DOI: 10.15439/2015F32
ACSIS, Vol. 5

# Modelling and Verification of Starvation-Free Mutual Exclusion Algorithms based on Weak Semaphores

Franco Cicirelli, Libero Nigro
Laboratorio di Ingegneria del Software
Dipartimento di Ingegneria Informatica Modellistica Elettronica e Sistemistica
Università della Calabria, Italy
{f.cicirelli@dimes.unical.it, l.nigro@unical.it}.

*Abstract—* **This paper proposes an original framework for modelling and verification (M&V) of starvation-free mutual exclusion algorithms based on weak semaphores, that are without a built-in waiting-process queue. The goal is to support the implementation of light-weight starvation-free semaphores useful in general concurrent systems including cyber physical systems. The M&V approach depends on UPPAAL. First known weak semaphores are modelled. Then they are exploited for model checking classic algorithms. Known properties are retrieved but subtle new ones are discovered. As part of the developed approach, a new algorithm is proposed which uses two semaphores of the weakest type, N bits (N being the number of processes) and a counter. This algorithm too is proved to be correct.**

## I. Introduction

MUTUAL exclusion is the well-known problem of synchronizing a group of concurrent processes (or threads) sharing some data variables, so as to avoid interferences on shared data. The problem is to ensure that processes can enter their critical section (i.e., a block of instructions accessing/modifying the shared data) one at a time. To be acceptable, though, a mutual exclusion algorithm should be also starvation-free, that is a process waiting to enter its own critical section should experiment a bounded waiting time. This in turn favors process fairness.

Commonly, mutual exclusion can be based on semaphores or on monitor locks. This paper focuses on starvation-free mutual exclusion algorithms based on *weak semaphores*, i.e., semaphores without an in-built process waiting queue ensuring a first-in-first-out awakening policy.

The results of this paper can be exploited for implementing light-weight starvation-free semaphores, useful in general concurrent applications and cyber physical systems (e.g., [1]), and also in distributed shared memory systems where it is challenging to build a classical queue-based semaphore when the processes belong to distinct physical computation nodes (address spaces).

The goal is to propose an original approach based on Timed Automata (TA) in the context of the UPPAAL toolbox [2], for modelling and verification through model checking [3] of any mutual exclusion algorithm designed on top of weak semaphores. The goal is similar to that described in [4] where an approach based on the use of the PVS theorem prover was developed. This paper argues that the use of a toolbox like UPPAAL can be preferable as a proving framework because it avoids the mathematical formalization necessary to specify and check properties of an algorithm. In addition the approach permits to disclose subtle aspects of a modelled algorithm, e.g., related to timing, which are normally out of reach of a theorem prover.

The modelling and verification (M&V) approach is applied to known classic algorithms, e.g. [5]-[8], of which are confirmed known properties. Nevertheless, some new properties (e.g., the existence of a zeno-cycle and of a time-sensitive behavior which affects the kind of the weak semaphores which can be used) are disclosed which were not previously documented in the literature. As a part of the accomplished work a novel algorithm based on the Morris one [5] is proposed which rests only on two weak semaphores, one counter and N bits. This algorithm too is proved to be starvation-free.

The developed proving framework can provide some new arguments on the E. Dijkstra conjecture [9] about the impossibility to build a starvation-free semaphore using only weak semaphores.

The paper is structured as follows. First an overview of the UPPAAL M&V concepts is furnished. Then the three known types of weak semaphores are introduced and modelled into UPPAAL. After that, a common vision [4] of classic starvation-free mutual exclusion algorithms is discussed. Then the developed M&V approach is applied to classic algorithms as well as to a new one proposed in this paper. A comparison of the algorithms properties is finally presented. Some indications about on-going and future work are given in the conclusions.

## II. UPPAAL Concepts

UPPAAL [2] is a popular and efficient toolbox based on Timed Automata (TA) [10] suited for modelling and verification of real-time systems. A timed automaton 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

This work was not supported by any organization

$x - y \sim k$ 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. A set of clock constraints used to label an edge it is called a *guard* and determines the possible values which can be assumed by the involved clocks for the corresponding state transition to be allowed. 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 attached to the location. Additionally, edges can be labeled by a set of clocks, which are reset as the corresponding transition is taken, and by an *action* label.

TA can be composed to form a network of concurrent TA whose semantics is based on interleaving of actions as well as 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 state graph of a network of TA (see, e.g., [11]) for checking formulas as in the following:

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

where $\emptyset$ and $\psi$ are state properties, e.g., clock constraints or boolean expressions over predicates on locations.

In addition to the support for classic TA, UPPAAL provides integer variables with a bounded set of values, arrays and structs, and a notion of automata *templates* which can be instantiated multiple times by specifying different values for their parameters.

Locations can also be labeled as being *committed* (C) or *urgent* (U) both of which must be abandoned with no time passing. Committed locations have precedence over urgent locations. UPPAAL provides also broadcast synchronizations. Channels can be declared to be urgent. An enabled synchronization on a urgent channel is required to occur without time passage.

Finally, it is worth mentioning the possibility of building a counterexample (i.e., diagnostic trace) of a not satisfied property, which can be analyzed in the simulator of the toolbox.

## III. MODELLING WEAK SEMAPHORES

A semaphore is an abstract object which hides an integer variable which can only be modified by the two atomic operations `P` and `V`. Fig. 1 shows an UPPAAL model of a basic *plain* binary semaphore, whose value can be `0` or `1`. The `P`/`V` operations are modelled through a matrix of channels. The first index `s` specifies the semaphore id. The second one carries the id of the requesting process. The model in Fig. 1 makes a non-deterministic selection of the requesting process at a `P` or `V` synchronization. A not chosen process rests blocked on the requesting `P!` or `V!` operation.

Of course the model in Fig. 1 is unfair: at the time of a `V`, if more processes are blocked on the `P!` synchronization on the same semaphore, one of them is chosen not-deterministically to proceed. It is also possible for the `V`-executor process to compete and reacquire immediately the semaphore. Therefore, every process can experiment an unbounded waiting. It is known that to turn an unfair semaphore into a fair one a queue can be added to the semaphore where processes which find the semaphore red (0) at the time of a `P` are stored in first-in-first-out order and then awaken in the same order at a subsequent `V`.

A challenging research goal in the literature is to achieve a fair semaphore using only a minimum number of unfair semaphores. In the case when the queue is replaced by a set, the semaphore becomes unfair and it is often referred to as a *buffered* [4] or *blocked-set* semaphore [12]. A third version of weak semaphore is the so called *polite* semaphore proposed in [4]. A *polite* semaphore is similar to a *plain* semaphore but forbids the process which executes a `V` operation to reacquire immediately the semaphore.

The three types of weak semaphores can be modelled in UPPAAL as shown in the Fig. from 2 to 4 where for generality a `P` operation is supposed to be immediately followed by a `GO` synchronization to unblock the `P`-requester process. A `V` operation, being not blocking, never requires to be followed by a `GO` synchronization.

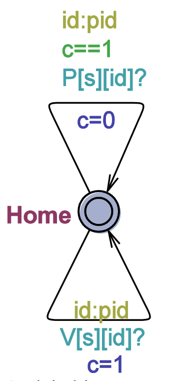

Fig. 1. A plain binary semaphore automaton

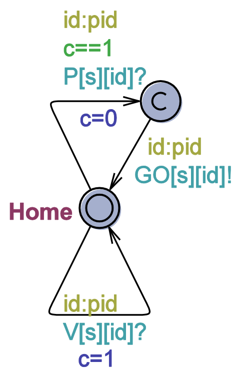

Fig. 2. Adopted `PlainBinary-Semaphore` automaton

Models in the Fig. 2 to 4 represent formal definitions of basic weak semaphores. Their correctness can be checked as follows. In the *polite* model in Fig. 3, the local variable `last` stores the id of the `V`-executor. The default value of `last` is `NONE`. At the time of a `P`, if the semaphore is green (`1`) and the process id is different from `last`, atomically the semaphore is turned to red (`0`) and the process (held in the `this` local variable) immediately receives the `GO` signal. Would the semaphore be red, or its id coincides with the `last` value, the process partially executes the `P` operation by incrementing the (local) counter `cnt` of blocked processes. A blocked process can be awaken by a `GO` synchronization raised in the right edge of Fig. 3, which completes its `P` operation by turning red the semaphore, decrementing the `cnt` variable and by assigning `NONE` to `last`. A subsequent `V` operation can (non-deterministically, as for a *plain* semaphore) unblock a waiting process, provided its id is different from `last`, or permit to a newly arrived process different from `last` to acquire the semaphore through a `P` operation.

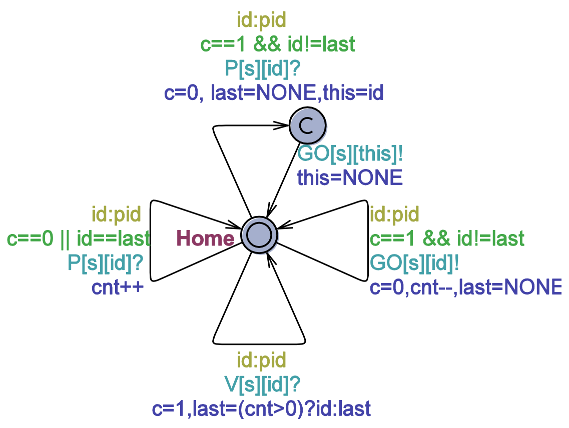

Fig. 3. `PoliteBinarySemaphore` automaton

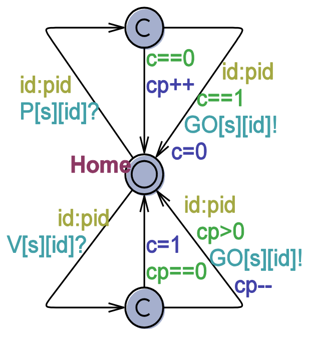

Fig. 4 `BufferedBinarySemaphore` automaton

If `s` is a *polite* binary semaphore, the following invariants hold:

1) `A[] s.last!=NONE imply s.cnt>0`
2) `A[] s.last==NONE || s.cnt>0`

Due to the invariant 1) it was omitted in the right edge with a `GO` synchronization of Fig. 3 the test `cnt>0` in the edge guard.

In the *buffered* semaphore model in Fig. 4 the buffer was purposely achieved implicitly in UPPAAL by the waiting locations of the set of processes which have just executed the `P!` operation and found red the semaphore. The number of such blocked processes is stored in the variable cp. At the time of a `V!`, if there are blocked processes, the semaphore is not turned to 1 and one of such processes is chosen non deterministically and receives the synchronization on the corresponding `GO` channel. It should be noted that both *polite* and *buffered* models exclude the `V`-executor to reacquire immediately the semaphore. But the *buffered* semaphore is stronger than the *polite* because at the time of a `V` operation, only one already blocked process can receive the semaphore pass. Fig. 5 summarizes global declarations of an UPPAAL model which makes use of any semaphore model in the Fig. from 2 to 4.

```
const int N=…;//number of processes
const int SEM=…;//number of weak semaphores
const int NONE=-1;

typedef int[0,N-1] pid;
typedef int[0,SEM-1] sid;

//semaphore IDs
…
//semaphore channels
urgent chan P[sid][pid];
urgent chan GO[sid][pid];
urgent chan V[sid][pid];
```

Fig. 5 Common global declarations for weak semaphores

Since a location without a clock invariant can disrupt liveness of an UPPAAL model being possible to remain in the location an arbitrary (potentially infinite) time, semaphore channels were declared as urgent. This way, without hurting model non-determinism, when a given operation is enabled it will be allowed to occur without time passage. This measure was adopted as a way to realize in UPPAAL the *finite delay property* [12] or *weak fairness* [4] of processes in a concurrent model, which requires a continuously enabled action eventually happens.

As a final remark, all the models in the Fig. from 2 to 4 can be implemented in a concurrent programming language (e.g., Java) using busy-waiting.

## IV. MODEL CHECKING STARVATION-FREE MUTUAL EXCLUSION ALGORITHMS BASED ON WEAK SEMAPHORES

If `S` is a fair binary semaphore initialized to 1, the usual pattern for achieving mutual exclusion among `N` (>2) competing processes accessing some shared data is the following:

**process**(p)=**loop** NCS; P(S); CS; V(S); **endloop**.

A problem which has received the attention of many researchers consists in the possibility of building a sound fair semaphore using only a minimal number of weak semaphores. Starting from late seventies some starvation-free mutual exclusion algorithms were proposed, though without an adequate proof of their correctness. Recently, in [4] a very interesting proof system based on the PVS theorem prover was defined and used to establish the correctness of three fundamental algorithms proposed in [5]-[7].

This paper claims that the approach and the results described in [4] are still unsatisfactory and that some properties exist to be discovered about those and other algorithms. A fundamental step in [4] was the identification of an abstract algorithm which facilitates the interpretation and analysis of the three mentioned algorithms.

The abstract algorithm is founded on the elevator metaphor: "While there are interested processes they enter the elevator at the first floor. When there are no processes arriving anymore, the elevator goes to the second floor and lets its occupants into CS, one by one. When the elevator is empty, it goes down again. When the elevator is not at the first floor, arriving processes have to wait. After CS, the processes go down by stairs."

The abstract algorithm uses 4 integer variables: `ne`, `nm`, `se` and `sm`. The first two variables model respectively the number of processes at the first floor waiting for the elevator, and the number of occupants within the elevator. The last two variables model respectively the doors at the first and second floor. Initially all variables are set to `0` except for the `se` which is set to `1`. The abstract algorithm is reproduced in Fig. 6 where atomic actions are enclosed within <…>.

```
process(p)=
loop
    NCS;
    <ne++>
    <await se greater-than 0; nm++; ne--;
      if ne==0 then sm++; se--; endif;>
    <await sm greater-than 0; sm--; nm--;>
    CS;
    <if nm greater-than 0 then sm++; else se++; endif;>
endloop.
```

Fig. 6 Mutual exclusion abstract algorithm

Correctness of the abstract algorithm can be verified in UPPAAL by deriving a corresponding native model like that

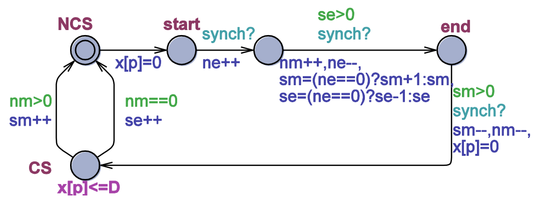

Fig. 7 UPPAAL Process automaton

shown in Fig. 7. Such a model only depends on the concurrency model of UPPAAL and in particular on the atomic actions labelling the various edges. The model consists of two TA: the `Process(const pid p)` (see Fig. 7) and the `Synch`(ronizer (see Fig. 8). The `Process` automaton embodies the mutex algorithm and is instantiated `N` (e.g., `N=4`) times.

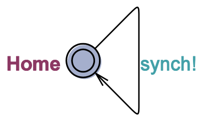

Fig. 8 The `Synch` automaton

All these instances share the algorithm variables, declared globally. Only one instance instead exists for the synchronizer. Each process instance `p` uses a clock `x[p]` to measure the waiting time before entering the critical section and the duration of the critical section.

The synchronizer is always ready to send a signal over the urgent unicast channel `synch`. Such a signal is a key to ensure progress to the model in Fig. 7 where some normal locations without clock invariants like `start` and `end`, are used.

Also the `NCS` location is without clock invariant, to mirror the fact that the non-critical section lasts an arbitrary number (also `0`) of time units.

The entry protocol of the mutex algorithm is played from the `start` to the `end` location. The exit protocol is coded on the arcs outgoing the `CS` location.

The following queries were used for property checking of the abstract algorithm.

```
1)A[] !deadlock                         satisfied

2)A[] forall(i:pid) forall(j:pid)
 Process(i).CS && Process(j).CS
 imply j==i                             satisfied

3)Process(0).start --> Process(0).CS
                                    not satisfied

4)A[] forall(i:pid) Process(i).end
 imply x[i]<=2*(N-1)*D                  satisfied

5)A[] forall(i:pid) Process(i).end
 imply x[i]<=2*(N-1)*D-1            not satisfied
```

Queries 1) and 2) check *safety properties*. Query 3) verifies a *liveness property*. Queries 4) and 5) check a *bounded liveness property*.

Satisfaction of query 1) guarantees the model has no deadlock (predefined keyword in UPPAAL). Query 2) ensures only one process at a time can be in the critical section. Query 3) checks if any process which finds itself in the `start` location eventually reaches the `CS` (critical section) location. Noteworthy, this property is not satisfied. Queries 4) and 5) check about the waiting time of each process before entering `CS` (whose duration is supposed to be at most `D` time units).

Note that clock `x[p]` is reset on entering `start` and on exiting `end`. It is confirmed that every process `p` has an *overtaking factor* of 2, i.e., its blocking time is determined by all the other processes which enter two times their `CS` before `p` can enter its `CS`.

Absence of liveness (query 3) is a direct consequence of the fact that the model has a *zeno-cycle*, i.e., it is possible for any process to (re)enter the `CS` an infinite number of times before any other process can enter its `CS`, by consuming `0` time. The zeno-cycle mirrors the fact that the critical section as well as the non-critical section can have a `0` duration, and that nothing forbids (non-determinism) the same process to always get the `synch` signals.

The zeno-cycle can be eliminated by guaranteeing the critical section necessarily consumes a finite (although very small) duration (the guard `x[p]>0` can be added to both edges exiting from the `CS` location). However, the existence of the zeno-cycle does not prevent the model checker to determine the worst-case waiting time of processes, in which case UPPAAL considers scenarios (behaviors) on the state graph where time is really advancing.

It should be noted that the presence of a zeno-cycle naturally expresses an intrinsic feature of the algorithm/model design. A different design can be without any zeno-cycle, independently from any consideration about timing.

The following invariants also hold for the model of Fig. 7:

```
A[] se==1 imply sm==0,
A[] sm==1 imply se==0,
```

which express functionality concerns of the abstract algorithm.

In [4] it is shown as some classic starvation-free mutual exclusion algorithms based on weak semaphores can be regarded as different interpretations of the abstract algorithm, where atomic operations are achieved by using a few unfair binary semaphores.

For brevity, in the following only the Morris algorithm [5] will be detailed. For the other studied algorithms, though, the experimental analysis will be synthetically reported.

## A. Morris Algorithm

Fig. 9 recapitulates the Morris algorithm which can be viewed as a concrete instance of the abstract algorithm of Fig. 6. The Morris algorithm uses three weak semaphores: `sb`, protecting specifically the `ne` counter holding the number of processes awaiting the elevator at the first floor, `se` and `sm` respectively controlling the door at the first and the second floor. They act as a split binary semaphore. Initially, `sb` and `se` are set to `1`, `sm` to `0`.

The initial values of the other variables are as in the abstract algorithm.

```
process(p)=
  loop
    NCS;
    P(sb); ne++; V(sb);
    P(se); nm++; P(sb); ne--;
    if ne>0 then V(sb); V(se);
    else V(sb); V(sm); endif;
    P(sm); nm--; CS;
    if nm>0 then V(sm); else V(se); endif;
  endloop.
```

Fig. 9 The Morris mutual exclusion algorithm

In Fig. 10 it is depicted an UPPAAL `Process` automaton corresponding to the algorithm in Fig. 9.

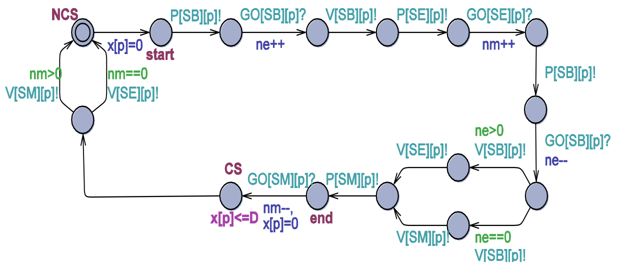

Fig. 10 UPPAAL Process model corresponding to the Morris algorithm of Fig. 9

In this case, the use of urgent semaphore channels avoids the recourse to other channels like `synch` of Fig. 8 in order to guarantee model progress. The model in Fig. 10 was model checked using the same queries 1) to 5) previously discussed for the abstract algorithm, and using for the `se` and `sm` semaphores the `PlainBinarySemaphore` template (Fig. 2) and separately checking the model behavior when the `sb` semaphore is implemented respectively as a *plain*, *polite*, or *buffered* binary semaphore, i.e., passing from the weakest to the strongest unfair semaphore.

In [4] a proof system was built to demonstrate that the Morris algorithm is correct, i.e., it is without deadlock, it ensures mutual exclusion and it guarantees a bounded waiting time (with an overtaking factor of `2`) for the blocked processes, for the sole case `sb`-*buffered* semaphore, `se`, `sm`-*plain* semaphores. From our analysis based on model checking it emerged that the Morris algorithm, even with `sb` being a *buffered* semaphore, always has a zeno-cycle which means, under the hypothesis of zero time duration of any action in the algorithm, that the overtaking factor for a blocked process is unbounded. Only when the critical section is supposed to consume even a very small time duration, the zeno-cycle disappears. Moreover, in the presence of timing of the critical section, the overtaking factor is effectively `2` as for the abstract algorithm but for any implementation of the `sb` semaphore. In other words, model checking the Morris algorithm, in the presence of timing, confirmed that the algorithm is correct with three *plain* binary semaphores, contrary to what is stated in [4] and [12].

The study of the Morris algorithm suggested to us the design of a simple variation of the algorithm based on two *plain* semaphores (the `se` and `sm` semaphores of the Morris algorithm), `N` bits and the `nm` counter. The algorithm is proposed in Fig. 11 and modelled in UPPAAL as in Fig. 12. It avoids the `sb` semaphore and uses instead an array `e` of `N` booleans, each element being associated to a distinct process.

```
process(p)=
  loop
    NCS;
    e[p]=true;
    P(se); nm++; e[p]=false;
    if ne() then V(se);
    else V(sm); endif;
    P(sm); nm--; CS;
    if nm>0 then V(sm); else V(se); endif;
  endloop.
```

Fig. 11 Proposed variation of the Morris algorithm

The array `e` replaces the `ne` counter of the abstract algorithm. Each process `p` sets `e[p]` to true when it starts waiting for the elevator at the first floor, and resets it to false when it enters the elevator, at which time the `nm` counter is incremented. Since each process manages its own element in the array `e`, no interference can ever occur on `e`. The test about the existence of other processes which want to enter the elevator at the first floor, previously based on the counter `ne`, it is now based on checking if there are some true elements in the array `e` (the check is actually delegated to a function `ne()` which returns true if some element in the array is true, false otherwise). Of course, a true value in `e` can be found in the current test or it will be sensed the next time.

Model checking the model in Fig. 12 confirmed that all the five queries proposed for the abstract algorithm are now satisfied. Also the liveness property (query 3) now holds, i.e., the new algorithm is without any zeno-cycle, and correctly operates even when timing is ignored.

## *B. Algorithms Comparison*

During the development of the modelling and verification approach described in this paper, besides the Morris algorithm, other starvation-free mutual exclusion algorithms based on weak semaphores were studied. Model checking results summarized in the Table 1 confirm known results in the literature and in some cases are more detailed. In the column of the semaphore types, the weakest admissible types for the algorithm are shown. More stronger versions could, but unnecessarily, be used. For instance, the `sb` (as in the Morris algorithm) semaphore of the Udding algorithm must be *buffered*. The other two semaphores can be *plain*. The Udding algorithm is no longer starvation-free if `sb` is implemented with a *polite* semaphore.

Analysis results concerning the Martin & Burch algorithm [6] coincide with those formally identified in [4]. The Haldar & Subramanian algorithm which relies on two semaphores and `2` bits [8] was also investigated in [13]. The weak semaphore type the authors assumed corresponds to a *buffered* one and the overtaking factor was indicated as

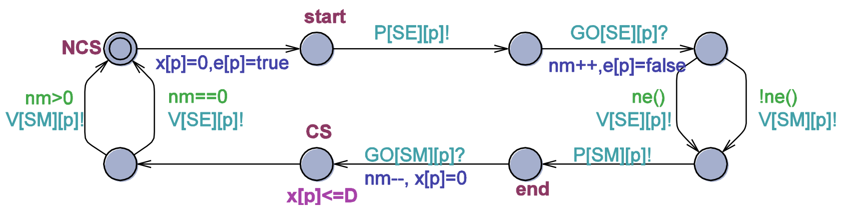

Fig. 12 Variation of the Morris algorithm based on two plain semaphores, N bits and the sole `nm` counter

TABLE I.
MODEL CHECKING RESULTS OF MUTUAL EXCLUSION ALGORITHMS.

| Algorithm | No of weak semaphores | Semaphore types | Zeno-cycle | Overtaking factor |
|---|---|---|---|---|
| Morris | 3 | 3 plain | yes | 2 |
| Morris variation proposed in this paper | 2 | 2 plain | no | 2 |
| Udding | 3 | 1 buffered – 2 plain | yes | 2 |
| Martin & Burch | 2 | 1 polite – 1 plain | yes | 2 |
| Haldar & Subramanian | 2 | 2 polite | yes | lesser than 2 |

being 2. However, the model checking approach developed in this work has shown that two *polite* semaphores suffice and that the waiting time of a process interested in entering its critical section is exactly 2*(N-1)*D-D, i.e., one critical section lesser than the other algorithms.

As it emerges from Table 1, the Morris variation algorithm proposed in this paper outperforms classic known algorithms. With respect to the Haldar & Subramanian algorithm, our algorithm uses only 2 binary semaphores of the weakest type (*plain*) although it uses some more memory (N bits plus the nm counter vs. 2 bits of the Haldar & Subramanian algorithm). Moreover, the proposed algorithm is the only one which is without zeno-cycles.

## V. Conclusions

The Dijkstra conjecture [9] about the impossibility of building a fair semaphore using a few weak semaphores was confuted by the development of algorithms proposed by Morris [5], Martin & Burch [6], Udding [7] etc. However the correctness proof of such algorithms was only partially provided, also considering the methodological approach proposed in [12] or the proof framework developed in [4] which does not allow a full analysis of mutual exclusion algorithms in the presence of the timing dimension.

In this paper an original proving framework based on timed automata (TA) and the Uppaal toolbox is proposed which permits modelling and full verification of the properties of starvation-free mutual exclusion algorithms based on weak semaphores, also in the presence of the timing dimension.

The approach models the three known types of weak semaphores: *plain* (the weakest type), *polite* and *buffered* (the strongest type). It is worthy of note that in its description, the Dijkstra conjecture implicitly refers to the use of *buffered* semaphores.

A key factor of the proposed approach is its modelling and analysis flexibility, being it possible to transparently replace a semaphore type with another one thus enabling a thorough study of a given algorithm.

The application of the approach confirms known properties of classic algorithms, but has the potential to discover subtle features of the considered algorithms such as the existence of a zeno-cycle or of a time-sensitive behavior which influences the kind of weak semaphores which can be actually used. All known algorithms suffer of a zeno-cycle, in the light of which the overtaking factor of a waiting process is (theorically) unbounded. However, when the critical section consumes an even infinitesimal time, the bounded waiting time and overtaking factor of the classic algorithms is effectively guaranteed. In this hypothesis the Morris algorithm is correct with three *plain* semaphores.

As part of this work, a variation of the Morris algorithm was designed which intrinsically eliminates any zeno-cycle, rests only on two *plain* semaphores and replaces a counter of the Morris algorithm with N bits. This new algorithm too was proved to be correct.

The paper contribution enables the implementation of light-weight starvation-free semaphores which can be exploited in general concurrent systems including cyber physical systems.

Prosecution of the research is geared at:

- Modelling and analysis of other mutual exclusion algorithms designed in terms of weak semaphores.
- Implementing weak semaphores and fair semaphores corresponding to mutual exclusion algorithms, in a concurrent programming language, e.g., Java.
- Experimenting with the use of weak semaphores in practical systems programming and in the development of cyber physical systems.
- Exploiting light-weight starvation-free semaphores in distributed shared memory systems, e.g., based on Java and the Terracotta middleware [14] which provides the vision of a "network heap" where shared data can be accessed by threads belonging to distributed JVMs.

## References

[1] S. Srbljic, D. Skvorc, M. Popovic, "Programming languages for end-user personalization of Cyber-Physical Systems", *Automatika*, Vol. 53, No. 3, pp. 294-310, 2012.
[2] G. Behrmann, A. David, K.G. Larsen, "A tutorial on Uppaal", In: *Formal Methods for the Design of Real-Time Systems*, M. Bernardo and F. Corradini Eds., Lecture Notes in Computer Science, Vol. 3185, Springer-Verlag, pp. 200-236, 2004.
[3] E.M. Clarke, O. Grumberg, D.A. Peled, *Model checking*, MIT Press, 2000.
[4] W.H. Hesselink, M. IJbema, M. "Starvation-free mutual exclusion with semaphores", *Formal Aspects of Computing*, DOI 10.1007/s00165-011-0219-y, 2011.
[5] J.M. Morris, "A starvation-free solution to the mutual exclusion problem", *Inf. Proc. Lett.*, Vol. 8, pp. 76-80, 1979.
[6] A.J. Martin, J.R. Burch, "Fair mutual exclusion with unfair P and V operations", *Inf. Proc. Lett.*, Vol. 21, pp. 97-100, 1985.
[7] J.T. Udding, "Absence of individual starvation using weak semaphores", *Inf. Proc. Lett.*, Vol. 23, pp. 159-162, 1986.
[8] S. Haldar, D.K. Subramanian, "An efficient solution of the mutual exclusion problem using unfair and weak semaphores", *ACM SIGOPS Operating Systems Review*, Vol. 22, pp. 60-66, 1988.
[9] E.W. Dijkstra, "A strong P/V-implementation of conditional critical regions", Tech. Rep., Tech. Univ. Eindhoven, EWD 651, www.cs.utexas.edu/users/EWD, 1977.
[10] R. Alur, D.L. Dill, "A theory of timed automata", *Theoretical Computer Science*, Vol. 126, pp. 183-235, 1994.
[11] F. Cicirelli, A. Furfaro, L. Nigro, "Model checking time-dependent system specifications using time stream Petri nets and Uppaal", *Appl. Math. Comp.*, Vol. 218, pp. 8160-8186, 2012.
[12] E.W. Stark, "Semaphore primitives and starvation-free mutual exclusion", *J. of ACM*, Vol. 29, pp. 1049-1072, 1982.
[13] H.P. Hofstee, K.R. Leino, L.A. van de Snepscheut, "Proof of mutual exclusion algorithm by Haldar and Subramanian", HPH11-0, California Institute of Technology, 18 December 1991.
[14] F. Cicirelli, A. Furfaro, A. Giordano, L. Nigro, "Performance of a multi-agent system over a multi-core cluster managed by Terracotta", In *Proc. of Symp. on Theory of Modeling & Simulation: DEVS Integrative M&S Symp.*, pp. 125-133, 2011.