{array}$$

$$\begin{array}{l}
\mathit{java\_await}(\mathit{self}) \triangleq \mathit{java\_await\_begin}(\mathit{self}) \\
\qquad \lor \mathit{java\_await\_unlocking}(\mathit{self}) \\
\qquad \lor \mathit{java\_await\_blocked}(\mathit{self}) \lor \mathit{java\_await\_end}(\mathit{self})
\end{array}$$

$$\begin{array}{l}
\mathit{java\_signal\_begin}(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“java\_signal\_begin”} \\
\qquad \land \text{ IF } \mathit{Len}(\mathit{cq}[\mathit{scid}[\mathit{self}]]) > 0 \\
\qquad\qquad \text{THEN } \land\ \mathit{lq}' = \mathit{Append}(\mathit{lq}, \mathit{Head}(\mathit{cq}[\mathit{scid}[\mathit{self}]])) \\
\qquad\qquad\qquad \land\ \mathit{cq}' = [\mathit{cq} \text{ EXCEPT } ![\mathit{scid}[\mathit{self}]] = \mathit{Tail}(\mathit{cq}[\mathit{scid}[\mathit{self}]])] \\
\qquad\qquad \text{ELSE } \land \text{ TRUE} \\
\qquad\qquad\qquad \land \text{ UNCHANGED } \langle \mathit{cq}, \mathit{lq} \rangle \\
\qquad \land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“java\_signal\_end”}] \\
\qquad \land \text{ UNCHANGED } \langle \mathit{pendingSignals}, \mathit{inPurgatory}, \\
\qquad\qquad \mathit{waiting}, \mathit{lastSignal}, \\
\qquad\qquad \mathit{entryOrder}, \mathit{stack}, \mathit{wcid}, \mathit{scid}, \\
\qquad\qquad \mathit{cclwcid}, \mathit{cclscid} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{java\_signal\_end}(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“java\_signal\_end”} \\
\qquad \land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \mathit{Head}(\mathit{stack}[\mathit{self}]).\mathit{pc}] \\
\qquad \land\ \mathit{scid}' = [\mathit{scid} \text{ EXCEPT } ![\mathit{self}] = \mathit{Head}(\mathit{stack}[\mathit{self}]).\mathit{scid}] \\
\qquad \land\ \mathit{stack}' = [\mathit{stack} \text{ EXCEPT } ![\mathit{self}] = \mathit{Tail}(\mathit{stack}[\mathit{self}])] \\
\qquad \land \text{ UNCHANGED } \langle \mathit{cq}, \mathit{lq}, \mathit{pendingSignals}, \\
\qquad\qquad \mathit{inPurgatory}, \mathit{waiting}, \\
\qquad\qquad \mathit{lastSignal}, \mathit{entryOrder}, \mathit{wcid}, \\
\qquad\qquad \mathit{cclwcid}, \mathit{cclscid} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{java\_signal}(\mathit{self}) \triangleq \mathit{java\_signal\_begin}(\mathit{self}) \\
\qquad \lor \mathit{java\_signal\_end}(\mathit{self})
\end{array}$$

$$\begin{array}{l}
\mathit{cclib\_enter\_begin}(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“cclib\_enter\_begin”} \\
\qquad \land\ \mathit{stack}' = [\mathit{stack} \text{ EXCEPT } ![\mathit{self}] = \langle [\mathit{procedure} \mapsto \text{“java\_lock”}, \\
\qquad\qquad\qquad \mathit{pc} \mapsto \text{“cclib\_enter\_check\_priority”}] \\
\qquad\qquad\qquad \circ\ \mathit{stack}[\mathit{self}]] \\
\qquad \land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“java\_lock\_begin”}] \\
\qquad \land \text{ UNCHANGED } \langle \mathit{cq}, \mathit{lq}, \mathit{pendingSignals}, \\
\qquad\qquad \mathit{inPurgatory}, \mathit{waiting}, \\
\qquad\qquad \mathit{lastSignal}, \mathit{entryOrder}, \mathit{wcid}, \\
\qquad\qquad \mathit{scid}, \mathit{cclwcid}, \mathit{cclscid} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{cclib\_enter\_check\_priority}(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“cclib\_enter\_check\_priority”} \\
\qquad \land \text{ IF } \mathit{pendingSignals} > 0 \lor \mathit{inPurgatory} > 0 \\
\qquad\qquad \text{THEN } \land\ \mathit{inPurgatory}' = \mathit{inPurgatory} + 1 \\
\qquad\qquad\qquad \land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“cclib\_enter\_to\_purgatory”}] \\
\qquad\qquad \text{ELSE } \land\ \mathit{pc}' = [\mathit{pc} \text{ EXCEPT } ![\mathit{self}] = \text{“cclib\_enter\_end”}] \\
\qquad\qquad\qquad \land \text{ UNCHANGED } \mathit{inPurgatory} \\
\qquad \land \text{ UNCHANGED } \langle \mathit{cq}, \mathit{lq}, \\
\qquad\qquad \mathit{pendingSignals}, \\
\qquad\qquad \mathit{waiting}, \mathit{lastSignal},
\end{array}$$

```
                                                     entryOrder, stack,
                                                     wcid, scid, cclwcid,
                                                     cclscid⟩

cclib_enter_to_purgatory(self) ≜ ∧ pc[self] = "cclib_enter_to_purgatory"
                                 ∧ ∧ stack′ = [stack EXCEPT ![self] = ⟨[procedure ↦ "java_await",
                                                                        pc        ↦ "cclib_enter_tohea
                                                                        wcid      ↦ wcid[self]]⟩
                                                                       ∘ stack[self]]
                                   ∧ wcid′ = [wcid EXCEPT ![self] = 0]
                                 ∧ pc′ = [pc EXCEPT ![self] = "java_await_begin"]
                                 ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                              inPurgatory, waiting,
                                              lastSignal, entryOrder,
                                              scid, cclwcid, cclscid⟩

cclib_enter_toheaven(self) ≜ ∧ pc[self] = "cclib_enter_toheaven"
                             ∧ inPurgatory′ = inPurgatory − 1
                             ∧ pc′ = [pc EXCEPT ![self] = "cclib_enter_end"]
                             ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                          waiting, lastSignal,
                                          entryOrder, stack, wcid,
                                          scid, cclwcid, cclscid⟩

cclib_enter_end(self) ≜ ∧ pc[self] = "cclib_enter_end"
                        ∧ pc′ = [pc EXCEPT ![self] = Head(stack[self]).pc]
                        ∧ stack′ = [stack EXCEPT ![self] = Tail(stack[self])]
                        ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                     inPurgatory, waiting,
                                     lastSignal, entryOrder, wcid,
                                     scid, cclwcid, cclscid⟩

cclib_enter(self) ≜ cclib_enter_begin(self)
                      ∨ cclib_enter_check_priority(self)
                      ∨ cclib_enter_to_purgatory(self)
                      ∨ cclib_enter_toheaven(self)
                      ∨ cclib_enter_end(self)

cclib_leave_begin(self) ≜ ∧ pc[self] = "cclib_leave_begin"
                          ∧ IF pendingSignals = 0 ∧ inPurgatory > 0
                                THEN ∧ ∧ scid′ = [scid EXCEPT ![self] = 0]
                                       ∧ stack′ = [stack EXCEPT ![self] = ⟨[procedure ↦ "java_signal"
                                                                           pc        ↦ "cclib_leave_r
                                                                           scid      ↦ scid[self]]⟩
                                                                          ∘ stack[self]]
                                     ∧ pc′ = [pc EXCEPT ![self] = "java_signal_begin"]
                                ELSE ∧ pc′ = [pc EXCEPT ![self] = "cclib_leave_resetlastsignal"]
```

$$\begin{array}{l}
\qquad\qquad\qquad\land \text{UNCHANGED } \langle stack, scid \rangle \\
\qquad\land \text{UNCHANGED } \langle cq, lq, pendingSignals, \\
\qquad\qquad\qquad inPurgatory, waiting, \\
\qquad\qquad\qquad lastSignal, entryOrder, wcid, \\
\qquad\qquad\qquad cclwcid, cclscid \rangle
\end{array}$$

$$cclib\_leave\_resetlastsignal(self) \triangleq \begin{array}[t]{l}
\land pc[self] = \text{"cclib\_leave\_resetlastsignal"} \\
\land \text{IF } lastSignal[1] \neq self \\
\qquad \text{THEN } \land lastSignal' = \langle 0, 0 \rangle \\
\qquad \text{ELSE } \begin{array}[t]{l} \land \text{TRUE} \\ \land \text{UNCHANGED } lastSignal \end{array} \\
\land pc' = [pc \text{ EXCEPT } ![self] = \text{"cclib\_leave\_unlocking"}] \\
\land \text{UNCHANGED } \begin{array}[t]{l} \langle cq, lq, \\ pendingSignals, \\ inPurgatory, \\ waiting, entryOrder, \\ stack, wcid, scid, \\ cclwcid, cclscid \rangle \end{array}
\end{array}$$

$$cclib\_leave\_unlocking(self) \triangleq \begin{array}[t]{l}
\land pc[self] = \text{"cclib\_leave\_unlocking"} \\
\land stack' = [stack \text{ EXCEPT } ![self] = \begin{array}[t]{l} \langle [procedure \mapsto \text{"java\_unlock"}, \\ \ \ pc \qquad\quad \mapsto \text{"cclib\_leave\_end"}] \rangle \\ \circ\, stack[self]] \end{array} \\
\land pc' = [pc \text{ EXCEPT } ![self] = \text{"java\_unlock\_begin"}] \\
\land \text{UNCHANGED } \begin{array}[t]{l} \langle cq, lq, pendingSignals, \\ inPurgatory, waiting, \\ lastSignal, entryOrder, \\ wcid, scid, cclwcid, \\ cclscid \rangle \end{array}
\end{array}$$

$$cclib\_leave\_end(self) \triangleq \begin{array}[t]{l}
\land pc[self] = \text{"cclib\_leave\_end"} \\
\land pc' = [pc \text{ EXCEPT } ![self] = Head(stack[self]).pc] \\
\land stack' = [stack \text{ EXCEPT } ![self] = Tail(stack[self])] \\
\land \text{UNCHANGED } \begin{array}[t]{l} \langle cq, lq, pendingSignals, \\ inPurgatory, waiting, \\ lastSignal, entryOrder, wcid, \\ scid, cclwcid, cclscid \rangle \end{array}
\end{array}$$

$$cclib\_leave(self) \triangleq \begin{array}[t]{l}
cclib\_leave\_begin(self) \\
\quad \lor cclib\_leave\_resetlastsignal(self) \\
\quad \lor cclib\_leave\_unlocking(self) \\
\quad \lor cclib\_leave\_end(self)
\end{array}$$

$$cclib\_await\_begin(self) \triangleq \begin{array}[t]{l}
\land pc[self] = \text{"cclib\_await\_begin"} \\
\land waiting' = [waiting \text{ EXCEPT } ![cclwcid[self]] = waiting[cclwcid[self]] + 1] \\
\land \text{IF } pendingSignals = 0 \land inPurgatory > 0 \\
\qquad \text{THEN } \land \land scid' = [scid \text{ EXCEPT } ![self] = 0]
\end{array}$$

```
                                              ∧ stack' = [stack EXCEPT ![self] = ⟨[procedure ↦  "java_signal"
                                                                                   pc        ↦  "cclib_await_
                                                                                   scid      ↦  scid[self]]⟩
                                                                                   ∘ stack[self]]
                                            ∧ pc' = [pc EXCEPT ![self] = "java_signal_begin"]
                                      ELSE  ∧ pc' = [pc EXCEPT ![self] = "cclib_await_actualawait"]
                                            ∧ UNCHANGED ⟨stack, scid⟩
                              ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                           inPurgatory, lastSignal,
                                           entryOrder, wcid, cclwcid,
                                           cclscid⟩

cclib_await_actualawait(self) ≜  ∧ pc[self] = "cclib_await_actualawait"
                                 ∧  ∧ stack' = [stack EXCEPT ![self] = ⟨[procedure ↦  "java_await",
                                                                        pc        ↦  "cclib_await_reente
                                                                        wcid      ↦  wcid[self]]⟩
                                                                        ∘ stack[self]]
                                    ∧ wcid' = [wcid EXCEPT ![self] = cclwcid[self]]
                                 ∧ pc' = [pc EXCEPT ![self] = "java_await_begin"]
                                 ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                              inPurgatory, waiting,
                                              lastSignal, entryOrder,
                                              scid, cclwcid, cclscid⟩

cclib_await_reentering(self) ≜  ∧ pc[self] = "cclib_await_reentering"
                                ∧ pendingSignals' = pendingSignals − 1
                                ∧ pc' = [pc EXCEPT ![self] = "cclib_await_end"]
                                ∧ UNCHANGED ⟨cq, lq, inPurgatory,
                                             waiting, lastSignal,
                                             entryOrder, stack, wcid,
                                             scid, cclwcid, cclscid⟩

cclib_await_end(self) ≜  ∧ pc[self] = "cclib_await_end"
                         ∧ pc' = [pc EXCEPT ![self] = Head(stack[self]).pc]
                         ∧ cclwcid' = [cclwcid EXCEPT ![self] = Head(stack[self]).cclwcid]
                         ∧ stack' = [stack EXCEPT ![self] = Tail(stack[self])]
                         ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                      inPurgatory, waiting,
                                      lastSignal, entryOrder, wcid,
                                      scid, cclscid⟩

cclib_await(self) ≜  cclib_await_begin(self)
                     ∨ cclib_await_actualawait(self)
                     ∨ cclib_await_reentering(self)
                     ∨ cclib_await_end(self)

cclib_signal_begin(self) ≜  ∧ pc[self] = "cclib_signal_begin"
```

```
                                   ∧ IF waiting[cclscid[self]] > 0
                                         THEN ∧ pc′ = [pc EXCEPT ![self] = "cclib_signal_actualsignal"]
                                         ELSE ∧ pc′ = [pc EXCEPT ![self] = "cclib_signal_end"]
                                   ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                                 inPurgatory, waiting,
                                                 lastSignal, entryOrder,
                                                 stack, wcid, scid, cclwcid,
                                                 cclscid⟩

cclib_signal_actualsignal(self) ≜ ∧ pc[self] = "cclib_signal_actualsignal"
                                   ∧ pendingSignals′ = pendingSignals + 1
                                   ∧ waiting′ = [waiting EXCEPT ![cclscid[self]] = waiting[cclscid[self]] −
                                   ∧ pc′ = [pc EXCEPT ![self] = "cclib_signal_setlastsignal"]
                                   ∧ UNCHANGED ⟨cq, lq, inPurgatory,
                                                 lastSignal,
                                                 entryOrder, stack,
                                                 wcid, scid, cclwcid,
                                                 cclscid⟩

cclib_signal_setlastsignal(self) ≜ ∧ pc[self] = "cclib_signal_setlastsignal"
                                    ∧ lastSignal′ = ⟨self, Head(cq[cclscid[self]])⟩
                                    ∧ ∧ scid′ = [scid EXCEPT ![self] = cclscid[self]]
                                      ∧ stack′ = [stack EXCEPT ![self] = ⟨[procedure ↦ "java_signal",
                                                                           pc        ↦ "cclib_signal_end"
                                                                           scid      ↦ scid[self]]⟩
                                                                          ∘ stack[self]]
                                    ∧ pc′ = [pc EXCEPT ![self] = "java_signal_begin"]
                                    ∧ UNCHANGED ⟨cq, lq,
                                                  pendingSignals,
                                                  inPurgatory, waiting,
                                                  entryOrder, wcid,
                                                  cclwcid, cclscid⟩

cclib_signal_end(self) ≜ ∧ pc[self] = "cclib_signal_end"
                          ∧ pc′ = [pc EXCEPT ![self] = Head(stack[self]).pc]
                          ∧ cclscid′ = [cclscid EXCEPT ![self] = Head(stack[self]).cclscid]
                          ∧ stack′ = [stack EXCEPT ![self] = Tail(stack[self])]
                          ∧ UNCHANGED ⟨cq, lq, pendingSignals,
                                        inPurgatory, waiting,
                                        lastSignal, entryOrder, wcid,
                                        scid, cclwcid⟩

cclib_signal(self) ≜ cclib_signal_begin(self)
                      ∨ cclib_signal_actualsignal(self)
                      ∨ cclib_signal_setlastsignal(self)
                      ∨ cclib_signal_end(self)
```

```
$proc\_enter(self) \triangleq$ $\land\ pc[self] = $ “proc_enter”
    $\land\ stack' = [stack$ EXCEPT $![self] = \langle[procedure \mapsto$ “cclib_enter”,
                                             $pc \mapsto$ “proc_in”$]\rangle$
                                             $\circ\ stack[self]]$
    $\land\ pc' = [pc$ EXCEPT $![self] = $ “cclib_enter_begin”$]$
    $\land$ UNCHANGED $\langle cq, lq, pendingSignals, inPurgatory,$
                  $waiting, lastSignal, entryOrder,$
                  $wcid, scid, cclwcid, cclscid\rangle$

$proc\_in(self) \triangleq$ $\land\ pc[self] = $ “proc_in”
    $\land\ entryOrder' = Append(entryOrder, self)$
    $\land\ pc' = [pc$ EXCEPT $![self] = $ “proc_await”$]$
    $\land$ UNCHANGED $\langle cq, lq, pendingSignals, inPurgatory,$
                  $waiting, lastSignal, stack, wcid, scid,$
                  $cclwcid, cclscid\rangle$

$proc\_await(self) \triangleq$ $\land\ pc[self] = $ “proc_await”
    $\land\ \exists\ cid \in CId :$
        IF $cid \neq 0$
            THEN $\land$ $\land\ cclwcid' = [cclwcid$ EXCEPT $![self] = cid]$
                   $\land\ stack' = [stack$ EXCEPT $![self] = \langle[procedure \mapsto$ “cclib_await”,
                                                            $pc \mapsto$ “proc_signal”,
                                                            $cclwcid \mapsto cclwcid[self]]\rangle$
                                                            $\circ\ stack[self]]$
                 $\land\ pc' = [pc$ EXCEPT $![self] = $ “cclib_await_begin”$]$
            ELSE $\land\ pc' = [pc$ EXCEPT $![self] = $ “proc_signal”$]$
                 $\land$ UNCHANGED $\langle stack, cclwcid\rangle$
    $\land$ UNCHANGED $\langle cq, lq, pendingSignals, inPurgatory,$
                  $waiting, lastSignal, entryOrder,$
                  $wcid, scid, cclscid\rangle$

$proc\_signal(self) \triangleq$ $\land\ pc[self] = $ “proc_signal”
    $\land\ \exists\ cid \in CId :$
        IF $cid \neq 0$
            THEN $\land$ $\land\ cclscid' = [cclscid$ EXCEPT $![self] = cid]$
                   $\land\ stack' = [stack$ EXCEPT $![self] = \langle[procedure \mapsto$ “cclib_signal”,
                                                            $pc \mapsto$ “proc_leave”,
                                                            $cclscid \mapsto cclscid[self]]\rangle$
                                                            $\circ\ stack[self]]$
                 $\land\ pc' = [pc$ EXCEPT $![self] = $ “cclib_signal_begin”$]$
            ELSE $\land\ pc' = [pc$ EXCEPT $![self] = $ “proc_leave”$]$
                 $\land$ UNCHANGED $\langle stack, cclscid\rangle$
    $\land$ UNCHANGED $\langle cq, lq, pendingSignals, inPurgatory,$
                  $waiting, lastSignal, entryOrder,$
                  $wcid, scid, cclwcid\rangle$

$proc\_leave(self) \triangleq$ $\land\ pc[self] = $ “proc_leave”
```

$$\begin{array}{l}
\land\ stack' = [stack\ \text{EXCEPT}\ ![self] = \langle [procedure \mapsto \text{"cclib\_leave"}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad pc \mapsto \text{"Done"}]\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \circ\ stack[self]] \\
\land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{"cclib\_leave\_begin"}] \\
\land\ \text{UNCHANGED}\ \langle cq, lq, pendingSignals, inPurgatory, \\
\qquad\qquad\qquad waiting, lastSignal, entryOrder, \\
\qquad\qquad\qquad wcid, scid, cclwcid, cclscid\rangle
\end{array}$$

$$\begin{array}{l}
procid(self) \triangleq proc\_enter(self) \lor proc\_in(self) \\
\qquad\qquad\qquad \lor\ proc\_await(self) \lor proc\_signal(self) \\
\qquad\qquad\qquad \lor\ proc\_leave(self)
\end{array}$$

$$\begin{array}{l}
Next \triangleq (\exists\ self \in ProcSet : \ \lor\ java\_lock(self) \lor java\_unlock(self) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ java\_await(self) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ java\_signal(self) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ cclib\_enter(self) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ cclib\_leave(self) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ cclib\_await(self) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ cclib\_signal(self)) \\
\qquad \lor\ (\exists\ self \in ProcId : procid(self)) \\
\qquad \lor\ \text{Disjunct to prevent deadlock on termination} \\
\qquad\quad ((\forall\ self \in ProcSet : pc[self] = \text{"Done"}) \land \text{UNCHANGED}\ vars)
\end{array}$$

$$\begin{array}{l}
Spec \triangleq \ \land\ Init \land \Box[Next]_{vars} \\
\qquad\quad\ \land\ \mathrm{WF}_{vars}(Next)
\end{array}$$

$$Termination \triangleq \Diamond(\forall\ self \in ProcSet : pc[self] = \text{"Done"})$$

END TRANSLATION

Invariants and temporal properties (theorems)

$$\begin{array}{l}
Types \triangleq \\
\qquad \text{Global variables} \\
\qquad \land\ cq \in [CId \rightarrow Seq(ProcSet)] \\
\qquad \land\ lq \in Seq(ProcSet) \\
\qquad \land\ pendingSignals \in Nat \\
\qquad \land\ inPurgatory \in Nat \\
\qquad \land\ waiting \in [CId \rightarrow Nat] \\
\qquad \land\ lastSignal \in (ProcSet \cup \{0\}) \times (ProcSet \cup \{0\}) \\
\qquad \text{Procedure } java\_await \\
\qquad \land\ wcid \in [ProcSet \rightarrow CId \cup \{defaultInitValue\}] \\
\qquad \text{Procedure } java\_signal \\
\qquad \land\ scid \in [ProcSet \rightarrow CId \cup \{defaultInitValue\}] \\
\qquad \text{Procedure } cclib\_await \\
\qquad \land\ cclwcid \in [ProcSet \rightarrow CId \cup \{defaultInitValue\}]
\end{array}$$

Procedure *cclib_signal*
$\land\ cclscid \in [ProcSet \rightarrow CId \cup \{defaultInitValue\}]$

THEOREM $TypePreservation \triangleq \Box Types$

---

0. $-$ *Mutex*

$Mutex \triangleq$
  $\forall\ pid1 \in ProcSet :$
    $\forall\ pid2 \in ProcSet :$
      $pc[pid1] = \text{“proc\_in”} \land pc[pid2] = \text{“proc\_in”} \Rightarrow pid1 = pid2$

---

1. $-$ Characteristic property: *La* que hemos chequeado con *UPPAAL*, que si se hace un signal sobre una condition no vacía, el siguiente thread en tomar acceso del monitor es el primero de dicha condition, y no alguno de los que estaban en la cola del lock mutex.

As an invariant with the instrumentation:
$CharacteristicAsInv \triangleq$
  $\forall\ pid \in ProcSet : pc[pid] = \text{“proc\_in”} \Rightarrow lastSignal[2] \in \{0, pid\}$

---

2. $-$ *ToHeaven*: se acaba saliendo del purgatorio

$InPurgatory(pid) \triangleq Member(cq[0], pid)$

$ToHeaven \triangleq \forall\ pid \in ProcSet : (InPurgatory(pid) \leadsto pc[pid] = \text{“proc\_in”})$

BY Auto
THEOREM $ToHeavenThm \triangleq InPurgatory(1) \leadsto pc[1] = \text{“proc\_in”}$
  $\langle 1 \rangle 1.$ ASSUME $InPurgatory(1) \land [Next]_{vars} \Rightarrow (InPurgatory(1)' \lor pc'[1] = \text{“proc\_in”}'),$
    $InPurgatory(1) \land \langle Next \rangle_{vars} \Rightarrow pc'[1] = \text{“proc\_in”},$
    $InPurgatory(1) \Rightarrow$ ENABLED $\langle Next \rangle_{vars}$
    PROVE $\Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next) \Rightarrow (InPurgatory(1) \leadsto pc[1] = \text{“proc\_in”})$
   BY $RuleWF1$
  $\langle 1 \rangle$.QED

---

3. $-$ *OrderPreservation*:

$NoOrderPreservation \triangleq$
  $\forall\ pid1 \in ProcSet :$
    $\forall\ pid2 \in ProcSet :$
      $(\land\ Member(cq[0], pid1)$
        $\land\ pc[pid2] = \text{“java\_lock\_blocked”})$
      $\leadsto$
      $(\land\ Member(entryOrder, pid1)$
        $\land\ Member(entryOrder, pid2)$

$$\quad \land\ Index(entryOrder,\ pid2) < Index(entryOrder,\ pid1))$$

$$OrderPreservation \triangleq \neg\ NoOrderPreservation$$