Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1361–1368

Modelling Java Concurrency: An Approach and a UPPAAL Library

Franco Cicirelli, Angelo Furfaro, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Università della Calabria, DIMES
I-87036 Rende (CS) - Italy
Email: f.cicirelli@dimes.unical.it, a.furfaro@dimes.unical.it, l.nigro@unical.it, f.pupo@unical.it

***Abstract*—To effectively cope with correctness issues of concurrent and timed systems, the use of formal tools is mandatory. This paper proposes an original approach to modeling and exhaustive verification of Java-based concurrent systems which relies on the popular UPPAAL model checker. More precisely, a library of UPPAAL timed automata (TA) reproducing the semantics of major Java concurrent and synchronization mechanisms was developed, which fosters a smooth transition from specification down to implementation. The library includes such common control structures like semaphores and monitors, both classic and Java specific. The paper describes the developed TA library and shows its practical use by means of examples. Finally, an indication of on-going and future work directions is drawn in the conclusion.**

I. INTRODUCTION

CURRENT and prospective availability of powerful multi-core (in the CPU) and many-core (in the Graphical Processing Unit or GPU) computing architectures, and the growing acceptance of Java as a key technology for building time-dependent embedded systems, challenges software developers to the construction of concurrent programs which can greatly benefit from the high-performance computing potential of such parallel machines. Concurrent algorithm design, though, is a well-known difficult task due to human inability to check, either through peer-review or by experimental tests, the correctness of a parallel program where multiple threads of control evolve simultaneously according to complex interleaving of their actions. Race conditions, deadlocks, starvations and so forth are common risks deriving from an improper use of locks.

The work described in this paper argues that to properly design and implement concurrent and time-dependent software systems, the use of formal tools is mandatory which can enable a *reasoning* on concurrency, which is of utmost importance both in an educational or industrial context. This paper describes current status of a research project on modeling and verification (M&V) of concurrent and timed systems which was preliminarily proposed in [1]. The approach is centered on Java as the target implementation language and UPPAAL [2], [3] as a popular, mature and efficient timed automata (TA) [4] based toolbox, which makes it possible to model check complex systems [5], [6]. Although the developed concurrent structures are Java-based, they can easily be ported to other concurrent languages as well. More precisely, this paper describes current shape of a UPPAAL catalog of concurrent control structures, which was significantly improved and expanded with respect to the initial version reported in [1].

This paper contribution can be related to the solutions proposed e.g. by Hamberg & Vaandrager in [7] and to the well-known approach FSP/LTSA [8]. Our work shares with [7] the use of the UPPAAL model checker and some common semaphore and monitor control structures. However, the catalog described in this paper is original, more general and efficient, and fosters different concurrent programming styles. In addition, proposed mechanisms were mainly inspired by Java concurrency features. The FSP/LTSA approach is based on a process algebra specification of a concurrent system (FSP or Finite State Processes), automatically transformed into an equivalent Labelled Transition System (LTS) expression which is model checked in the toolbox LTSA (LTS Analyzer). A system specification must finally be implemented into Java. However, the FSP specification language does not favor the expression of FIFO based concurrent control structures (e.g. of a semaphore). Moreover, FSP/LTSA adopts a discrete time model which can complicate the verification of realistic models. A semantic gap exists between an FSP specification and a corresponding implementation in Java of a system model. Obviously, in general, an implementation cannot be proved to be a faithful concretization of a specification, but a reduction in the above semantic gap, as proposed in this work, can *help* achieving a correct implementation.

The paper is structured as follows. First basic concepts of UPPAAL are summarized. Then a running modeling example is introduced. The paper goes on by describing the developed TA catalog for modeling concurrent Java programs. Then the library is practiced through the chosen example. The discussion puts into evidence a general approach for modeling a Java thread-safe class. Finally, an indication of on-going and future work is given in the conclusion.

II. AN OVERVIEW TO UPPAAL

A system [2] is the parallel composition of multiple timed automata modeled as *template processes*, which can have parameters, can be instantiated, and consist of *atomic actions*. Parallel composition means that UPPAAL is capable of analyzing all the possible action interleavings of the component processes.

978-1-4673-4471-5/$25.00 © 2013, IEEE

TA synchronize to one another by CSP-like channels (*rendezvous*) which carry no data values. Asynchronous communication is provided by broadcast channels where a single sender can engage in a synchronization with a (possibly empty) group of receivers. The sender of a broadcast signal in no case is blocked. Locations (states) of an automaton are linked by a set of *edges* (transitions). Time is handled by means of *clock* variables. Clocks can only be reset and compared against to a nonnegative integer constant. All the clocks of a model increase automatically at the same rate of advancement of the (hidden and dense) system time. UPPAAL extends basic TA with integer (and boolean) variables and arrays of integers, clocks and channels. Declarations can be global (shared by all the TA in a model) or local to a TA. In latest versions of the toolbox, C-like functions and structures are permitted.

Edges can be annotated by three (optional) components: (i) a *guard*, (ii) a *synchronization* action (? for input and ! for output) on a channel, and (iii) an *update* consisting of a set of clock resets and variable assignments. The update of an output command is executed before that of the matching input command.

A clock *invariant* can be attached to a location as a *progress* condition. The timed automaton can remain into the location as long as its invariant gets not violated. UPPAAL offers also *committed* and *urgent* locations which must be exited immediately (without passage of time), and *urgent channels* whose synchronizations must be fired as soon as possible. Committed locations have priority with respect to urgent locations.

UPPAAL consists of a graphical editor, a simulator and a verifier (model checker). The simulator executes a specification and visually documents the reached execution state by traversing the model state graph. The simulator is useful for model debugging and for examining a diagnostic trace (counter example) built by the verifier. For exhaustive property assessment, the verifier must be used which tries to build the reachability graph of the model, where execution states are organized into equivalence classes based on *time zones* (clock inequalities system).

Safety (e.g., absence of deadlocks) and bounded liveness (e.g. an end-to-end time constraint) properties can be verified by reachability analysis using a subset of TCTL formulas [2]. Admitted formulas (see below) refer to local state properties, i.e. boolean expressions over predicates on locations and integer variables and clock constraints.

$E <> \varphi$ means "*Possibly* φ" (a state can be reached in which φ holds).

$A[]\ \varphi$ means "*Invariantly* φ" (in all states φ holds).

$E[]\ \varphi$ means "*Potentially Always* φ" (a path exists where φ holds in all reached states).

$A <> \varphi$ means "*Always Eventually* φ" (equivalent to: $not\ E[]\ not\ \varphi$).

$\varphi - - > \psi$ means "φ *always leads to* ψ" (equivalent to: $A[]\ (\varphi\ imply\ A <> \psi)$.

III. A MODELING EXAMPLE

A classic yet representative concurrent example which can be modeled and verified using UPPAAL is the Dining-Philosophers problem (see e.g. [9], [10], [11]). N philosophers (e.g. $N = 5$), seat around a table which has a never ending big plate of spaghetti. Philosophers are equipped by a own plate and a single fork (at its left). Philosophers spend their life by thinking and, when they become hungry, try to get the two forks at its left and its right so as to take some spaghetti and then switching to eating. A thinking phase consumes from 2 to 10 time units. An eating requires from 4 to 12 time units. Forks are kept by the philosopher for the whole duration of its eating. When the philosopher finishes eating, it puts forks (hopefully after cleaning them) on the table and turns to thinking again. The availability of forks can now make some adjacent colleague get them and pass to eating as well. The problem is to ensure that the system is live (no deadlock occurs) and that it is bounded the waiting time a hungry philosopher experiments before achieving the forks (absence of starvation).

Fig. 1 shows a "native" model for philosopher i, which directly depends on the basic UPPAAL features. A global array of boolean `fork`, initialized to all true, holds the status of the fork resources. Forks relevant to the i-th philosopher have indexes i (left) e $(i + 1)\%N$ (right). Adjacent philosophers have identifiers respectively $(i + 1)\%N$ (left colleague) and $(i + (N - 1))\%N$ (right colleague).

The model is safe: a philosopher either picks both forks or none, but it is incorrect from the point of view of starvation. A hungry philosopher waits for forks in the `WAITING` location. When the fork status changes, a signal over the broadcast channel `check` is sent which allows all interested philosophers to review their status and possibly switch to the `EATING` location. On a system with N instances of the basic TA, the following queries can be issued:

```
1. A[] !deadlock                        (satisfied)
2. A[] forall(i:pid) Philosopher(i).EATING imply
   !Philosopher((i+1)%N).EATING &&
   !Philosopher((i+(N-1))%N).EATING   (satisfied)
3. Philosopher(0).THINKING-->Philosopher(0).EATING
                                      (not satisfied)
```

Query 2. confirms only not adjacent philosophers can be eating simultaneously. Native UPPAAL models tend to be concise and efficient (in space and time) for model checking. However, a native model has to be intuitively implemented e.g. in Java, relying on the reasoning on the problem solution allowed by model analysis. Ultimately, the action vocabulary of the source model (atomic actions, broadcast signals etc.) has to be transformed in the vocabulary of the target language (synchronized blocks and wait/notifyAll operations, or semaphores etc.). All of this can create problems in achieving a correct implementation. To shorten the semantic gap between modeling and implementation, source model design can be driven by implementation aspects. The following describes a library of UPPAAL TA which furnishes reusable templates for common concurrent control structures.

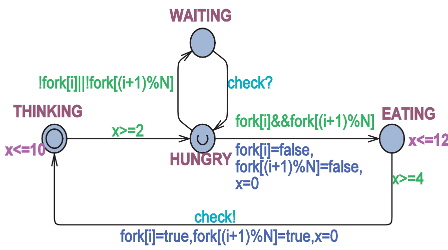

Fig. 1. A UPPAAL native model for the problem of Dining-Philosophers

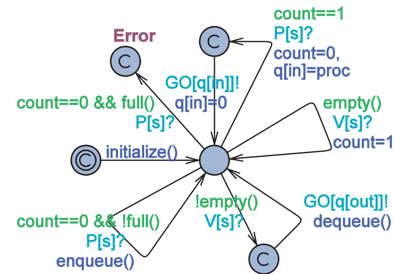

Fig. 2. BINARYSEMAPHORE automaton

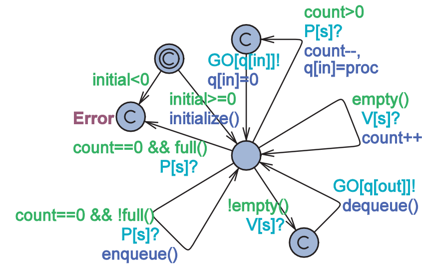

Fig. 3. SEMAPHORE automaton

IV. A TA CATALOG FOR CONCURRENT SYSTEMS

An original library of UPPAAL TA was developed which includes classic binary/counting semaphores, Java inspired semaphore, built-in monitor structure of Java, lock/condition monitor of Java, Hoare monitor, Active Oberon inspired monitor [12], exchangers, barriers etc. Other synchronizers can be added.

A. Semaphore structures

Fig. 2 and 3 respectively show a binary semaphore and a counting semaphore automata whose design tries to balance ease of use with efficient analysis.

Semaphore processes in Fig. 2 and 3 are strong in that they ensure FIFO management of waiting processes. Template parameters include the unique id of the semaphore, the initial number of permits and the expected queue size. A violation of the queue size determines the `Error` location is entered and the verification is deadlocked. Classic `P/V` operations are implemented as channel arrays `P[.]/V[.]` whose dimension mirrors the number of semaphores used in the model. A `P` operation to a semaphore `s`, is requested by a synchronization `P[s]!`. The requesting process is assumed to follow the pattern (see also Fig. 10) of putting 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. Being a meta variable, `proc` does not contribute to the state part of the model. A further channel array `GO[.]`, whose dimension is the number of processes in the model, is used for blocking the requesting process until the semaphore assigns a permit to it. As a consequence, a `P[s]!` operation issued by process `p` should always be followed by `GO[p]?` synchronization. It is worth noting that 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 the `P` operation 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 proposal in [7], our semaphores use less variables. For instance, the identity of the requesting process during a `P` operation which finds green a binary semaphore, is temporarily stored in the surely empty internal queue of the semaphore. The modeler often experiments that even by dropping one single redundant variable can avoid state explosion during the construction of the state graph, thus facilitating model checking.

Fig. 4 portrays the `JSemaphore` automaton which was inspired by the behavior of `java.util.concurrent Semaphore` class. 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. The use of `JSemaphore` rests on the channel arrays `Acquire[.]`, `Release[.]`, `PermitsAvailable[.]`, `GO[.]`, and the use of 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 following a `PermitsAvailable[s]!` operation. A `GO[p]?` synchronization must follow an `Acquire[s]!` or a `PermitsAvailable[s]!` command. More precisely, it is at the time of `GO[p]?` unblocking that `perm` is filled of the semaphore permits number.

It should be noted that both classic and Java specific semaphore TA are useful in practical concurrency modeling. 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.

B. Monitor structures

Although widely used, semaphores are often viewed as a low level concurrent abstraction mechanism, where a misuse

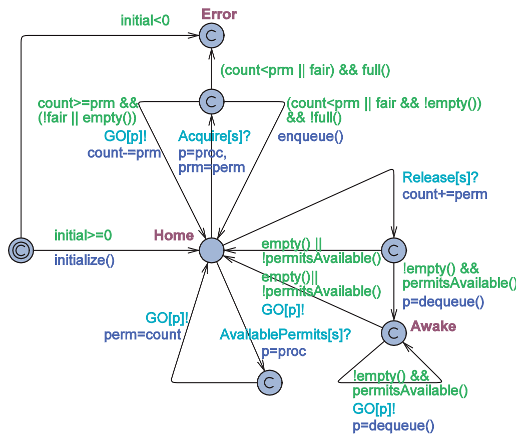

Fig. 4. JSEMAPHORE automaton

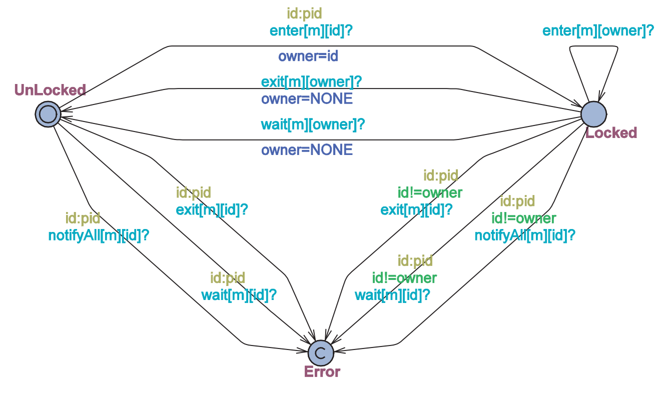

Fig. 5. JMONITOR automaton

of `P/V` operations can easily lead to a deadlock. Monitors, 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 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 which are assigned to both processes and the control structure.

Java adopts the Lampson&Redell [13] 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 the condition requires coming back to waiting or instead the process can go on because the condition is satisfied. Broadcast signaling is not blocking for the signaler process. An awaken process has to compete in reacquiring the lock for it to actually resume execution.

The Hoare monitor (e.g. [9], page 234) has 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 then proceed. The signaler, on the other hand, is put to wait on 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] at page 240 where it is argued, besides any runtime implication and number of context switches, that Lampson&Redell monitor can be superior in the most general case.

An example of a monitor which facilitates the developer by transferring responsibilities from the programming level to the control structure, was adopted in the Active Oberon language [12]. Here the programmer has only to deal with the logic of conditions which, as long as they do not hold, prescribe a process has to wait. Signaling and process awaking is hidden in the control structure.

In the following, a series of developed monitor TA is presented.

Fig. 5 depicts the `JMonitor` automaton which allows to model concurrent objects according to the Java built-in monitor. 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 for 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 identifier `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 owns a lock which can be used as a monitor. The lock holds one implicit condition, whose meaning is only known to the modeler/programmer. The lock object is associated with a *wait-set* where both entering processes which find the lock closed, or processes within a synchronized block (based on the lock object) but whose condition prescribes waiting, are put (although the two kind of waiting processes are clearly distinguished to one another) and suspended. Processes which are suspended for a `wait` operation can only be awaken by a `notifyAll` operation which does not free the lock. Other processes awake as the lock/monitor is up to be abandoned (at an `exit` or `wait` operation). In the proposed implementation, the wait-set is purposely realized implicitly. Processes requesting `enter` simply are blocked if the monitor is already locked.

Processes which execute `wait` are supposed to move into a location (see `WAITING` in Fig. 11) from which they can only exit following a relevant `notifyAll` signal. Towards

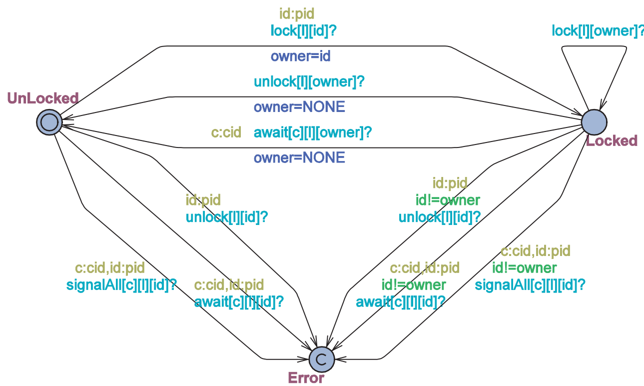

Fig. 6. LOCK automaton

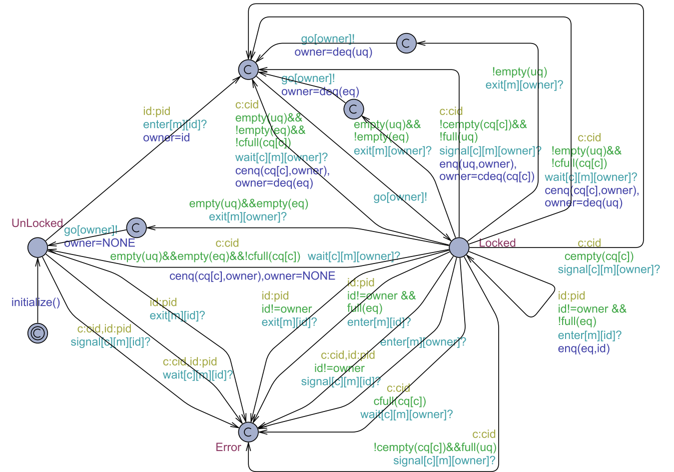

Fig. 7. HOAREMONITOR template

this, channels `notifyAll[.][.]` are declared as broadcast channels.

The automaton in Fig. 5 maintains the identity of the monitor `owner`, which is used both to realize reentrancy and to check for erroneous operations, which in Java correspond to raising an `IllegalMonitorStateException`.

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 pattern also requires that an awaken process from a wait location has to explicitly compete in reacquiring the lock (this operation is hidden in the Java `wait()` method of class `Object`). The design pattern makes it possible to implement also 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*), thus competing for the lock before checking the condition.

In reality the Java built-in monitor also offers a `notify` operation to awake *one* unspecific process suspended in the wait-set. For generality reasons the automaton in Fig. 5 only implements the `notifyAll` (broadcast) operation because, as discussed e.g. in [14] at pages 181-183, the use of `notify` can cause what is known as the `Lost-Wakeup-Problem`.

Since Java 5, the `java.util.concurrent` package also provides a refinement of the built-in monitor through the lock/condition control structure. In this version, it is possible to introduce both a lock object and a certain number of condition objects linked to the lock object. As a consequence, processes can be suspended on the different conditions and the signaling mechanism can be directed to all the processes waiting on a certain condition.

Fig. 6 depicts the Lock automaton which realizes the locking mechanism (and its reentrancy) and also handles the relevant conditions.

Monitor operations are captured by two dimensional `lock[lid][pid]`, `unlock[lid][pid]` array channels, whose first dimension is related to the sub-range of lock unique identifiers used in a model, and whose second dimension is tied to the process unique identifiers, and three dimensional `await[cid][lid][pid]`, `signalAll[cid][lid][pid]` array channels where the first dimension consists of the unique condition identifiers of a given lock identifier.

As in the case of the `JMonitor` automaton, the enter wait-set and condition wait-set are realized implicitly, as well as `signalAll[cid][lid][pid]` channels are declared as broadcast channels.

The same conventions discussed for `JMonitor` apply here: a waiting process on a condition `c` is supposed to wait into a suitable location of the automaton, from where the process can exit following a `signalAll` or a timeout. It then has to compete for reacquiring the lock and is in charge of rechecking the relevant condition.

As it is common, the Hoare monitor can be achieved on top of semaphores, in particular binary semaphores. If N are the conditions of the monitor, $N+2$ semaphores are to be used: one as mutex, another for the urgent mechanism of signalers and N for conditions. In Fig. 7, a slightly different but equivalent and more efficient automaton implementation is proposed which rests on $N+2$ queues. The monitor can be used through the matrices of channels `enter[mid][pid]`, `exit[mid][pid]`, `wait[cid][mid][pid]`, `signal[cid][mid][pid]`, `go[pid]` where `mid`, `cid` and `pid` are respectively the integer sub-ranges of monitor unique identifiers, relevant condition unique identifiers, unique process identifiers. The pattern of use does not necessarily depend on the while-loop required by built-in Java monitor. A process waiting on a condition is, in general, guaranteed that the condition holds when it is signaled. The monitor is assumed to be not reentrant. As a rule, a synchronization on the `go[.]?` channel must follow each invocation (!) of `enter`, `exit`, `wait` or `signal` operation.

A simplification of the Hoare monitor is provided by the Active Oberon monitor (see Fig. 8) where the signaling operation is removed. The burden (and the risks) of proceeding by an awaken process whose condition cannot possibly hold,

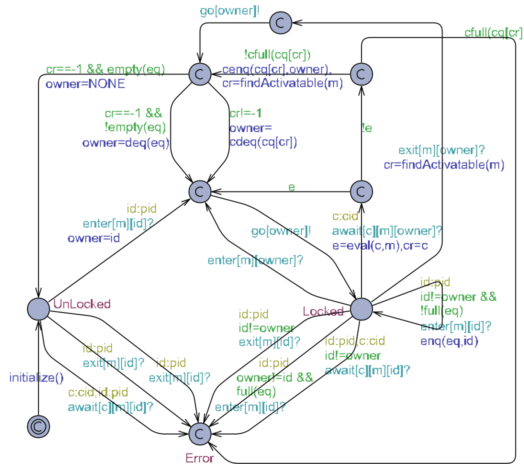

Fig. 8. AOMONITOR template

are eliminated by the control structure which manages an implicit signaling and the transfer of the lock to an awaken process. Therefore, the resultant modeling/programming style becomes more concise with respect to the Hoare monitor.

The `AOMonitor` automaton (Fig. 8) is supposed to be reentrant. Its use depends on the matrices of channels `enter[mid][pid]`, `exit[mid][pid]`, `await[cid][mid][pid]`, `go[pid]`. A `go[.]?` synchronization is required after each invocation (!) of `enter`, `exit` or `await` operation. Any waiting room is realized as a FIFO queue.

The modeler has to introduce a global `bool eval(cid,mid)` function, model specific, which receives a condition id and its monitor id and returns true if the logical boolean expression which is associated with the given condition of the given monitor, holds; otherwise `eval()` returns false.

The `findActivatable(mid)` function used in Fig. 8 scans the list of conditions of the given monitor and returns, if there is one, the identifier of the first condition which is found satisfied; the function returns -1 if the search fails. The result of `findActivatable` is used to transfer the control (together with the monitor lock) to the oldest process waiting on the given condition.

In order to avoid starvation, the control structure maintains the last index of success on the list of conditions so as to start the next lookup from the next position and cyclically.

V. PUTTING THE LIBRARY INTO ACTION

Usefulness of the developed library was assessed through several examples. In the following, the use of the library is demonstrated by applying it to the Dining-Philosophers problem. Modularity issues suggest separating the application processes from the details of concurrency control which in Java are embedded into a thread-safe class. Therefore it is convenient to organize the Philosopher model as shown in Fig. 9 and introducing a `Manager` model which exposes

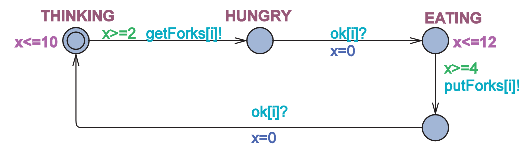

Fig. 9. The PHILOSOPHER automaton

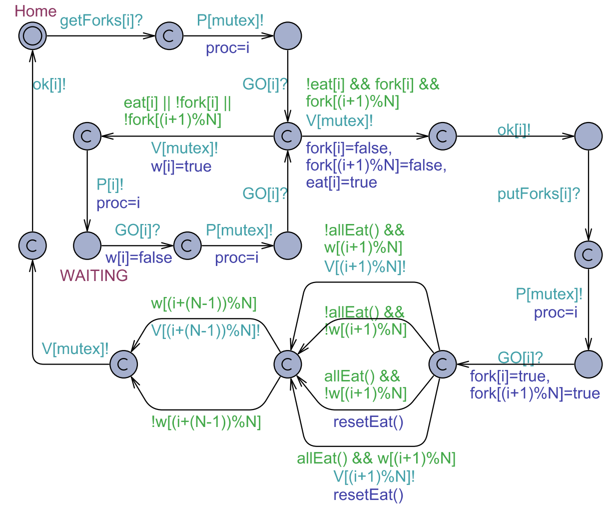

Fig. 10. Manager automaton based on semaphores

a suitable interface to philosopher processes and hides the synchronization constraints. In particular, the channel arrays `getForks[pid]`, `putForks[pid]` e `ok[pid]` are assumed to define the `Manager` interface. Since each philosopher can block in the manager model, N identical instances of the `Manager` are created, each one corresponding to a distinct philosopher. All the manager instances, though, share global data e.g. the boolean array `fork[.]` about free/occupied status of forks. Both `Philosopher` and `Manager` have one single parameter i (of type `pid`) which furnishes the identity of the philosopher. Following a `getForks[i]!` or `putForks[i]!` operation, the philosopher expects an `ok[i]?` synchronization confirming that the requested operation was carried out. Again, in a Java implementation the `ok` signal is redundant because the extended rendezvous is automatically provided by `getForks/putForks` methods of the `Manager` class.

A. Manager based on semaphores

In Fig. 10 is depicted a `Manager` automaton which uses $N + 1$ binary semaphores.

One semaphore is used for mutual exclusion (`mutex`, initialized to 1). The remaining N semaphores, one per philosopher, are waiting rooms or conditions (always kept to 0). Condition identifiers coincide with philosopher identifiers. The model in Fig. 10 uses two further boolean arrays: `w[pid]` and `eat[pid]`. The former serves to know if a

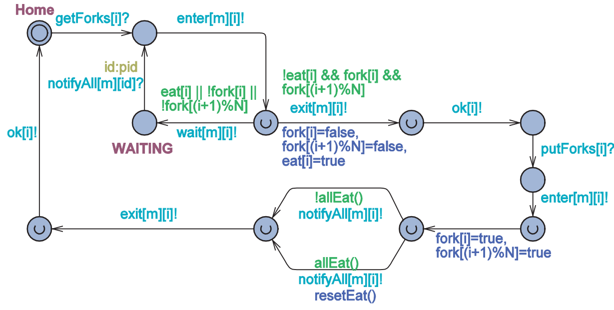

Fig. 11. Manager automaton based on Java monitor

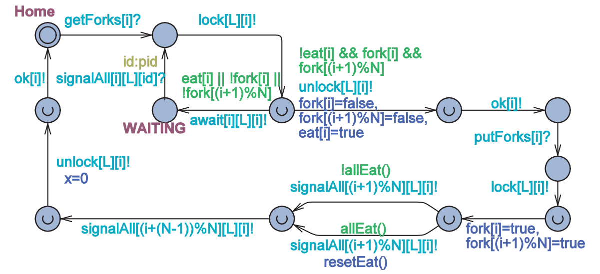

Fig. 12. Manager automaton based on lock/condition monitor

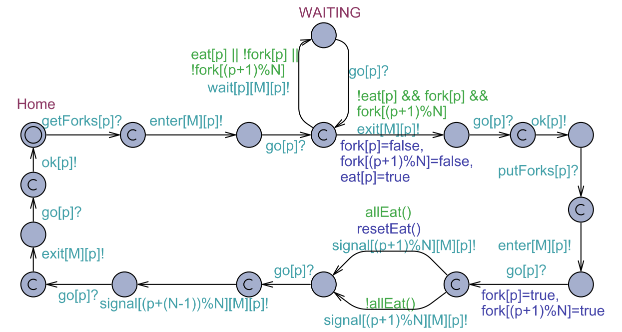

Fig. 13. Manager based on the Hoare monitor

given philosopher is waiting for forks. The latter is used for implementing a simple strategy for avoiding starvation. Philosophers are supposed to eat at turns. A turn finishes when all philosophers have eaten. A `getForks` request is not responded either because some fork is unavailable or the philosopher has already eaten in the current turn. The latest philosopher who eats resets the array `eat` so as to start a new turn.

All the three queries suggested in section III, are now satisfied. Query 3 concerning proving absence of starvation deserves some further comments. It is a liveness property. UPPAAL is most apt to verify safety and bounded liveness properties. General liveness can be difficult to assess. In a normal location, in fact, an automaton can stay an arbitrary amount of time. To help checking liveness properties, Urgent/Committed locations or urgent channels should be used. In Fig. 10, the use of committed locations was preferred.

A bounded liveness property for the model of Fig. 9 and Fig. 10 concerns the worst case amount of time a philosopher stays in the `WAITING` location of its manager, waiting for the other colleagues to complete the current turn. Using $N = 5$ philosophers, and adding a decoration clock y to the `Manager` model, which is reset at each `getForks[i]?` request, the following query was issued to the UPPAAL verifier:

```
A[] Manager(0).WAITING imply Manager(0).y<=64
```

The query was found satisfied, but changing the upper bound to 63 the query no longer holds. This result corresponds to a $(10-(2+4))*(N-1)$ remaining thinking time for the other partners, and then a $12*(N-1)$ worst case eating time of remaining colleagues.

As a final remark, except for the `GO[pid]` and `ok[pid]` channels which are required only in the UPPAAL models, the automata in Figures 9 and 10 can directly be expressed in Java code.

B. Manager based on JMonitor

Using the built-in Java monitor, a `Manager` model like that shown in Fig. 11 can be achieved. With respect to the semaphore based solution, it requires less space and time for the analysis.

A similar model to that in Fig. 11 was built using the lock/condition monitor, which is slightly more efficient due to reduced partial order following a signaling operation. The new model is portrayed in Fig. 12.

It should be noted that since enter/lock requests can be delayed by the monitor being already locked, such operations should not exit from urgent locations. In addition, because of broadcast signaling, there is no need to pay for the `w[pid]` boolean array used in the semaphore based solution.

C. Manager based on Hoare monitor

It was interesting achieving a Hoare monitor based model for manager, to compare expressiveness and guarantees during signaling with Java built-in or lock/condition based versions. The model is portrayed in Fig. 13. As expected, this particular example does not allow to exploit the normal guarantees of the Hoare monitor: i.e. that a signaled process waiting on a condition is sure its condition holds when awaken by a signal. In fact, putting forks is only a partial fulfillment for the precondition of adjacent philosophers to be able to get forks. As a consequence, without going to put much burden on the signaler processes, the right solution consists in envisioning the while-loop also in a signalee so as to come back to waiting if the precondition for awaking does not actually hold. Without the while-loop, UPPAAL confirms the model is incorrect.

Model of Fig. 13 is more expensive in terms of space and time for verification with respect to models in Fig. 11 and Fig. 12. This is due to the use of queues for conditions and related bookkeeping data.

D. Manager based on Active Oberon monitor

This version of the `Manager` model is illustrated in Fig. 14.

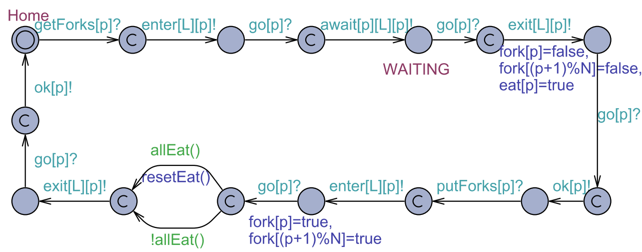

Fig. 14. Manager automaton based on the Active Oberon monitor

The following is the `eval()` function which was prepared for working with the Dining-Philosophers problem:

```
bool eval( cid c, mid m ){
  if( !eat[c]&&fork[c]&&fork[(c+1)%N] )
    return true;
  return false;
}//eval
```

The solution in Fig. 14 is the most concise and easy to follow from the modeler/programmer point of view. Its analysis performance, though, is similar to the Hoare monitor version because the model has almost the same data requirements.

Verification experiments with $N = 5$ philosophers were carried out on a Win 8, 12GB, Intel Core i7-3770K, 3.50GHz. To figure out efficiency of the various models, the query which checks for the absence of deadlocks lasts in the worst case in about 25sec with a RAM peak of about 100MB.

VI. Conclusion

The UPPAAL library of timed automata (TA) proposed in this paper is effective for modeling and verification (M&V) of Java-based concurrent and timed programs. It includes both semaphores and monitor control structures. The Java built-in monitor or its refinement based on lock/condition are often preferable both from the M&V perspective and the implementation viewpoint. The Active Oberon monitor offers the most concise level for modeling and implementing a thread-safe class.

Being not primitive in Java, Hoare monitor and Active Oberon monitor classes were achieved on top of semaphores. By the way, a `BinarySemaphore` class was also realized which is weakly bisimilar to the automaton in Fig. 2.

A nondeterministic point e.g. in the Hoare monitor class accompanies the implementation of the `wait` operation which in general must free the lock and put the requesting process to sleep. Whereas UPPAAL atomic actions hide the problem (e.g. through a committed location), in a concrete implementation the alea point could be handled by ensuring that before relinquishing the lock the wait operation starts a new time-slice. This could be achieved by using the Thread `yield()` method. However (see discussion in [15] at page 287) the `yield()` method can often behave as a no operation. A better provision could be a `Thread.sleep(1)` if 1 millisecond is the time resolution of the underlying operating system.

The library is currently in use in an undergraduate course on systems programming and the response of students is positive.

A major benefit of the catalog and of the UPPAAL model checker rests on the possibility of favoring a *reasoning on concurrency*.

On-going and future work are geared to:

- Optimizing the library so as to improve the efficiency of model checking activities.
- Extending the library with other concurrency control structures, e.g. based on the concept of software transactional memory [14] which delivers a different and attractive style of concurrent programming.
- Extending the approach based on the UPPAAL model checker to M&V of lock-free concurrent objects [14] which are often perceived as a grand challenge for an exploitation, in parallel and embedded software systems, of the computing potential of current and future multi-core/many core machines.

Acknowledgment

Authors are grateful to Christian Nigro for his contribution during the design and realization of the UPPAAL library proposed in this paper and its support in Java.

References

[1] F. Cicirelli, L. Nigro, and F. Pupo, “Modelling and verification of concurrent programs using UPPAAL,” in *Proc. of 25th European Conference on Modelling and Simulation*, Krakow, Poland, 2011, pp. 525–533.
[2] R. Alur and D. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.
[3] G. Behrmann, A. David, and K. Larsen, “A tutorial on UPPAAL,” in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS 3185, M. Bernardo and F. Corradini, Eds. Springer, 2004, pp. 200–236.
[4] “UPPAAL, on-line,” www.Uppaal.org.
[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, pp. 8160–8186, 2012.
[6] F. Cicirelli, A. Furfaro, L. Nigro, and F. Pupo, “Development of a schedulability analysis framework based on PTPN and UPPAAL with Stopwatches,” in *Proc. of the IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications (DS-RT'12)*, Dublin, Ireland, 2012, pp. 57–64.
[7] R. Hamberg and F. Vaandrager, “Using model checkers in an introductory course on operating systems,” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 101–111, 2008.
[8] J. Magee and J. Kramer, *Concurrency: state models & Java programs*. John Wiley & Sons, Ltd., 2006.
[9] W. Stallings, *Operating Systems: Internals and Design Principles*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2005.
[10] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.
[11] A. S. Tanenbaum, *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2001.
[12] P. Reali, “Active oberon language report,” http://bluebottle.ethz.ch/languagereport/index.html, 2002.
[13] B. Lampson and D. Redell, “Experience with processes and monitors in Mesa,” in *Proc. of the seventh ACM symposium on Operating systems principles*, Pacific Grove, California, USA, 1979, pp. 43–44.
[14] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, revised version of first ed. Elsevier Science Limited, 2012.
[15] B. Joshua, *Effective Java*, 2nd ed. Addison Wesley, 2008.