Reasoning on Concurrency: An Approach to Modeling and Verification of Java Thread-safe Objects

Franco Cicirelli, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria - 87036 Rende (CS) – Italy
Email: f.cicirelli@deis.unical.it, {l.nigro,f.pupo}@unical.it

***Abstract*–Development of concurrent and time-dependent software systems is currently growing in its strategic importance due to the diffusion of powerful multi-core/many-core machines. To effectively cope with current and prospective concurrency demands, formal tools have to be used. A library of reusable UPPAAL timed automata was achieved, which enables a reasoning on concurrency. The library is tailored to Java. However, similar solutions could be also developed to work with other languages as well. This paper outlines library design and focuses on its exploitation for model-based prediction of the correctness of thread-safe Java objects.**

Keywords-Modeling and verification; concurrent systems; Java; thread-safe objects; model checking; UPPAAL.

I. INTRODUCTION

This work argues that to properly design and implement concurrent systems, the adoption of formal tools, which can support a *reasoning on concurrency* is mandatory.

This paper introduces the design of a UPPAAL library of timed automata [1-3], which improves preliminary experience described in [4]. Library design was mainly inspired by Java concurrency features. Common synchronization mechanisms such as semaphores and monitors, both classic and Java specific, are provided. On top of these mechanisms, new synchronizers, tailored to particular programming styles, can be built. The library enables modeling of a concurrent program according to implementation aspects, thus reducing the semantic gap which normally exists between a specification model and its vocabulary (e.g., atomic actions, broadcast synchronization, etc.) and a corresponding implementation. Analysis activities are based on exhaustive verification and model checking [5], [6].

The paper proposes an approach to modeling and verification (M&V) of Java thread-safe objects and illustrates it by practical examples.

The model-based prediction approach can be related to the work of Hamberg and Vaandrager [7] and to the known Finite State Processes (FSP)/Labeled Transition System Analyzer (LTSA) tool developed by Magee and Kramer [8]. With the first work our approach shares the use of the UPPAAL model checker. However, our library is characterized by its volition of being Java tailored. In addition, some common structures like semaphores appear to be more efficient. The second approach is based on the FSP process algebra. An FSP model is transformed into a labeled transition system to be analyzed by the LTSA tool. However, this approach does not allow, for instance, the expression of a FIFO policy (e.g., for awaking processes waiting on a semaphore). In addition, the use of a discrete time model can complicate the verification of complex models.

The paper is structured as follows. Section II outlines the developed UPPAAL. Section III proposes an approach for M&V of Java thread-safe objects. Section IV describes a more complex modeling example. Finally, an indication of research directions which deserve further work is given in the conclusion.

II. CONCURRENCY CONTROL IN UPPAAL

A library of UPPAAL template processes (i.e., timed automata –TA–) was developed, which provides such common concurrent structures as semaphores and monitors [9], both classic and Java specific. The following gives an outline of the library contents.

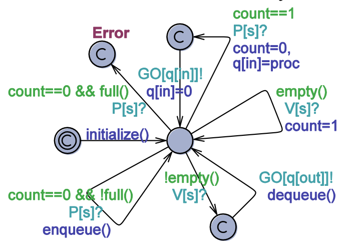


Figure 1. The BinarySemaphore automaton

Fig. 1 shows an automaton for classic binary semaphore. A similar construction exists for a general, counting semaphore. Classic P/V operations are implemented as unicast channel arrays P[.]/V[.] whose dimension mirrors the number of semaphores used in a model. A P operation on a semaphore s is expressed by raising a synchronization P[s]!. The requesting process is assumed to put into a global (meta) variable proc its unique process id at the time of P[s]!. Variable proc is used only during the atomic action of P[s]!, with the receiving semaphore which frees it immediately by storing the proc value in a local variable. A further channel array GO[.], whose dimension coincides with the number of processes in the model, is used for blocking the requesting process until the semaphore assigns the permit to the process. The use of GO is implicit in the operation P in a programming language, but in UPPAAL it serves the purpose of transforming a *strict rendezvous* (P[.]!) into an *extended rendezvous*, which terminates when the semaphore completes the handling of P[.]! and allows the requesting process to unblock. A V[s]! request never blocks the requesting process and normally does not require the proc mediation.

With respect to the realization proposed in [7], our semaphores use less variables thus favoring model checking. For instance, the identity of the requesting process during a P operation, which finds a binary semaphore green (count==1) is temporarily stored in the surely empty internal queue of the semaphore.

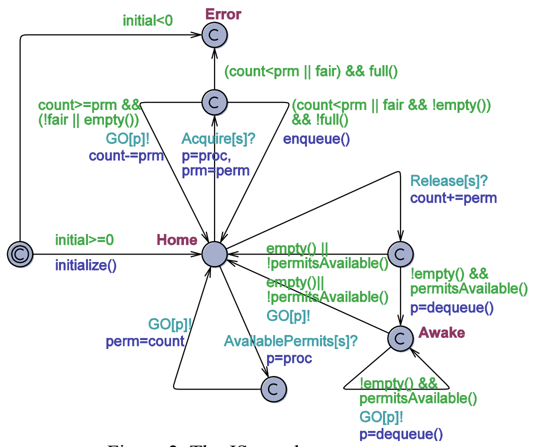


Figure 2. The JSemaphore automaton

A semaphore automaton (JSemaphore) directly based on a provided Java (java.util.concurrent.Semaphore) class is shown in Fig. 2. Differences from classic semaphores concern the possibility of acquiring/releasing atomically a number of permits greater than 1. In addition, a fair parameter can be used to request a FIFO behavior of acquire requests. This way, an acquire operation, in the case there are some waiting processes, puts the requesting process at the end of the semaphore queue even if permits are available.

JSemaphore relies on an interface consisting of channel arrays Acquire[.], Release[.], PermitsAvailable[.], GO[.], and exploits two global variables: proc and perm. The perm variable stores, at the time of an Acquire[s]! or Release[s]!, the number of involved permits, and contains the number of available permits of the semaphore just after a PermitsAvailable[s]! operation. A GO[p]? synchronization must follow an Acquire[s]! or a PermitsAvailable[s]! command. It is at the time of a GO[p]? unblocking operation, that perm is actually filled of the semaphore permits number.

It is worth noting that whereas a burst of release operations on a JSemaphore instance used as a mutex, will increase the permits number arbitrarily, in the case of a BinarySemaphore, a burst of V's can never augment the internal count beyond 1.

Although widely used, semaphores are often viewed as a low level concurrent abstraction mechanism, where a misuse of P/V operations can easily lead to a deadlock. Monitors (e.g., [9]), on the other hand, represent a higher level concurrent control structure, which naturally acts as a guardian of an abstract data type, e.g., encapsulated into a Java class. Monitors are a key for achieving thread-safe classes by offering control over mutual exclusion among class methods (synchronized blocks or critical sections of code) and suspension/signaling from within a critical section. Different kinds of monitors are defined in the literature, which are characterized by different programming styles and guarantees/obligations that are assigned to both processes and the control structure.

Java adopts, as a basic solution, the Lampson & Redell [10] monitor structure with broadcast signaling, where suspended processes in a synchronized block are responsible of re-checking a condition in a while loop to see, at each awaking, if it is necessary to coming back to waiting or the process can finally exit its waiting status. Broadcast signaling does not block the executing process. An awaken process has to compete in reacquiring the lock for it to actually resume execution.

Directly based on the built-in Java monitor structure is the JMonitor automaton presented in Fig. 3.

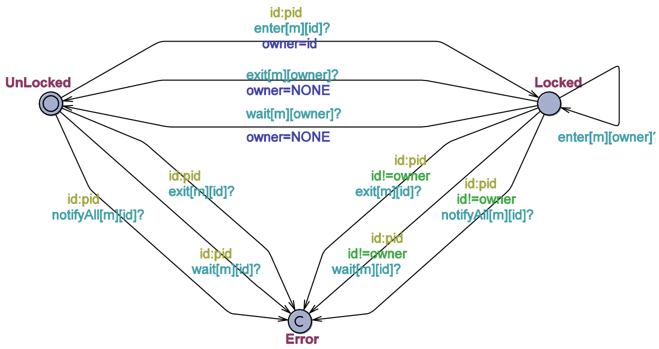


Figure 3. The JMonitor automaton

A monitor instance can be operated using such channel arrays as enter[mid][pid], exit[mid][pid], wait[mid][pid], notifyAll[mid][pid], which accommodate for the possible existence of multiple monitor instances in a model. Types mid and pid respectively are integer sub-ranges of unique identifiers of monitors and processes used in the model. For instance, enter[m][p]!/exit[m][p]! are used by a process p to explicitly enter/exit to/from a synchronized block based on monitor m. Similarly, wait[m][p]!/notifyAll[m][p]! serve respectively to suspend the requesting process p until its condition holds (in a while loop), and to awake all the processes suspended on monitor m.

Every Java object has an implicit lock which can be used as a monitor. The lock holds one implicit condition, whose meaning is established by the modeler/programmer. The lock object is associated with a *wait-set* where both entering processes which find the lock closed, and processes within a synchronized block (based on the lock object) whose condition prescribes waiting, are put (although the two kind of waiting processes are clearly distinguished to one another). Processes which are suspended for a wait operation can only be awaken by a notifyAll operation. The notifyAll operation does not free the lock. Other processes awake as the lock/monitor is up to be abandoned. In the proposed implementation, the wait-set is purposely realized implicitly. Processes requesting an enter are simply blocked if the monitor is already locked. Processes which execute wait are supposed to move into a location (see W1 and W2 in Fig. 6) from which they can only exit following a relevant notifyAll[.][.]? signal. Towards this, channels notifyAll[.][.] are declared as broadcast. Following a notifyAll signal, an awaken process has to compete for re-acquiring the lock (see edges exiting the W1 or W2 location in Fig. 6). Whereas this is implicit in the Java wait() method, it is explicit in the proposed modeling pattern, thus revealing a semantic issue.

The automaton in Fig. 3 maintains the identity of the monitor owner, which is used to realize reentrancy and to check for erroneous operations, which in Java correspond to raising an IllegalMonitorStateException, e.g., invoking a wait or notifyAll operation out of a synchronized block. The implicit realization of the wait-set complies with the Java specification and lets processes which try to enter the monitor and awaken processes to be handled non deterministically and thus without any privilege. The design makes it possible to implement a *timed wait*. In this case, from the wait location (now provided of a clock invariant) the process can also exit when the clock goes beyond a given time limit (*timeout*).

In reality, the Java built-in monitor also exposes a notify operation to awake *one*, although unspecific, process which is suspended in the wait-set. For generality reasons, the automaton in Fig. 3 only implements the notifyAll operation because, as discussed in [11], the use of notify can cause a *Lost-Wakeup-Problem*, i.e., a notify signal can be lost and the system enters a bad status.

On the basis of JMonitor, a monitor structure based on the Java Lock/Condition framework was also achieved, which allows to split the waiting processes among different conditions (*waiting rooms*) associated with a given lock. The signaling mechanism can be directed to a specific condition.

The library also includes some other classic monitors like the Hoare monitor [9], which in a case can be built on top of semaphores. The Hoare monitor owns a different signaling mechanism: when a process (*signaler*) changes the status of the data structure so that a (possibly) waiting process (*signalee*) on a condition can be awaken because the condition holds, control is immediately transferred to the signalee (together with the lock), which is thus the only process which can proceed. The signaler, on the other hand, is put to wait in an urgent queue from where it gets unblocked as soon as the monitor is up to become free.

A discussion about Lampson & Redell vs. Hoare monitors can be found in [9] where it is argued, besides any runtime implication (e.g., number of context switches), that Lampson & Redell monitor can be superior in the most general case.

III. AN APPROACH TO M&V OF THREAD-SAFE OBJECTS

In the following, the proposed M&V approach is demonstrated by achieving a synchronizer based on two-way (or *rendezvous*) communications. The modeling example, which is original, can be used as a (partial) proof for programming a class like Exchanger<T> as provided in the java.util.concurrent package. The mechanism is intended to be used by two processes, which both play the sender/receiver roles. When the time arrives for a synchronization/communication, the earliest process which comes to the appointment awaits the partner. When the latest process arrives, processes exchange some information, then exit the synchronization thus returning to concurrent execution. In particular, the exchanger allows each process to send a message to the partner and to receive the message sent by the partner. Obviously, the mechanism can easily reproduce a CSP synchronous communication, when a partner is assigned the sender role and the other the receiver role.

An exchange synchronizer can be easily modeled by using the UPPAAL native features, as depicted in Fig. 4.

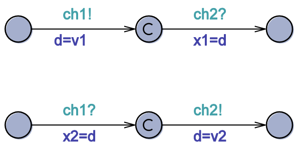


Figure 4. A native UPPAAL model for an exchanger

Fig. 4 assumes that each process transmits a local value v_i and receives from the partner an information to be stored in local variable x_i. A (meta) global variable d is used for the information exchange. Two unicast channels ch1 and ch2 are used where, for instance, ch1 can be declared as urgent. A committed location ensures an atomic exchange of information. Once the synchronization starts, on ch1, it is immediately followed (without a time passage) by a synchronization on ch2. Correctness of the model in Fig. 4 depends, among the other, on the fact that in a channel synchronization, the update (e.g., d=v1) of the sender (e.g., ch1!) is executed *before* the update (x2=d) of the receiver.

However, a native model like that in Fig. 4 cannot be immediately translated into Java, simply because the UPPAAL vocabulary of atomic actions, urgent channels, committed locations, etc. *is not* supported in the target language. The modeler/programmer is thus forced to intuitively achieve Java code by using the basic vocabulary of Java, e.g., synchronized blocks, wait/notifyAll, etc. However, the problem remains that an implementation *cannot* be proved to be a faithful representation of its specification model.

```
public interface Exchanger<T> {
  T exchange( T v );
}//Exchanger

public class ExchangerMJ<T> implements Exchanger<T>{
  private T d;
  private boolean partner = false, release = false;
  private Object m = new Object(); //lock/monitor object
  public T exchange( T v ){
    synchronized( m ){
      while( release ) //protection from a prompt re-enter
        try{ m.wait(); }catch( InterruptedException e ){}
      T x=null;
      if( !partner ){
        d = v; partner = true;
        while( partner ) //waiting for partner
          try{ m.wait(); }catch( InterruptedException e ){}
        x = d; release = false;
        m.notifyAll();
      }
      else{
        x = d; d = v; partner = false; release = true;
        m.notifyAll();
      }
      return x;
    }
  }//exchange
}//ExchangerMJ
```
Figure 5. A Java thread-safe class for the exchanger

To reduce the semantic gap existing between an UPPAAL model and a Java code, the model can embody implementation aspects. Stated in other terms, the model can mirror, through reverse-engineering, some Java code.

In Fig. 5, it is shown a Java code realizing the exchanger synchronizer. The Exchanger<T> interface exposes only the method exchange(v), which transmits v and receives the value sent by the partner. The ExchangerMJ<T> class implements, in a case, the Exchanger<T> interface in terms the of the native Java monitor.

Two waiting points exist in the exchange method in Fig. 5: one when the first process arrives and finds the partner is lacking (partner is false); another one for blocking a process, which finds a synchronization release in progress. When the latest process comes to the synchronization point, it finds partner=true, takes the transmitted data and sends its own information, then assigns false to partner, states that a release is up to commence, executes notifyAll and, finally, exits the synchronization block. Now, it is possible for the just exited process, to re-enter immediately the monitor whereas the last synchronization is still not finished. Apart for data overwriting problems, from Fig. 5 it is easy to see that a deadlock situation would occur.

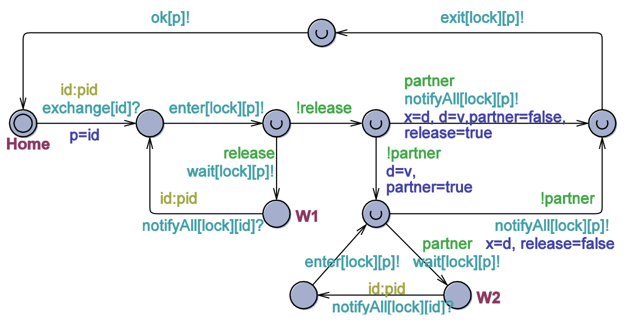


Figure 6. The Exchanger automaton based on JMonitor

Fig. 6 portrays a UPPAAL model for the ExchangerMJ class (integers are the exchanged data), which is based on the JMonitor automaton. Two arrays of channels are used: exchange[.] and ok[.], whose dimension is the number of processes. A process p requests an exchange through the operation exchange[p]! and then blocks on receiving an ok[p]? synchronization (see e.g., Fig. 7). Exchanger receives as a parameter the unique identifier of the lock object to be used internally for the synchronization. Since each participating process can wait at a different point in the control structure, two instances of the Exchanger automaton must be created, each serving a different process. Each instance links to the requisting process through a nondeterministic select (see the edge outgoing from Home in Fig. 6). In addition, information of the Exchanger (e.g., variables partner, release, etc.) are to be declared global to the model.

The use-pattern of Exchanger model in Fig. 6 is illustrated in Fig. 7 and Fig. 8, where a Producer sends the sequence 1, 2, 3 to a Consumer. Both processes are characterized by a process identifier (p) established at configuration time.

For proper behavior of the application, it is important that the consumer receives exactly the same data transmitted by the producer at each synchronization point. To check correctness of this behavior, the consumer model reaches the Error location as soon as it discovers an incorrect received data.

A system can be configured as indicated in Fig. 9.

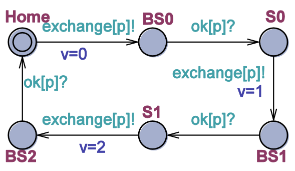


Figure 7. Producer automaton

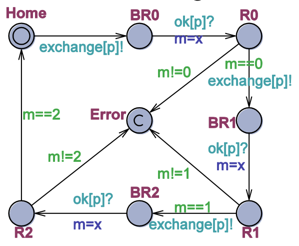


Figure 8. Consumer automaton

```
// Place template instantiations here.
ex0=Exchanger(LOCK);
ex1=Exchanger(LOCK);
//ex0=OptimisticExchanger(LOCK);
//ex1=OptimisticExchanger(LOCK);
prod=Producer(PROD);
cons=Consumer(CONS);
// List one or more processes to be composed into a system.
system JMonitor,ex0,ex1,prod,cons;
```

Figure 9. Producer/consumer system configuration

The following queries were issued to the UPPAAL model checker. The answers confirm all the queries are satisfied.

1) A[] !deadlock
2) A[] cons.R0 imply (prod.BS0 || prod.S0 || prod.BS1)
3) A[] cons.R1 imply (prod.BS1 || prod.S1 || prod.BS2)
4) A[] cons.R2 imply (prod.BS2 || prod.Home || prod.BS0)

Since there are no deadlocks, the consumer is guaranteed it never reaches the Error location. Query 2) ensures that when the consumer receives 0, the prod(ucer can't have completed the transmission of the next int. The producer completes the transmission of 1 when it enters the S1 location in Fig. 7. Similar considerations apply to queries 2) and 3).

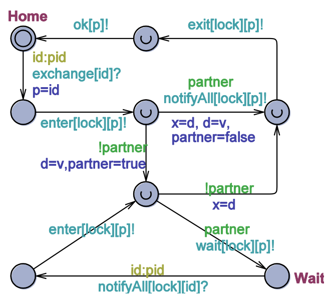
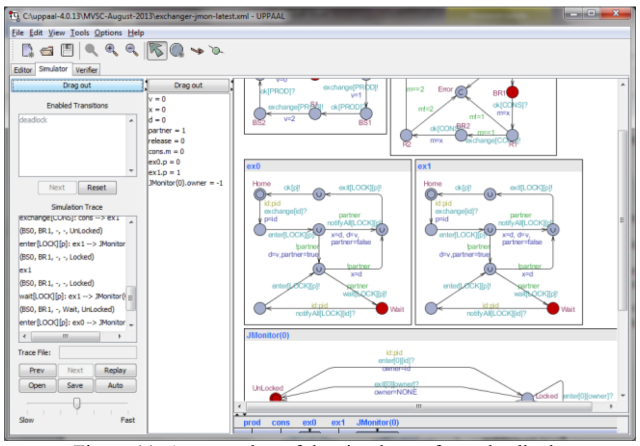


Figure 10. The OptimisticExchanger automaton

An optimistic variant of the Exchanger is shown in Fig. 10, which differs from Fig. 6 only because the first waiting point is eliminated.


C:\uppaal-4.0.13\MVSC-August-2013\exchanger-jmon-latest.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Drag out
Enabled Transitions
deadlock
Next Reset
Simulation Trace
Trace File:
Prev Next Replay
Open Save Auto
Slow Fast
prod cons ex0 ex1 JMonitor(0)

```
v = 0
x = 0
d = 0
partner = 1
release = 0
cons.m = 0
ex0.p = 0
ex1.p = 1
JMonitor(0).owner = -1
```

Figure 11. A screenshot of the simulator after a deadlock

By adjusting the system configuration in Fig. 9 so as to include two instances of the OptimisticExchanger template, it emerges that the first query is no longer satisfied. Asking the verifier to generate a diagnostic trace and opening it in the simulator, confirms the model is deadlocked (see Fig. 11).

For demonstration purposes, Fig. 12 shows an exchanger model based on semaphores. Two binary semaphores (with identifiers MUTEX and WAIT) are used: one as a mutex (initialized to 1), the other as a waiting room (initialized to 0).

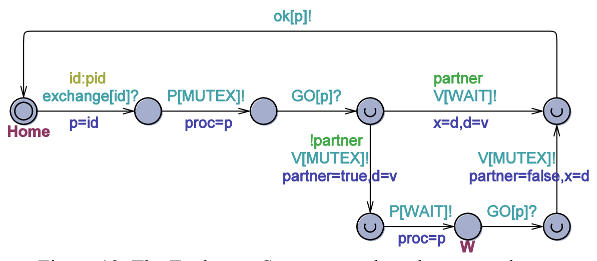


Figure 12. The ExchangerS automaton based on semaphores

Model in Fig. 12 rests on the "*pass the baton*" design pattern [12], i.e., when the latest process arrives (partner=true), it awakes the partner through a V[WAIT]! operation and then exits without releasing the mutex. As a consequence, the earliest process gets awaken *with* the mutex transferred to it. Therefore, a prompt re-enter is thus forbidden because mutex is occupied, and the release variable of Fig. 6 is useless.

Also, a system based on ExchangerS was configured and found correct by model checking.

As a final remark, except for the GO[.] synchronizations, the model in Fig. 12 directly maps on Java code.

IV. SECOND M&V EXAMPLE

The concept of a binary semaphore, corresponding to the UPPAAL model in Fig. 1, can be introduced in Java through a class as shown in Fig. 13. The realization relies on the language native monitor.

In order to check the correctness of the BinarySemaphore class, it was modeled in UPPAAL as depicted in Fig. 14, using the JMonitor automaton and the approach described in section III. Since the BinarySemaphoreMJ rests on JMonitor, it is assumed that also at the time of a V[.] operation the requesting process assigns its identifier to the global proc variable.

A notable difference between the automaton in Fig. 14 and the Java code in Fig. 13 concerns the realization of the linked waiting list, which in Fig. 14 is based on a bounded array managed as a FIFO queue. In addition, the toAwake integer variable is turned into a bounded int variable of UPPAAL, whose upper bound is qs+1 where qs is the queue size, established through a parameter of the BinarySemaphoreMJ template. As a consequence, when executing a burst of V's, it is useless to advance toAwake beyond this upper limit.

A key point of the model in Fig. 14 is the use of committed locations. The goal is to ensure that a P or V operation, once started, is conducted to a conclusion (i.e., re-entering the Home location or reaching the WaitTrue location) in an atomic way.

```
public interface Semaphore {
   void P();
   void V();
}//Semaphore

public class BinarySemaphore implements Semaphore{
   private int count, toAwake=0;
   private List<Thread> waitList=new LinkedList<Thread>();
   private Object m=new Object(); //lock-monitor object
   public BinarySemaphore ( int count ){
      if( count <0 || count >1 ) throw new IllegalArgumentException();
      this. count = count;
   }
   public void P(){
      synchronized( m ){
         if( count==0 ){
            waitList.add( Thread.currentThread() );//arrival order
            while( true ){
               try{ m.wait(); }catch( InterruptedException e ){}
               if( toAwake>0 &&
                  waitList.get(0)==Thread.currentThread() ){
                  toAwake--; waitList.remove(0);
                  if( toAwake>0 ){
                     if( waitList.size()>0 ) m.notifyAll();
                     else{ count =1; toAwake=0; }
                  }
                  break;
               }
            }//while
         }
         else count=0;
      }
   }//P
   public void V(){
      synchronized( m ){
         if( waitList.size()==0 ) count=1;
         else{ toAwake++; m.notifyAll(); }
      }
   }//V
}//BinarySemaphore
```

Figure 13. A FIFO BinarySemaphore Java class

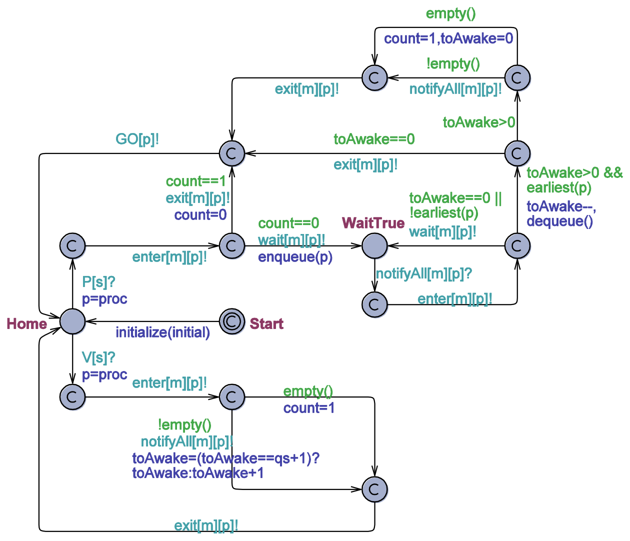


Figure 14. The BinarySemaphoreMJ automaton

An experimental verification frame was designed with the aim of comparing one instance (identifier S1) of BinarySemaphore in Fig. 1 and one instance (identifier S2) of BinarySemaphoreMJ in Fig. 14. Both instances receive a same sequence of P/V operations and the goal was to assess that both instances evolve exactly in the same way. Three process automata (see Fig. 15) were prepared: pProcess1, which acts on semaphore S1, pProcess2 which operates on semaphore S2, and vProcess which uses both S1 and S2. Process instances receive as parameters: the unique process identifier (pid) p, and the names of used semaphores.

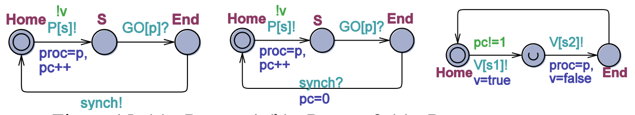


Figure 15. (a) pProcess1 (b) pProcess2 (c) vProcess automata

Since V operations are not blocking, one single instance of vProcess can be used to ensure that a V is actually issued to S1 *and* S2. Because a P operation can block the requesting process, one instance of pProcess1 and one instance of pProcess2 were used. A critical point in the process design was how to guarantee atomicity of the blocks {P[S1], P[S2]} and {V[S1], V[S2]}. Towards this, a global bool variable v is used, which is true if the V block is in execution. For the atomicity of the first block a global counter pc is employed, which is incremented each time a PO operation is launched. A V block can be started provided a P block is not in progress and similarly a P block can be started if no V block is in progress. Exiting a P block (see End location in Fig. 16 and Fig. 17) is ensured by a unicast channel signal synch, which is raised by a pProcess and received by other. At each synchronization over synch, pc is reset. Similarly, at the time of the second V of a V block, the variable v is reset. It should be noted that Fig. 15 (c) guarantees that a burst of V blocks can occur. Obviously, the semaphores S1 and S2 are supposed to be initialized to the same value.

The actual system configuration used for the verification experiments is shown in Fig. 16.

```
// Place template instantiations here.
bs=BinarySemaphore( S1, 0, PROC-2 );//id, init val, queue size
BS0=BinarySemaphoreMJ( S2, MON, 0 );//id, mon id, init val
BS1=BinarySemaphoreMJ( S2, MON, 0 );
p0=pProcess1( P0, S1 );
p1=pProcess2( P1, S2 );
v2=vProcess( V2, S1, S2 );
// List one or more processes to be composed into a system.
system JMonitor,bs,BS0,BS1,p0,p1,v2;
```
Figure 16. System configuration for the experimental frame

The following queries (all satisfied) were used for model checking:

1) A[] !deadlock
2) A[] p0.Home && p1.Home && v2.Home && BS0.Home && BS1.Home && bs.Home imply bs.count==count && empty() && bs.empty()
3) E<> p0.Home && p1.Home && v2.Home && BS0.Home && BS1.Home && bs.Home && count==0 && count==bs.count
4) E<> p0.Home && p1.Home && v2.Home && BS0.Home && BS1.Home && bs.Home && count==1 && count==bs.count
5) A[] p0.S && p1.S && (BS0.WaitTrue||BS1.WaitTrue) imply count==0 && bs.count==count && size()==1 && size()==bs.size() && first()==P1 && bs.first()==P0

Query 2) guarantees that when all automata are in their Home location, the semaphores have the same count value and their queues are both empty. Queries 3) and 4) show that in the same states of query 2), the semaphores can be both green or red. Query 5) verifies that when processes p0 and p1 are in the S location, i.e., they have both requested a P operation, in the case the BinarySemaphoreMJ is in the WaitTrue location, it effectively follows that both semaphores are red (count==0), their internal queues have the same size and in particular P0 is waiting in S1 and P2 is waiting in S2.

Meaning of queries from 2) to 5) ensures that after each *complete* execution of a block of P or V operations, the two semaphores have equivalent states.

Although the above described verification frame cannot replace a formal (weak) bisimulation proof of the two automata in Fig. 1 and Fig. 14, it provides important information about the correct behavior of BinarySemaphoreMJ and then of the Java thread-safe class in Fig. 13.

All the verification experiments were carried out on a Win 8, 12GB, Intel Core i7-3770K, 3.50GHz.

V. Conclusion And Future Work

The Uppaal timed automata library and the M&V approach for concurrent systems proposed in this paper are useful in the practical case, and are under experimentation in an undergraduate course on systems programming. The solutions, although inspired by Java concurrency, can be adapted to work with other concurrent programming languages as well.

On-going and future work is geared at:

- Improving the supporting library of basic concurrent control structures and synchronizers.
- Extending the library in order to experiment with alternative but influencing concurrency schemes like the *software transactional memory* [11], which in the next future should be made available, e.g., in Java.

Acknowledgments

Authors thank Christian Nigro for his contribution to the development of the M&V approach described in this paper.

References

[1] R. Alur and D.L. Dill, “A theory of timed automata,” Theoretical Computer Science, vol. 126, no. 2, 1994, pp. 183-235.
[2] G. Behrmann, A. David, and K.G. Larsen, “A tutorial on Uppaal,” In Formal Methods for the Design of Real-Time Systems, LNCS 3185, Springer, 2004, pp. 200-236.
[3] Uppaal on-line, www.uppaal.org
[4] F. Cicirelli, L. Nigro, and F. Pupo, “Modelling and verification of concurrent programs using Uppaal,” ECMS’2011, 2011, pp. 525-533.
[5] F. Cicirelli, A. Furfaro, and L. Nigro, “Model checking time-dependent system specifications using Time Stream Petri Nets and Uppaal,” Applied Mathematics and Computation, vol. 218, no. 16, 2012, pp. 8160-8186, Elsevier.
[6] E.M. Clarke, O. Grumberg, and D.A. Peled, “Model Checking,” Cambridge, MA, MIT Press, 1999.
[7] R. Hamberg and F. Vaandrager, “Using model checkers in an introductory course on operating systems,” Operating System Review, vol. 42, no. 6, 2008, pp. 101-111.
[8] J. Magee and J. Kramer, “Concurrency – State models and Java programming,” John Wiley & Sons, Ltd., 2006.
[9] W. Stallings, “Operating Systems: Internals and design principles,” Prentice-Hall, 2005.
[10] B.W. Lampson and D.D. Redell, “Experience with processes and monitor in Mesa,” In Proc. of SOSP, 1979, pp. 43-44.
[11] M. Herlihy and N. Shavit, “The art of multiprocessor programming,” Elsevier, Revised version of First Edition, Morgan & Kaufmann Publishers, 2012.
[12] A.K. Reek, “Design patterns for semaphores,” ACM SIGCSE’04, 2004.