MODELLING AND VERIFICATION OF CONCURRENT PROGRAMS USING UPPAAL

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

KEYWORDS
Modelling and verification, simulation, concurrency, mutual exclusion, synchronizers, timed automata, UPPAAL, Java.

ABSTRACT

This paper describes the design and implementation of a library of reusable UPPAAL template processes which support reasoning and property checking of concurrent programs, e.g. to be realized in the Java programming language. The stimulus to the development of the library originated in the context of a systems programming undergraduate course. The library, though, can be of help to general practitioners of concurrent programming which nowadays are challenged to exploiting the potentials of modern multi-core architectures. The paper describes the library and demonstrates its usage to modelling and exhaustive verification of mutual exclusion and common concurrent structures and synchronizers. UPPAAL was chosen because it is a popular and continually improved toolbox based on timed automata and model checking and it is provided of a user-friendly graphical interface which proves very important for debugging and property assessment of concurrent models. Java was considered as target implementation language because of its diffusion among application developers.

INTRODUCTION

Current technological trend on multi-core machines challenges developers to exploit concurrency in general purpose applications which can have a performance gain from the computational parallelism offered by modern personal computers. However, as students and developers know, concurrent programs are hard to design and difficult to debug. Common experimented problems include race conditions, deadlocks and starvation (Stallings, 2005)(Silberschatz et al, 2010). Motivated by the desire to help students of a systems programming undergraduate course to have a more critical approach to concurrent programming, authors have designed and prototyped a reausable library of UPPAAL (Bengtsson and Yi, 2003)(Behrmann et al., 2004) template processes. The library enables a concurrent solution to be formally modelled as a network of timed automata (Alur and Dill, 1994), to animate it in simulation to check qualitative behaviour thus making a preliminary debug, and to prove (provided the model is not too large) functional/temporal properties of the system at hand through model checking (Clarke et al., 2000)(Cicirelli et al., 2007)(Cicirelli et al., 2009)(Furfaro and Nigro, 2007). The approach is similar but independent and original with respect to that described in (Hamber and Vaandrager, 2008). A key factor of the work described in this paper concerns the development of concurrent structures and synchronizers which are inspired by the concurrent package of the Java programming language. The UPPAAL toolbox was chosen because it is popular, it is continually improved and it is efficient (in space and time) in the handling of large model state graphs. Moreover, the toolbox offers a friendly graphical user interface which facilitates reasoning upon model behaviour.
This paper describes (part of) the developed library and demonstrates its usefulness by studying mutual exclusion algorithms and by showing some common concurrent synchronizers which are available in the Java programming language. Concurrent models are then applied to a sample problem. The approach makes it simple to transform a concurrent solution model into a corresponding Java implementation. The solutions, though, can be ported to other languages as well. Finally, conclusions are drawn with an indication of further work.

MUTUAL EXCLUSION ALGORITHMS

Concurrent processes accessing shared data require two kinds of mechanisms (see e.g. (Stallings, 2005)(Silberschatz et al., 2010)): mutual exclusion which guarantees only one process at a time can enter its critical section, and synchronization, i.e. the possibility for a process in a critical section to suspend its execution when the data values do not permit the process to complete its operations. In this section the focus is on mutual exclusion based on busy-waiting by “pure software” solutions (other solutions can be based on the hardware support, e.g. test and set instructions or the interrupt system). Such mutual exclusion algorithms are normally discussed in a systems programming course for introducing students to race conditions and

Proceedings 25th European Conference on Modelling and Simulation ©ECMS Tadeusz Burczynski, Joanna Kolodziej Aleksander Byrski, Marco Carvalho (Editors)
ISBN: 978-0-9564944-2-9 / ISBN: 978-0-9564944-3-6 (CD)

interference problems among concurrent processes. In the following, algorithms for N>2 processes are considered. Examples include the Bakery algorithm and the Eisenberg and McGuire algorithm ((Silberschatz et al., 2010) page 302). Fig. 1 shows a pseudo code of the generic process according to the Eisenberg and McGuire algorithm.

```
//shared variables used by the algorithm
enum pState {idle, want_in, in_cs}
pState flag[n]; //all elements initialized to idle
int[0,n-1] turn; //no particular initialization
//ith process
int[0,n] j;
do{ //enter part
    while(true){
            flag[i]=want_in; //I want to enter my critical section
            j=turn; //give priority to non idle processes, if there
            //are any, from turn to i clockwise
            while(j!=i){ //busy waiting
              if( flag[j]!=idle ) j=turn;
              else j=(j+1)%n;
            }
            flag[i]=in_cs; //I "enter" my cs
            j=0;
            //it there exists in the entire ring a
            //process with in_cs status ?
            while( (j<n) && (j==i ||flag[j]!=in_cs) ) j++;
            if( (j>=n) && (turn==i || flag[turn]==idle) )
                /*no*/ break;
            //yes, waits
    }
    turn=i; //its my turn
    //critical_section
    //exit part
    //starting from me (turn==i)
    //search the first not idle process
    j=(turn+1)%n;
    while( flag[j]==idle ) j=(j+1)%n;
    turn=j; //give it its turn
    flag[i]=idle;
    //non_critical_section
}while(true)
```

Figure 1. Eisenberg and McGuire mutual exclusion algorithm for N processes

Now the goal is to model in UPPAAL the algorithm in Fig. 1 and proving it fulfils all the three basic properties: (a) only one process at a time can enter its critical section, (b) a process waiting for entering its critical section would not delay infinitely (absence of starvation), (c) no assumption is made about the relative speed of the processes. The modelling strategy purposely allows to concentrate on the essential of the algorithm while ensuring a certain efficiency of the model checking. The model abstracts away the duration of the single instructions carried out during the entry section and the exit section of the protocol, thus making it possible to determine the delay time of a process waiting to enter its critical section, in terms of the number and duration of critical sections executed by other processes. The “high resolution” approach, in which every single instruction is modeled and timed (e.g. each instruction consumes 1 time unit) is instead advocated in (Hamber and Vaandrager, 2008). Fig. 2 shows the proposed UPPAAL model for the generic Process of Eisenberg and McGuire algorithm. Duration of the critical section is supposed to be in the [2,6] time interval. The template Process receives its unique id i as parameter.
The use of committed locations mirrors the assumption that instructions executed during the entry/exit part are supposed to be time negligible with respect to the critical section duration. Of particular concern is the realization of the busy-waiting during the enter part. The process enters the Busy_Wait location from which it exits at each change of shared variables. To this purpose a broadcast channel check is used. The process which enters or exits from its critical section forces all processes in busy waiting to reconsider their situation. The following global UPPAAL declarations were used:

```
const int N=5; //number of processes
typedef int[0,N-1] pid; //process identifier subtype
typedef int[0,2] pState;
broadcast chan check;
const int idle=0;
const int want_in=1;
const int in_cs=2;
pid turn;
pState flag[N]={idle,idle,idle,idle,idle};
clock x[N]; //process clocks
clock y[N]; //decoration clocks
```

The system declaration section consists only of:

```
system Process;
```

which ensures, due to the pid parameter of the Process template, that N instances of the template are created to populate the model. These instances have names Process(0), ..., Process(N-1).

Table 1 shows the queries used to verify the mutual exclusion algorithm. Query 1 verifies the absence of deadlocks. Queries 2 and 3 check, with different syntax, the fundamental mutual exclusion property: no more than one process can find itself into the critical section. Queries 4 and 5 respectively determine minimal and maximal delay when waiting for entering the critical section. Query 4 is not satisfied if a value greater than 0 is used. Query 5 is not satisfied if a value lesser than 24 is used. Decoration clocks y[i] are reset when a process starts the enter part of the protocol and measure the elapsed time of waiting. Obviously, queries 4 and 5 have the same conclusion for any process i. It is guaranteed that the waiting time is bounded and amounts as upper bound to (N-1) critical sections. Queries 6 and 7 check progress properties. In particular, query 6 guarantees that a process which starts the enter part of the protocol, eventually enters the critical section (this is of course also confirmed by bounded waiting time). Similarly, query 7 says that a process which starts the enter part of the protocol always comes back to home (Non_CS location).

Table 1

	Query	Result
1	A[] !deadlock	satisfied
2	E<> Process(0).In_CS+Process(1).In_CS+ Process(2).In_CS+Process(3).In_CS+ Process(4).In_CS>1	not satisfied
3	A[] (forall(i:pid) Process(i).In_CS imply (forall(j : pid) j!=i imply !Process(j).In_CS))	satisfied
4	A[] Process(0).End_Enter imply y[0]>=0	satisfied
5	A[] Process(0).End_Enter imply y[0]<=24	satisfied
6	Process(0).Start_Enter --> Process(0).In_CS	satisfied
7	Process(0).Start_Enter --> Process(0).Non_CS	satisfied

The model in Fig. 2 can easily be adapted to be analyzed using the "high resolution" approach suggested in (Hamber and Vaandrager, 2008). In this case the check channel is useless and the Busy_Wait location can be eliminated.
Variable lock is true when a process wanting to enter is allowed to begin its critical section. Other details should be self-explanatory. This algorithm too ensures a bounded waiting time of at most (N-1) critical sections.

CATALOG OF REUSABLE CONCURRENT MODELS

Mutual exclusion algorithms like those shown in the previous section can be the basis for implementing high level concurrent structures. Normally they are not directly used by the concurrent programmer which prefers instead to use such constructs as monitors, semaphores etc. which can provide both mutual exclusion and synchronization mechanisms. In the following some reusable UPPAAL templates are proposed which model some frequently used concurrent structures which are at the basis of common concurrent design patterns (Grand, 2002).

The description makes some reference to Java concurrency (Goetz et al., 2006) but the solutions can be ported also to other programming languages. For brevity, some constructions like the Hoare's monitor, barrier, exchanger etc., are not reported although they are implemented.

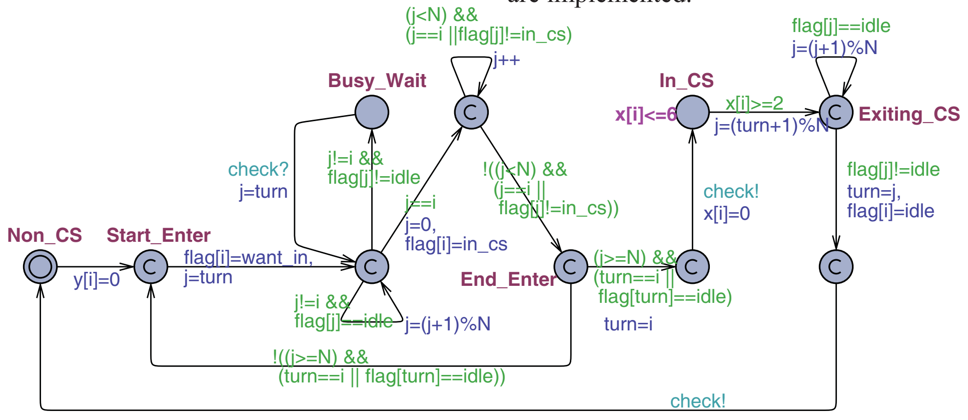

Figure 2. An UPPAAL Process template for Eisenberg and McGuire mutual exclusion algorithm

Java native monitor

The essential semantics of the Java native monitor, which derives from the Lampson and Redell monitor (Lampson and Redell, 1979), is that any object has a lock which provides a waiting room, that waiting processes are not necessarily awaken in FIFO order and that notifying methods make only ready-to-run one or more waiting processes without giving to them any privilege with respect to newly arriving threads. All of this suggests the following structure for a typical entry procedure of a thread-safe class:

```
return_type entry_proc(params) throws InterruptedException{
     synchronized( m ){
          while( condition_for_waiting_is_true )      m.wait();
          update_operation
          m.notify[All]();
          …
     }
}//entry_proc
```

m is the object which provides the lock, i.e. it is the monitor object. **m** can be **this** but often (better) is convenient for it to be a private object (Bloch, 2008) of the guarded class. In the following, a UPPAAL model is proposed which rests on four operations: enter, exit, wait and notifyAll (which is of more general use than notify) which are realized as channels, and a local boolean lock variable which holds the lock status.

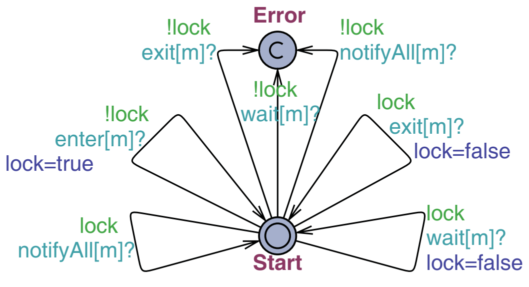

Figure 3. UPPAAL template of a Java native monitor

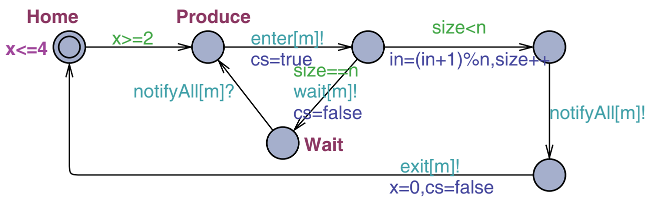

Figure 4. a) Producer model

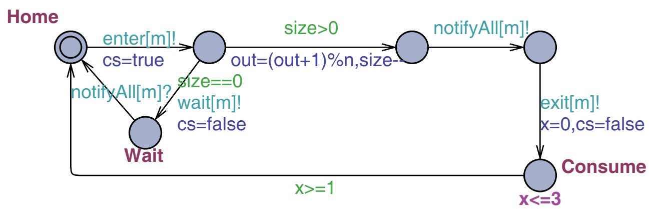

Figure 4. b) Consumer model

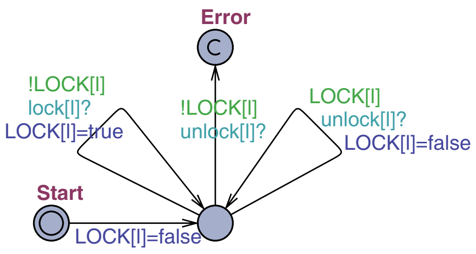

Figure 5. a) Lock template model

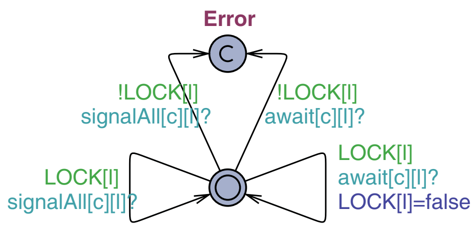

Figure 5. b) Condition template model

A waiting process can only be awaken by a notifyAll operation (the interruption mechanism is ignored). Fig. 3 portrays the template model for a Java monitor.
Global declarations for introducing one or more monitors in an application model are:

```
const int MONITORS=…; //number of monitor objects
typedef int[0,MONITORS-1] mid; //monitor unique identifiers
//monitor operations as array of channels
chan enter[MONITORS];
chan exit[MONITORS];
chan wait[MONITORS];
urgent broadcast chan notifyAll[MONITORS];
```

A monitor model initializes by assigning false to the associated lock variable. Invoking a wait/notifyAll/exit on a monitor whose lock is false is a fatal error (the committed Error location is entered which has no outgoing edge).

A wait[m]? synchronization opens the monitor lock. It is up to the invoking process to enter a waiting location from which it exits on receiving a notifyAll signall. Fig. 4 shows a producer/consumer model with a bounded buffer. Producer and consumer instances receive a unique process id in the relevant category (p_id for producers and c_id for consumers) and the monitor object upon which mutual exclusion and synchronization are based. Models in Fig. 4 clarify that an awaken process has to re-gain the monitor as any newly arriving process. When a process updates the buffer, it awakes all the waiting processes by a notifyAll[m]! which is a broadcast channel. Each producer/consumer instance owns a local boolean cs variable (useful for analysis purposes) for registering if it is or not in the critical section.

The producer/consumer model was checked with a varying number of producers and consumers. The following two queries (which are satisfied) check that at any time at most one process can be in its critical section:

```
A[] (forall(i:p_id) Producer(i,0).cs imply (forall(j:p_id) j!=i
imply !Producer(j,0).cs) && (forall(k:c_id)
!Consumer(k,0).cs))

A[] (forall(i:c_id) Consumer(i,0).cs imply (forall(j:c_id) j!=i
imply !Consumer(j,0).cs) && (forall(k:p_id)
!Producer(k,0).cs))
```

Lock/Condition

Starting from Java 5, the concurrent Java library provides an alternative mechanism to the built-in monitor, which is based on the concept of a lock and associated conditions. The structure is just syntactic sugar built on the monitor (lock) mechanism. Now, though, processes can wait on different rooms (conditions) of the same lock. The lock/unlock operations are defined on a lock object, whereas await/signal[All] are the operations on a condition. Only the signalAll operation is considered (the signal method would awake a process without any order). In the UPPAAL modelling, the association of conditions to lock is achieved by using a bi-dimensional array of channels where the first index selects a condition, the second one the associated lock. Fig. 5 shows the developed Lock (with parameter lock id l) and Condition (with parameters the condition id c and lock id l) UPPAAL (sub) models. The array of LOCK booleans storing the lock statuses is made global so as to be shared by a lock and its conditions. Of course, the programming model is very similar to that shown for the Java native monitor: each use of enter[m]!/exit[m]! is replaced by a use of lock[l]!/unlock[l]!, an use of wait[m]! is replaced by await[c][l]! where c is a condition of l, an use of notifyAll[m]! is replaced by signalAll[c][l]! for awaking all the waiting processes on condition c.

Semaphores

Can be counting or binary semaphores (see e.g. (Silberschatz et al., 2010)). They can be used for mutual exclusion and synchronization purposes (Downey, 2007). In the following, the names of the operations on semaphores are the classic P and V (Dijkstra, 1965). The proposed implementation uses a bounded queue for storing the identifiers of processes waiting on the semaphore. The awaking of waiting processes follows the FIFO order. Each semaphore holds a private counter which cannot go negative, and stores the number of permits available on the semaphore. The following globals help defining the semaphore models:

```
const int PROCESSES=…; //number of processes
typedef int[0,PROCESSES-1] pid; //process ids subtype
const int SEMAPHORES=4; //number of semaphores
typedef int[0,SEMAPHORES-1] sid; //semaphore ids subtype
//semaphore operation-channels
chan P[SEMAPHORES];
chan V[SEMAPHORES];
chan GO[PROCESSES];
pid proc; //process id trying to P(ass through the semaphore
```

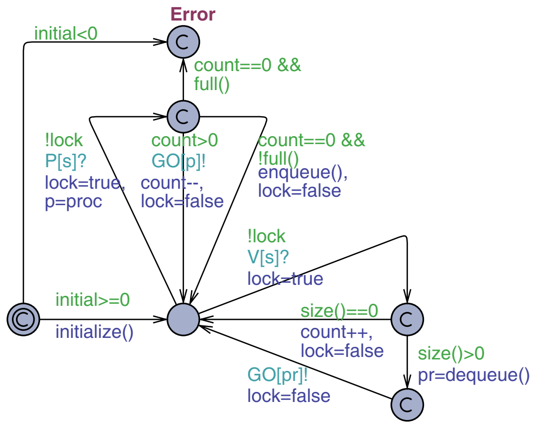

Figure 6. a) Counting semaphore model

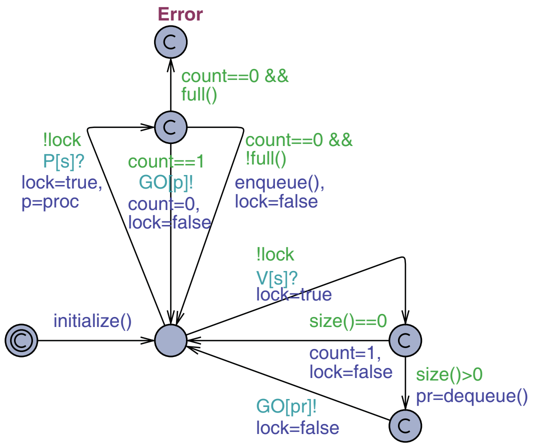

Figure 6. b) Binary semaphore model

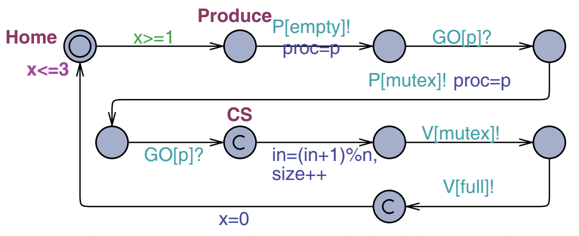

Figure 7. a) Producer model with semaphores

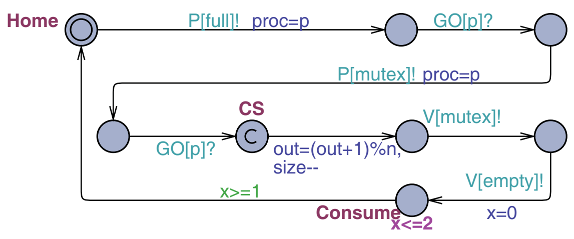

Figure 7. b) Consumer model with semaphores

Specific constant names can also introduced globally to make more readable a process model when accessing selected semaphores. Fig. 6 portrays the UPPAAL templates for the counting and binary semaphores. Parameters of both models are the semaphore id, the initial value (which for a binary semaphore is restricted to be 0 or 1), the expected length of the waiting queue. The design pattern which follows from the models in Fig. 6 is that a process which executes a P[s]! operation on a semaphore s must assign, in the update part of the command, its own process identifier to the global variable proc. In addition, following a P[s]! synchronization, the process has to wait for a GO[p]? synchronization which unblocks the process. Models in Fig. 6 immediately release a GO command if a permit exists at the time of a P[s]!. Note that indexes of the array of GO channels are process ids and not semaphore ids.
Model implementation rests on a few C-like functions which hide the counter initialization and the management of the waiting queue of the semaphore. Mutual exclusion of P/V atomic operations is ensured by a local lock object of the semaphore, initialized to false. To clarify the use of the semaphore models, Fig. 7 shows the producer/consumer models of classical bounded buffer application. The models receive as parameter their process id p (of type pid). Three semaphores are used: mutex (binary semaphore), empty and full (counting semaphores) having a number of permits, from time to time, which reflects respectively the number of empty/full slots in the bounded buffer. The ids of these semaphores are introduced in the global declarations.
The instructions for setting up the UPPAAL system model composed of two producers, one consumer and buffer capacity of n, are as in the following:

```
//template process instances
Mutex=BinarySemaphore(mutex /*sem id*/,
        1/*initial value*/,2/*queue size*/);
Empty=Semaphore(empty,n,2);
Full=Semaphore(full,0,1);
prod1=Producer(0);
prod2=Producer(1);
cons=Consumer(2);
//system configuration
system Mutex,Empty,Full,prod1,prod2,cons;
```

The following query (which is satisfied) checks that mutual exclusion is correctly enforced on the three processes:

A[] cons.CS+prod1.CS+prod2.CS<=1

Another template model (JSemaphore) was developed which was inspired by the Java Semaphore class. It allows atomically to withdraw/deposit more than one permit at a time. The channel-operations are AcquireX[jsid], ReleaseX[jsid], AvailablePermits[jsid] where jsid is the id of the semaphore in this particular category, and X can be absent to express the default of 1 permit, or can be a natural up to a given allowed maximum. The AcquireX[] channels correspond to acquireUninterruptibly(...) methods of the Java Semaphore class. The same conventions on classic semaphores apply here: the global proc variable must be assigned the process id at the time of an acquire, which must be followed by a GO[]? command for unblocking. A process acquiring multiple permits at once will block if the requested number of permits is not available. A release command updates the number of permits of the semaphore and (possibly) awakes the oldest awaiting process, provided its permit request is now fulfilled. A process can check the number of available permits through the operation AvailablePermits[jsid] whose use must update the global proc in the usual way, and be followed by a GO[]? command as for an acquire command. The requesting process will find the output of AvailablePermits[jsid] in a global variable which is specified as the fourth parameter (passed by reference) to the JSemaphore template.

EXAMPLES

The following reports a few examples based on some of the UPPAAL developed concurrent structure models. When transforming a UPPAAL model to Java it is important to reflect that GO? synch? and similar synchronizations required in UPPAAL are implicit in the suspensive character of Java methods (e.g. wait(), s.P() on a semaphore s etc.).

Sharable resource

The problem (Reek, 2004) concerns a sharable resource which can be accessed according to the rules: (a) as long as there are fewer than three processes using the resource, new processes can start using it right away, (b) once there are three processes using the resource, all three must leave before any new processes can begin using it.
A first solution is based on the Java native monitor (or the equivalent lock/condition structure). Fig. 8 depicts a template model for the generic Process accessing the resource. Process has two parameters: its process id p and the monitor m.
The variable release is true if currently there is a release of processes from the resource according to rule (b). Variable active stores the number of processes which are currently using the resource. Both must be acted under mutual exclusion. A monitor m is used as a guardian of the resource. As long as the number of active processes is 3 or there is a release in progress, the asking process is forced to wait (it reaches the Wait location and frees the monitor). On exiting from the critical section, if active is equal to 3, release is set to true. In any case the exiting process decrements the active counter. When active goes to 0, a notifyAll[m]! is issued and release is reset to false. Note that if active is 0 but no release was in progress, the notifyAll[m]! signal reduces to a no-operation because no processes are really waiting.

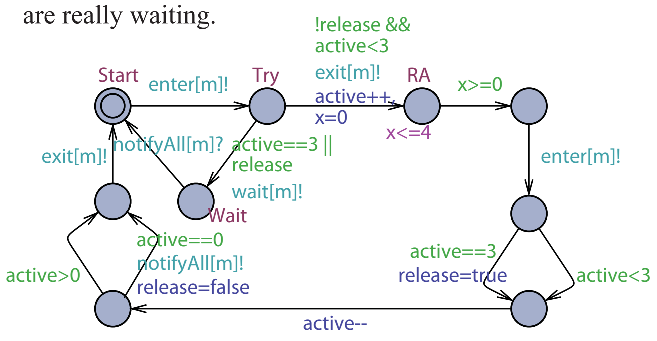

Figure 8. Process model based on the Java native monitor

A system model with 5 instances of Process was verified. It was found to be free of deadlocks but without liveness guarantee for any process. Liveness was checked e.g. with a query like this: Process(0,0).Start-->Process(0,0).RA (Resource Access) which is not satisfied. The query continues to be not satisfied even when the Try location is turned to be committed and the enter/exit/wait channels are declared urgent.

```
public class Manager {
    private int active=0;
    private boolean release=false;
    private Object m=new Object();
    public void want_to_enter() throws InterruptedException{
            synchronized( m ){
                while( active==3 || release ) m.wait();
                active++;
            }
    }//want_to_enter
    public void exit() throws InterruptedException{
            synchronized( m ){
                if( active==3 ) release=true;
                active--;
                if( active==0 ){
                    m.notifyAll();
                    release=false;
                }
            }
    }//exit
}//Manager
```

Figure 9. A Java thread-safe class corresponding to model in Fig. 8

The problem is that process selection at entering and process awaking from waiting are not deterministic. A Java thread-safe class corresponding to the model in Fig. 8 is portrayed in Fig. 9.

Fig. 10 shows a solution based on semaphores, which mimics a solution based on the Hoare's monitor. Two binary semaphores MUTEX and WAIT are used. WAIT serves to block a process when active is 3 or there is a release in progress. The solution exploits the "Pass the Baton" design pattern (Reek, 2004), i.e. when an exiting process finds the conclusion of a release and that there are waiting processes, it awakes (the oldest) one and passes to it the mutual exclusion. On the other hand, when release is true or there is no waiting process, the exiting process frees the MUTEX.

The application model was model checked and found free of deadlocks too. Liveness was checked by the queries:

Process(0).W --> Process(0).RA
Process(0).Start --> Process(0).RA

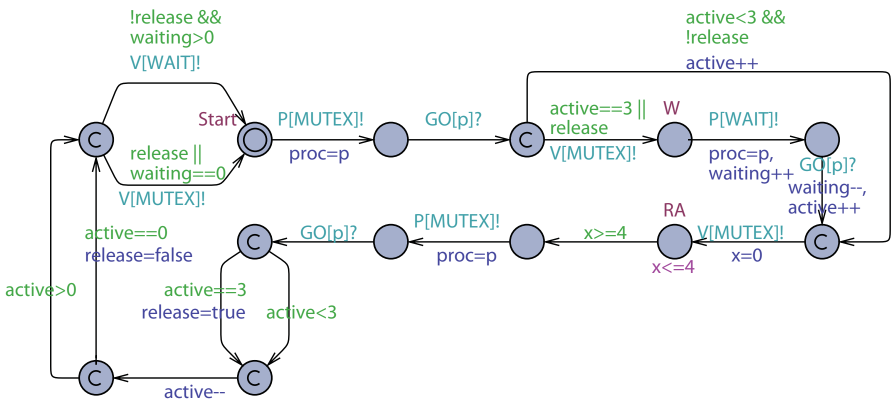

Figure 10. Process model based on semaphores

The above queries verify respectively if process 0 (or any other process) always is able to reach RA from W (start waiting), or from Start.
These queries are not satisfied because of the non urgent character of P/V and GO channels, together with the fact that an UPPAAL timed automaton is not forced to abandon as soon as possible a normal location. Changing the channels to urgent, both the queries are satisfied, mirroring that a process is eventually chosen from the waiting queue of semaphores (FIFO behavior).

Termination problems

The first problem considered is proposed in (Stallings, 2005) and involves five processes: three of type A and two of type B. The goal is finding the minimum number of semaphores and using exclusively P's and V's on these semaphores so as to have always that the five processes terminate according to the sequence ABAAB. Instead of trying intuitively to find a solution, the following suggests a Petri net (see Fig. 11) which models in abstract terms a solution. Transition tA models a process A termination. Transition tB models a process B termination. Net topology and initial marking mirror the number of A and B processes (see places A and B) and the constraints on the termination sequence (see places cA and cB and weights of cB-tB and tB-cA arcs).
Obviously, there is no general rule to guide the transformation from a specification to an implementation which is guaranteed to be correct with respect to the specification. In this case, though, by interpreting places as semaphores and their initial marking as the initial value of the semaphores, and interpreting token withdraw and token deposit during transition firing respectively as P's and V's on the relevant semaphores, one can achieve a semaphore implementation from the net model. An important aspect to reproduce in the semaphore implementation is the atomicity of transition firing.

In reality, semaphores corresponding to places A and B can be omitted because in the implementation the number of processes A and B is implicitly represented by instances of their class/template. As a consequence, five semaphores could be used: cA, cB, mA, mB, mAB where mA and mB are mutex semaphores guarding A processes each other and B processes each other, whereas mAB regulates mutex among As and Bs. As a first attempt, Fig. 12 sketches semaphore declaration and initialization, and the body of A and B process types:

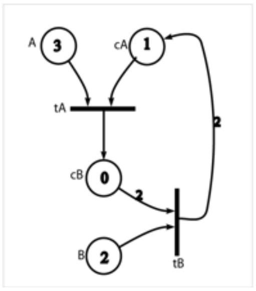

Figure 11. Petri net model for the termination problem ABAAB

The implementation in Fig. 12 is redundant: mA and mB can be eliminated by resting on cA initialization which excludes multiple A to initiate firing of transition tA in Fig. 11, and cB initialization along with the FIFO property of employed semaphores, so as to allow only one B to fire transition tB. Proposed implementation using three semaphores is shown in Fig. 13.

```
Semaphore cA<-1, cB<-0
BinarySemaphore mA<-1, mB<-1, mAB<-1
A:
P(mA)
P(cA)
P(mAB)
V(cB)
V(mAB)
V(mA)
```

Figure 12. a) Global declarations and A process body sketch

```
B:
P(mB)
P(cB)
P(cB)
P(mAB)
V(cA)
V(cA)
V(mAB)
V(mB)
```

Figure 12. b) B process body sketch

Figures 14 and 15 depict UPPAAL models for processes A and B.

The semaphores cA and cB have ids respectively CA and CB. Figures 16 and 17 show a decoration automaton Checker along with the Synch automaton and urgent synch channel which were designed (with the help of counters tA and tB which count respectively the number of terminated A processes and B processes) to demonstrate correctness of the simplified solution.

```
Semaphore cA<-1, cB<-0
BinarySemaphore mAB<-1
A:
P(cA)
P(mAB)
V(cB)
V(mAB)
```

Figure 13. a) Minimal A process body sketch

```
B:
P(cB)
P(cB)
P(mAB)
V(cA)
V(cA)
V(mAB)
```

Figure 13. b) Minimal B process body sketch

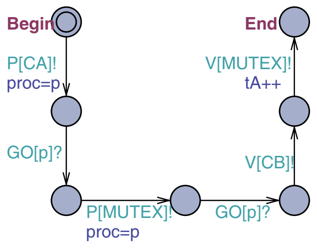

Figure 14. Automaton of A process

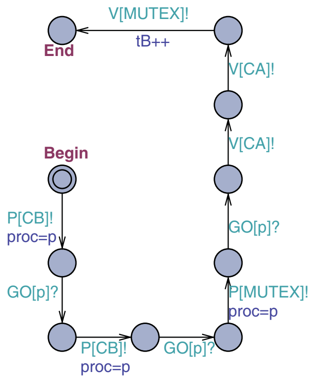

Figure 15. Automaton of B process

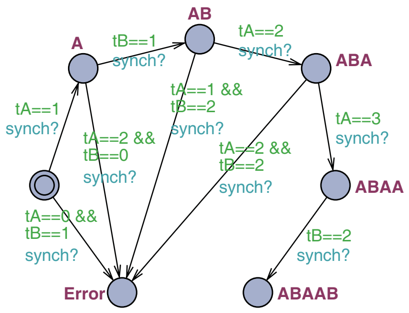

Figure 16. Checker automaton

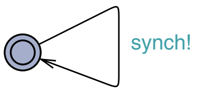

Figure 17. Synch automaton

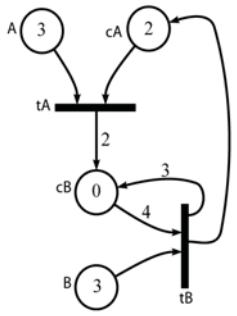

Figure 18. Petri net model for the AABABB termination problem

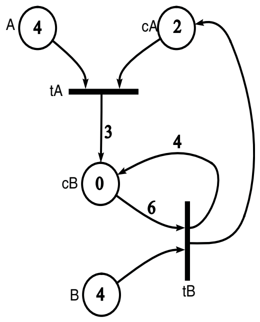

Figure 19. A Petri net model for the problem AABABABB

Correctness of the semaphore implementation of the ABAAB termination problem was verified by the query:

```
A[] !Checker.Error
```

which was find satisfied. The same method was applied to the termination problem AABABB with three A and three B. In this case was designed the Petri net model of Fig. 18.

A correct and minimal semaphore implementation based on four classic semaphores is sketched in Fig. 20. The more resource demanding termination problem of 8 processes AABABABB proposed in (Stallings, 2005), was solved according to the Petri net in Fig. 19 and the JSemaphore automaton. Similarly to the AABABB problem, four classical semaphores could be used but the corresponding UPPAAL model is hard to verify.

In Fig. 21 it is sketched a verified correct solution based on three JSemaphore automata, of which one serves as a mutex. Since a process B acquires 6 tokens at once or none and blocks, the mutex mB semaphore of Fig. 20 is no longer required. The resulting reduced model was found more amenable for the model checker.

```
Semaphore cA<-2, cB<-0
BinarySemaphore mAB<-1, mB<-1

A:                    B:
P(cA)                 P(mB)
P(mAB)                P(cB)
V(cB)                 P(cB)
V(cB)                 P(cB)
V(mAB)                P(cB)
                      P(mAB)
                      V(cA)
                      V(cB)
                      V(cB)
                      V(cB)
                      V(mB)
                      V(mAB)
```

Figure 20. a) A process body sketch

Figure 20. b) B process body sketch

```
JSemaphore cA<-2, cB<-0, mAB<-1

A:                    B:
Acquire(cA)           Acquire(cB,6)
Acquire(mAB)          Acquire(mAB)
Release(cB,3)         Release(cA)
Release(mAB)          Release(cB,4)
                      Release(mB)
```

Figure 21. a) A process body sketch

Figure 21. b) B process body sketch

All the experiments were carried out on a Win7 64 bit, Intel i5 Core 750 @ 2.67GHz, with 6GB RAM.

CONCLUSIONS

This paper proposes an approach based on UPPAAL for modelling and exhaustive verification of concurrent programs, e.g. destined to be implemented in Java. Some common patterns mainly inspired by the Java concurrent package were abstracted as reusable template processes of UPPAAL which can be easily integrated and composed in user-defined project models. The reasoning and visibility capabilities enabled by the UPPAAL toolbox are of paramount importance in the didactic (but also in other contexts) of concurrent programming which is a well-know difficult task to master. Nevertheless concurrency is emerging as a crucial factor for future complex application developments which can greatly benefit from the computing potentials offered by modern multi-core processor architectures. A lesson learned from the described experience is that rigorous modelling of concurrent structures not only help proving correctness of a solution but the efforts behind modelling and analysis highlight semantics of a concurrent pattern and can guide the implementation in an object-oriented language like Java.

Prosecution of the research aims to

- improving and extending the library of reusable concurrent models, e.g. with control mechanisms such as the Active Oberon (Active Oberon, on-line) monitor which has boolean conditions and an implicit signalling mechanism
- building a reference collection of solution models for significant classes of concurrent applications
- experimenting with other model checkers such as SMV, PVS, TLA+ etc.

REFERENCES

Active Oberon, http://bluebottle.ethz.ch/languagereport/index.html

Alur R. and D.L. Dill (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2), pp. 183-235.

Behrmann G., David A., Larsen K.G. (2004). A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, M. Bernardo and F. Corradini Eds., LNCS 3185, Springer, pp. 200-236.

Bengtsson J., Yi W. (2003).Timed automata: semantics, algorithms and tools. J. Desel, W. Reisig, and G. Rozenberg (Eds.): *ACPN 2003*, *LNCS 3098*, pp. 87–124.

Bloch J. (2008). *Effective Java*. 2nd Edition, Addison-Wesley.

Clarke E.M., Grumberg O., Peled D.A. (2000). *Model checking*. MIT Press.

Cicirelli F., Furfaro A. & Nigro L. (2007). Using TPN/Designer and UPPAAL for modular modelling and analysis of time-critical systems. *Int. J. of Simulation Systems, Science & Technology*, 8(4):8-20, http://ducati.doc.ntu.ac.uk/uksim/journal/Vol-8/No-4/cover.htm.

Cicirelli F., A. Furfaro A., Nigro L. (2009). Modelling and Analysing Real Time System Specifications using Time Stream Petri Nets. In *Proc. of 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software (WRTP/RTS'09)*, Mragowo, Poland, October 12-14, pp. 35-42.

Dijkstra E.W. (1965). Cooperating sequential processes. Technological University, Eindhoven, The Netherlands, http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html

Downey A. (2007). *The Little Book of Semaphores*. Green Tea Press, 2.1.2 Edition.

Furfaro A., Nigro L. (2007). Modelling and schedulability analysis of real-time sequence patterns using Time Petri Nets and UPPAAL. In *Proc. of Int. Workshop on Real Time Software (RTS'07)*, Wisla, Poland, pp. 821-835.

Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. (2006). *Java concurrency in practice*. Addison Wesley Professional.

Grand M. (2002). *Patterns in Java*, Vol. 1, Wiley.

Hamberg R., Vaandrager F. (2008). Using model checkers in an Introductory Course on Operating Systems, *ACM SIGOPS Operating Systems Review*, Vol. 42, Issue 6.

Lampson B.W., Redell D.D. (1979). Experience with processes and monitor in Mesa. *In Proc. of SOSP*, pp. 43-44.

Reek K.A. (2004). Design patterns for semaphores. In Proc. of SIGCSE’04 Technical Symposium on Computer Science Education, Vol. 36, Issue 1.

Stallings W. (2005). *Operating Systems: Internals and Design Principles*. Prentice-Hall.

Silberschatz A., Galvin P.B., Gagne G. (2010). *Operating System Concepts*. 8th Edition, Wiley.