Model Checking Mutual Exclusion Algorithms Using UPPAAL

Franco Cicirelli, Libero Nigro and Paolo F. Sciammarella

Abstract This paper proposes an approach to modelling and exhaustive verification of mutual exclusion algorithms which is based on Timed Automata in the context of the popular UPPAAL toolbox. The approach makes it possible to study the properties of a mutual exclusion algorithm also in the presence of the time dimension. For demonstration purposes some historical algorithms are modelled and thoroughly analyzed, going beyond some informal reasoning reported in the literature. The paper also proposes a mutual exclusion algorithm for $N \geq 2$ processes whose model checking confirms it satisfies all the required properties.

Keywords Mutual exclusion algorithms · Model checking · Timed automata · UPPAAL

1 Introduction

Mutual exclusion is the well-known problem faced by a collection of $N \geq 2$ asynchronous processes sharing some data variables. To avoid interference and ultimately unpredictable behavior of shared data, processes should access one at a time the common variables, i.e., they should execute one at a time their critical sections.

F. Cicirelli · L. Nigro (✉) · P.F. Sciammarella
Department of Informatics, Modelling, Electronics and Systems Science (DIMES), University of Calabria, Rende, CS, Italy
e-mail: l.nigro@unical.it

F. Cicirelli
e-mail: f.cicirelli@dimes.unical.it

P.F. Sciammarella
e-mail: p.sciammarella@dimes.unical.it

R. Silhavy et al. (eds.), *Software Engineering Perspectives and Application in Intelligent Systems*, Advances in Intelligent Systems and Computing 465,
DOI 10.1007/978-3-319-33622-0_19

Different solutions are described in the literature ranging from hardware to software based solutions. Software solutions can depend on blocked-queue semaphores or on a monitor structure. More challenging are solutions which make use of a few weak and unfair semaphores [1, 2] or they can be *pure*-software solutions, i.e., based on some communication variables and synchronization protocols regulating the *try* (or *enter*) and the *exit* sections which respectively precede and follow an execution of the critical section (CS). A process engaged into the try-protocol competes for entering its CS. A process not interested in entering its CS, e.g., because it has just finished executing its CS and the exit-protocol, runs its so called non critical section (NCS).

Methods for proving the correctness of mutual exclusion algorithms include proof-theoretic approaches [1] and model checking [3, 4]. Model checking can be preferable because it ensures formal modelling and automates the analysis activities. In particular, the use of Timed Automata [5] in the context of the popular and efficient Uppaal toolbox [6] is advocated in this work, because it permits an in depth exploration of the behavior of an algorithm also along the time dimension, which can be difficult in a proof-theoretic approach.

This paper proposes a modelling approach based on Uppaal and applies it to property checking of two known algorithms and to a new developed one. The afforded analysis enables to go beyond the informal reasoning often reported in the literature (see, e.g., the different indications about the worst case waiting time of competing processes in the $N > 2$ scenario of the Peterson algorithm [7, p. 101]). All of this testifies the known difficulties in mastering a concurrent solution due to complex action interleavings.

The paper argues that the proposed approach is of interest today in the software engineering domain because it helps the development of correct concurrent systems which can exploit the execution performance of multi-core machines.

The remainder of this paper is organized as follows. First the basic concepts, concurrency model and specification language of Uppaal are summarized. Then the proposed modelling approach for mutual exclusion algorithms is presented. After that the Dekker's algorithm for 2 processes [8], the Peterson's algorithm for $N \geq 2$ processes [9] and a new algorithm for $N \geq 2$ processes are modelled and thoroughly analyzed. Finally, conclusions are presented with an indication of on-going and future work.

2 An Overview to Uppaal

A timed automaton [5, 6] is a finite automaton augmented with a set C of real-valued variables named *clocks*. Clocks model the time elapsing and are assumed to grow synchronously at the same pace of the hidden system time. Constraints, of the form $x \sim k$ or $x - y \sim k$ where x and y are clocks, k is a non-negative integer and $\sim \in \{\leq, <, =, >, \geq\}$, are called *clock constraints* and can be introduced to restrict the behavior of the automaton. The set of all possible

constraints over a set of clocks C is denoted by $B(C)$. A set of clock constraints used to label an edge it is called a *guard*. Clock constraints of the type $x \sim k$ can also be used to label locations and are called *invariants*. An automaton can stay in a location as long as the clocks satisfy the location invariant. Additionally, edges can be labeled by a set of clocks which are reset as the corresponding transition is taken, and by an *action* label ranging over a finite alphabet Σ.

Formally, a timed automaton A is as a tuple (L, l_0, E, I), where:

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $E \in L \times B(C) \times \Sigma \times 2^C \times L$ is a set of edges and
- $I: L \to B(C)$ is the invariant function.

The notation $l \to^{g,a,r} l'$ stands for $(l, g, a, r, l') \in E$, where g is a guard, a is an action, r is a set of clocks to be reset. The state of a timed automaton is a pair (l, u) where l is a location and u is a clock valuation, i.e., a function that associates a non-negative real value to each clock, and $(l_0, 0_{|C|})$ is the initial state. Let u be a clock valuation on a set of clocks C, and $d \in R_+$ a delay; $u + d$ denotes the clock assignment that maps all $x \in C$ to $u(x) + d$ and $r \subseteq C$, $[r \mapsto 0]u$ denotes the clock assignment that maps to 0 all clocks in r and agree with u for the other clocks. The semantics of a TA is defined by two state transition rules, namely *delay* and *action* transitions:

- $$(l, u) \to^{d} (l, u + d) \text{ if } (u + d) \in I(l) \text{ for any } d \in R_+$$
- $$(l, u) \to^{a} (l', u') \text{ if } l \to^{g,a,r} l', u \in g, u' = [r \mapsto 0]u \text{ and } u' \in I(l').$$

In the general case, for a given state (l, u) there are a continuous infinity values of d for which the first rule can be applied and hence, while the first state component l can assume only a finite set of values, the possible clock valuations are continuous infinity. Therefore, the state-space of a TA is infinite and uncountable. Despite this, reachability analysis of TA is decidable [5] because the infinite states of a TA can be partitioned into a finite set of equivalence classes called *zones*. A zone is a solution of a set of clock constraints.

TA can be composed to form a network of concurrent TA whose semantics depends on action interleavings and hand-shake synchronizations. UPPAAL adopts the notion of a *channel* for input and output action synchronization and uses a CSP-like notation. The edge of automaton labeled with `ch!` (output action), where `ch` is a channel, matches with an edge of another automaton labeled with `ch?` (input action). At a given time it may exist more than one pair of enabled and matched edges in which case a choice is made non-deterministically. Taking a transition (edge) in an automaton denotes an *atomic action* in the TA concurrent model. Moreover, the update of a sender is executed *before* that of a receiver.

The UPPAAL model-checker generates *on-the-fly* the zone graph of a network of TA for checking a subset of TCTL (Timed Computation Tree Logic) formulas [10] as in the following:

- $E <> \varnothing$ (Possibly $\varnothing$, i.e., a state exists where $\varnothing$ holds)
- $A[] \varnothing$ (Invariantly $\varnothing$, equivalent to: $not\, E <> not\, \varnothing$)
- $E[] \varnothing$ (Potentially Always $\varnothing$, i.e., a state path exists over which $\varnothing$ always holds)
- $A <> \varnothing$ (Always eventually $\varnothing$, equivalent to: $not\, E[]\, not\, \varnothing$)
- $\varnothing - - > \psi$ ($\varnothing$ always *leads-to* ψ, equivalent to: $A[]\, (\varnothing\, imply\, A <> \psi)$)

where $\varnothing$ and ψ are state properties (formulas), e.g., clock constraints or boolean expressions over predicates on locations. Verification of properties expressed by the above formulas reduces to reachability analysis, which is accomplished by traversing the zone graph associated with a TA network, and intersecting the zone associated with the formula with the zone of visited state nodes.

To facilitate the modelling task, integer variables with a bounded set of values and array of integers, clocks or channels can be introduced. A notion of automata *templates* which can be instantiated with different values for their parameters is supported. Integer variables, clocks and channels can be declared globally into a TA network, locally to a template, or used as template parameters. Locations can also be labeled as being *committed* (C) or *urgent* (U) both of which must be abandoned with no time passing. The exits from simultaneous committed locations are interleaved to one another, as well as the exits from urgent locations are interleaved each other, but committed locations have precedence over urgent locations. Channels can be declared to be urgent. An enabled synchronization on a urgent channel is required to occur without time passage.

UPPAAL consists of an editor, a simulator and the model checker. It is worth mentioning the possibility of building a counterexample (or diagnostic trace) of a not satisfied property, which can be analyzed in the simulator. The diagnostic trace furnishes evidence of a sequence of transitions bringing the model in a state not fulfilling the property.

3 Modelling Approach Based on UPPAAL

The common structure of processes regulated by a given mutual exclusion algorithm is assumed to be the following:

```
process(i) = loop NCS; try-protocol; CS; exit-protocol;
endloop.
```

The parameter *i* identifies the generic instance of the process abstracted by an UPPAAL template. The critical section is modelled by a normal location (*CS*) with an associated clock invariant to enforce a maximum duration (*C* constant). The used clock is a local variable of the process. The non critical section is mapped to a

normal location (*NCS*) which acts as the initial one for the process automaton. The absence of a clock invariant for *NCS* ensures an arbitrary time can elapse before the next arrival of the process competing for entering its *CS*.

A critical point of process modelling is the representation of the actions of the try and exit protocols. To reproduce action interleaving and non determinism among processes, each elementary action (a variable assignment or a single condition test) is mapped onto an atomic action of the UPPAAL model, i.e., an update or a guard of a command of an edge exiting from a location. In order to ensure the *finite-delay* or *weak-fairness* property of the concurrent model [1], that is an action which is continuously enabled eventually fires, one could attach to the source location of an action a clock invariant mirroring its maximal duration. For simplicity, though, without any generality loss, the source location of an action is modelled as an urgent one. Therefore, each basic action is supposed to consume a negligible time with respect to the *CS*. However, to enable time advancement of a process, e.g., in the try-protocol, a critical point is the modelling of busy-waiting loop structures of the type: **`while`** `(cond) body`, where `cond` can be a complex condition and `body` can be void. Such loops are achieved by using a normal location (where time is allowed to pass) whose exiting is forced by an edge with guard `!cond` (`cond` is thus evaluated atomically and can exploit a user-defined function or the basic operators `exists`, `forall`, and `sum` which UPPAAL offers for checking arrays) and an urgent channel synchronization (see the unicast channel `synch` in Fig. 4). To preserve action interleaving and non determinism of processes during a complex condition evaluation, though, the exiting from a busy waiting location represents a *tentative* exit, thus it is followed by a detailed evaluation of `!cond` split into its component parts. During the detailed evaluation, the control can be transferred again to the busy-waiting location would `cond` be found still satisfied.

The worst case waiting time for a competing process can then be evaluated by observing the (hopefully bounded) number of *CS* s which are executed on behalf of competing processes, before the current one is enabled to enter its *CS*. Such a number (overtaking factor) can be determined either by counting the number of by-passes experimented by a competing process or by bounding the process clock during the try protocol.

The following properties will be verified on a modelled mutual exclusion algorithm.

Mutual exclusion (safety)–One and only one process can enter its *CS* at one time.

Deadlock free (safety)–The execution of the try/exit protocols in no case would induce a deadlock among processes.

Progress (liveness)–A process executing into its *NCS* would not forbid other processes to enter their *CS*.

Starvation free (bounded liveness)–A process competing for entering its *CS* eventually succeeds; that is, the number of by-passes of other competing processes is bounded.

4 Dekker's Algorithm

It has been the first algorithm proposed in 1962 for solving the mutual exclusion problem [8] for $N = 2$ processes. Three globals are used: boolean $b[0], b[1]$ and the integer k. Figure 1 shows the code of process i ($j = 1 - i$ denotes the partner process). $b[i] = true$ expresses willingness of process i to entering its *CS*. Process i can actually enter its *CS* when the turn variable k evaluates to i. The initial value of k can be either 0 or 1.

Figures 2, 3 and 4 show a corresponding UPPAAL model achieved according to the approach described in Sect. 3. The template process in Fig. 3 is named *Process* and admits one single *const* parameter i of type *pid*, the integer subrange type of process identifiers. Clock x serves to measure the elapsed time during the *CS*, or the waiting time during the try-protocol. Global declarations of the model and the local declarations of the $Process(i)$ template are collected in Fig. 2. Also the system configuration with implicit instantiation controlled by the *pid* parameter of *Process* (the two instances have names $Process(0)$ and $Process(1)$) is portrayed in Fig. 2. The *Synch* automaton is depicted in Fig. 4. It is always ready to send a signal on the urgent *synch* channel.

Absence of deadlocks was checked by the query $A[]!deadlock$ (*deadlock* is an UPPAAL keyword) which is satisfied. Mutual exclusion was checked by the query: $A[]forall(i:pid)forall(j:pid)Process(i).CS \&\& Process(j).CS\ imply\ i==j$ which is satisfied (would two processes be simultaneously in their *CS*, then the two processes are necessarily the same process). The progress property was checked by a query like $E <> Process(0).CS \&\& Process(1).NCS$ which is satisfied (it should be noted that being identical the two processes, one query suffices).

General liveness of the model was checked by query: $Process(0).while -->Process(0).CS$ which is *not* satisfied. This in turn mirrors the model (and the algorithm) has a zeno-cycle, which means any process can

Fig. 1 Dekker's algorithm

```
while( true ){
    NCS
    b[i] = true;
    while( b[j] ){
      if( k == j ){
        b[i] = false;
        while( k == j );
        b[i] = true;
      }
    }
    CS
    k = j;
    b[i] = false;
}
```

```
//Global declarations
const int C=4;
const int N=2;
typedef int[0,N-1] pid;
bool b[pid]={false,false};
pid k=0; //or 1
urgent chan synch;

//Process local declarations
clock x;
const pid j=1-i;

// List one or more processes to
// be composed into a system.
system Process,Synch;
```

Fig. 2 UPPAAL declarations

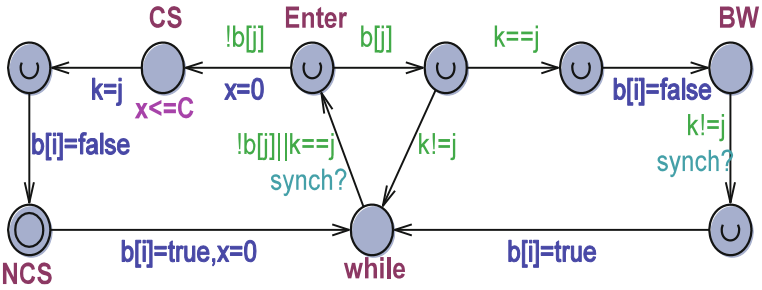

Fig. 3 Dekker's *Process*(i) automaton

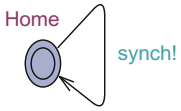

Fig. 4 The Synch automaton

enter/exit its critical section consecutively an infinite number of times and in 0 time (both *CS* and *NCS* are exited in 0 time). Therefore, the overtaking factor is theoretically infinite. The zeno-cycle disappears by assuming the critical section has a duration strictly greater than zero (time-dependent behavior). As a consequence, the guard $x > 0$ was added to the model in Fig. 3 on the edge exiting the *CS* location and the remaining properties were verified on this modified model.

The starvation-free property was assessed by finding the supreme value of the clock x using the query: $\sup\{Process(0).while\}: Process(0).x$. Clock x is reset when the process starts competing, i.e., it reaches the *while* location. Similarly, it is reset when entering the *CS* location, i.e., abandoning the *Enter* location with $b[j] == false$. The maximum value of the waiting time was found to be $2*C$, i.e., there are, in the worst case, two by-passes of $Process(1)$ whereas $Process(0)$ is competing and vice versa.

The worst case can happen when both processes are in the *while* location and $b[0] = true \wedge b[1] = true \wedge k = 1$. As a consequence $Process(0)$ reaches the *BW* location waiting for $k! = 1$. One time $Process(1)$ enters its *CS* and consumes *C* time units. On exiting the *CS* location, it executes $k = j$ (where j denotes $Process(0)$) and resets its $b[i]$ then enters its *NCS* which is immediately abandoned, i.e., in 0 time, then starts again competing by raising $b[i] = true$. Always in 0 time $Process(1)$ reaches again *Enter* and being $b[j] == false$ it re-enters *CS*, thus consuming another critical section. The problem is that, for non determinism, $Process(0)$ in *BW* cannot immediately benefit of the update $k = j$ executed by $Process(1)$. But before exiting the second *CS* certainly $Process(0)$ reaches its *while* location. Now being $k = 0$ the next access to critical section will be granted only to $Process(0)$. The above behavior was confirmed by Uppaal by answering to the query $E <> Process(0).while \,\&\&\, Process(0).x == 2*C$ and by generating a corresponding diagnostic trace. It is worth noting that the worst case waiting time of processes also holds on the original model in Fig. 3.

5 Peterson's Algorithm

The algorithm was proposed in [9] to handle both the case $N = 2$ and the more general case of $N > 2$ processes. Figure 5 portrays the general version of the algorithm and the declarations of a corresponding Uppaal model.

Processes are numbered $1, 2, \ldots, N$. They have to climb a ladder in order to enter the *CS*. The ladder has *N*-1 levels numbered from 1 to *N*-1. The process who reaches the *N*-1 highest level enters its *CS*. In the case more processes try to step a same level, at least one of them stops moving. Two arrays are used: $q[pid]$ and $turn[level]$. $q[i]$ contains the level occupied by $Process(i)$. $turn[j]$, where j is a level, stores the identifier of a process at level j.

The Uppaal model of Peterson's algorithm was studied for $N \in [2..5]$. First of all the query $Process(1).for -- > Process(1).CS$ was found not satisfied, testifying, as in the Dekker's algorithm, the existence of a zeno-cycle which disappears by adding the guard $x > 0$ to the edge exiting the *CS* location (see Fig. 6). Using queries similar to the Dekker's model, it was found the Peterson's model satisfies the mutual exclusion and the progress properties. In addition the model is both deadlock and starvation free. The overtaking factor was determined using the query:

Fig. 5 Peterson's algorithm and Uppaal declarations

```
while( true ){
  NCS
  for( j=1; j<=L; j++ ){
    q[i]=j;
    turn[j]=i;
    wait until(
    (for all k≠i,(q[k]<j))or
    (turn[j]≠i) );
  }
  CS
}

//Uppaal global declarations
const int C=4;
const int N=…;
const int L=N-1; //levels
typedef int[1,N] pid;
typedef int[1,L] level;
int[0,N] q[pid];
pid turn[level];
urgent chan synch;

//Process local declarations
int[1,L+1] j;
int[1,N+1] k;
clock x;
```

Fig. 6 Peterson's *Process*(i) automaton

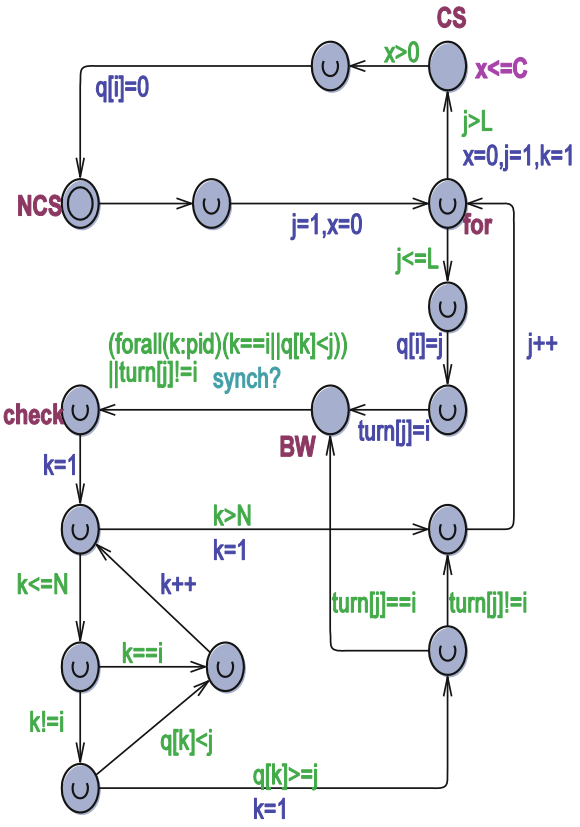

Table 1 Overtaking factor versus N

N	$WCWT$	OF
2	4	1
3	12	3
4	24	6
5	40	10

$\sup\{Process(1).for\}: Process(1).x$. Table 1 shows the collected results for $N \in [2..5]$. $WCWT$ denotes the emerged worst case waiting time for a trying process in the *for* location, $OF = WCWT/C$ is the corresponding overtaking factor, where $C = 4$ time units is the maximum duration of the critical section.

It emerged that in the Peterson's algorithm a trying process has a worst case overtaking factor of $((N-1)*N)/2$. This result agrees with the calculation reported in [11] but contrasts with the indications contained in [12, 13] which predicted a worst case number of by-passes of $N-1$.

The size and possible values of the arrays $q[]$and$turn[]$ critically affect the dimension of the state graph and the model checking activities. To favor the model checker, provisions were taken in Fig. 6 to assign a default value to a variable as soon as it gets unused (see the integer variables j and k). Checking the overtaking factor in the case $N = 5$ required about 31 min of wall clock time with a peak of used memory of about 19 GB. Experiments were carried out using UPPAAL version 4.1.19 64bit on a Linux machine, Intel Xeon CPU E5-1603@2.80 GHz, 32 GB.

6 Proposed Algorithm

As a further demonstration of the application of the proposed modelling and verification approach, the following describes an example of a new algorithm[1] for $N \geq 2$ processes (see Figs. 7, 8, 9) which uses fewer variables and it is more efficient than, e.g., the Peterson's algorithm. The processes are numbered $0, 1, \ldots, N-1$.

The algorithm uses an array $b[pid]$ of N booleans, two boolean variables *lock* and *cs_busy*, and a *turn* integer variable. Initially all the booleans are set to *false* and $turn = 0$. To access *CS*, $Process(i)$ must "climb a ladder" of three steps.

$Process(i)$ starts competing by putting *true* in $b[i]$. Climbing can fail at each of the first two steps (denoted respectively by the actions $turn = i$ and $lock = true$) thus forcing $Process(i)$ to restart trying (from the label *L*). The last step is signaled by the action $cs\_busy = true$. After *CS*, $Process(i)$ puts *false* in $b[i]$, then it makes a modular search in the b array looking for the first process j, if there are any, which is trying. The critical section grant is then transferred to it by assigning *false* to $b[j]$,

[1]The contribution of Domenico Spezzano to the design of this algorithm is acknowledged.

Fig. 7 Proposed algorithm

```
while( true ){
   NCS
   b[i]=true;
L: while((turn!=i||lock)&&b[i]);
   if(!cs_busy) turn=i;
   if((turn!=i||lock)&&b[i]) goto L;
   lock=true;
   if((turn!=i||cs_busy)&&b[i]){
     if(!cs_busy) lock=false;
     goto L;
   }
   cs_busy=true;
   CS
   b[i]=false; j=(i+1)%N;
   while(j!=i&&!b[j])j=(j+1)%N;
   if(j==i){
     cs_busy=false;
     lock=false; turn=0;
   }
   else b[j]=false;
}//while
```

Fig. 8 Uppaal declarations

```
//Global declarations.
const int C=4;
const int N=4;
typedef int[0,N-1] pid;
bool b[pid]=
{false,false,false,false};
bool lock=false;
bool cs_busy=false;
pid turn=0;
urgent chan synch;


//Process local declarat.
pid j;
clock x;
```

otherwise the trying protocol is re-initialized. Figure 9 portrays the *Process*(i) template automaton.

The model was verified for a number of processes N ranging in the interval [2..5]. It satisfies *all* the required properties without assumptions about the *CS* duration. The algorithm has no zeno-cycle and the overtaking factor is linear (although waiting processes are not *FIFO* managed) in the number of processes,

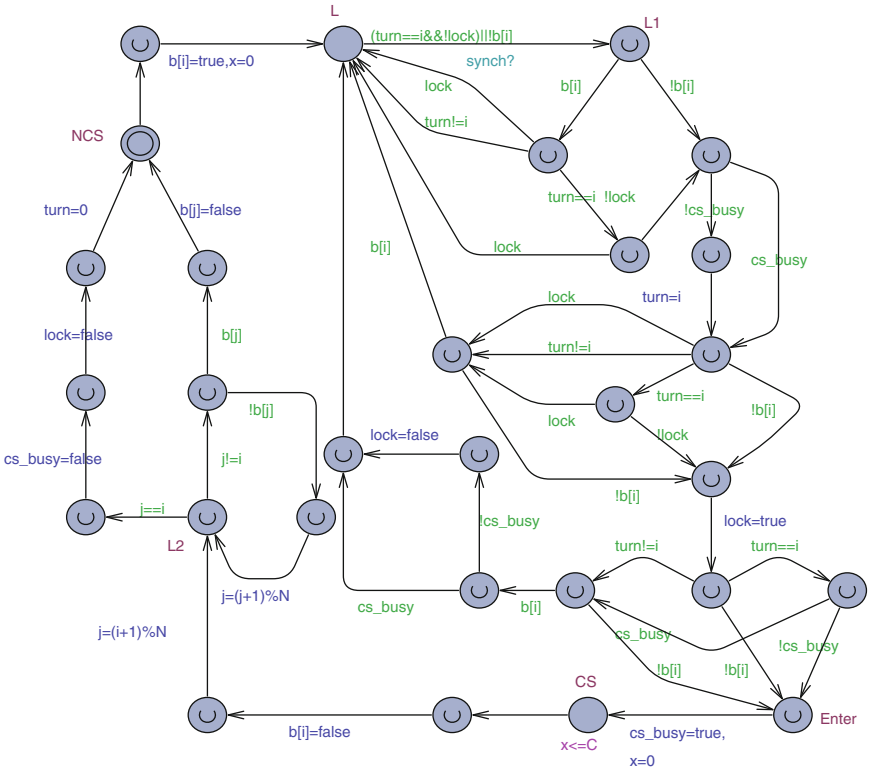

Fig. 9 Proposed $Process(i)$ automaton

i.e., in the worst case, a competing process has to wait for one by-pass of each of the remaining $N-1$ processes. For $N = 5$, a verification query is responded in about $1min$ of wall clock time with a memory peak of about $950MB$. Starvation-free behavior was checked by the satisfied queries: $A[]Process(0).Enter\ implyProcess(0).x <= (N-1)*C$ and $Process(0).L-->Process(0).CS$.The first query fails when the clock $Process(0).x$ is compared with $(N-1)*C-1$ or a lower quantity. The second query ensures bounded waiting for a trying process, and absence of zeno-cycles.

7 Conclusions

The analysis of concurrent systems is notoriously hard for the complex action interleavings occurring among the involved processes. All of this can introduce subtle errors in a concurrent program which cannot be dominated by informal reasoning or by program testing.

Mutual exclusion algorithms represent an active area of concurrency since the sixties, and some errors are famous about the characterization of their properties [7]. Nowadays, the availability of mature and powerful model checkers like Uppaal [6] has the potential to improve the situation by enabling formal modeling and exhaustive exploration of a concurrent program.

This paper proposes a model checking approach based on Timed Automata and Uppaal for the systematic analysis of mutual exclusion algorithms. With respect to proof-theoretic approaches (e.g., [1]) the proposal facilitates an in-depth verification of an algorithm also in the presence of the time dimension.

The approach was successfully applied to many existing algorithms of which known properties were confirmed and new properties disclosed in doubt situations. The article contributes to a better understanding of two historical algorithms like Dekker [8] and Peterson [9]. The approach was also exploited in the design and analysis of a new algorithm for $N \geq 2$ processes, which fulfils all the required properties. Prosecution of the work is geared at improving the approach and to experiment with its application to other algorithms.

References

1. Hesselink, W.H., IJbema, M.: Starvation-Free Mutual Exclusion With Semaphores. Formal Aspects of Computing (2011). DOI 10.1007/s00165-011-0219-y
2. Cicirelli, F., Nigro, L.: Modelling and verification of starvation-free mutual exclusion algorithms based on weak semaphores. Proc. FedCSIS **2015**, 785–791 (2015)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
4. Cicirelli, F., Furfaro, A., Nigro, L.: Model checking time-dependent system specifications using time stream petri nets and Uppaal. Appl. Math. Comput. **218**, 8160–8186 (2012)
5. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**, 183–235 (1994)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science, vol. 3185, pp. 200–236. Springer (2004)
7. Alagarsamy, K.: Some myths about famous mutual exclusion algorithms. ACM SIGACT News **34**(3), 94–103 (2003)
8. Dijkstra, E.W.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages, pp. 43–112. Academic Press, New York (1968)
9. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
10. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for Real-time systems. In: Proceedings of Seventh Annual IEEE Symposium on Logic in Computer Science, pp. 414–425. IEEE Computer Society Press (1990)
11. Raynal, M.: Algorithms For Mutual Exclusion. MIT Press (1986)
12. Kowalttowski, T., Palma, A.: Another solution of the mutual exclusion problem. Inf. Proce. Lett. **19**(3), 145–146 (1984)
13. Hofri, M.: Proof of a mutual exclusion algorithm—a 'class'ic example. ACM SIGOPS OSR **24**(1), 18–22 (1990)