https://www.aimspress.com/journal/aci

Applied Computing and Intelligence
5(2): 236–263.
DOI: 10.3934/aci.2025014
Received: 12 July 2025
Revised: 26 September 2025
Accepted: 27 September 2025
Published:16 October 2025

Research article

Formal verification of mutual exclusion algorithms in anonymous memory

Libero Nigro[1,*] **and Franco Cicirelli**[2]

[1] University of Calabria, DIMES, 87036 Rende, Italy
[2] CNR-National Research Council of Italy-Institute for High Performance Computing and Networking (ICAR)-87036 Rende, Italy

* **Correspondence:** Email: libero.nigro@unical.it; Tel: +393392396819.

Academic Editor: Pasi Fränti

Abstract: This paper applied a formal method based on timed automata and the Uppaal toolbox to the specification and verification of mutual exclusion algorithms operating on anonymous memory. Anonymous memory is a challenging variation of shared memory where common variables (registers) do not have globally agreed-upon names. Instead, each process has its own view and naming for the shared registers. In addition, the cooperating/competing processes are supposed to be symmetric, that is, they execute the same code, and process identifiers are unknown. A recent proposal of Taubenfeld's mutual exclusion algorithm with $m \geq 7$ (m odd) shared anonymous registers was chosen as a concrete case study. Such an algorithm was informally studied by the authors together with the establishment of some possibility and impossibility results. This paper's first contribution was to confirm the correctness of the algorithm for $N = 2$ processes when atomic registers are used. The solution no longer holds for a smaller number of registers, e.g., $m = 5$, or when non-atomic registers are used. The paper also showed that for $N > 2$, the algorithm violates the mutual exclusion. To extend the use of the algorithm to $N > 2$ processes, this paper developed a novel organization where the basic solution for two processes is used as the arbitration unit in a binary, standard state-of-the-art tournament tree. Some timed variations of the model were considered, which permit the study of the algorithm under model checking and/or statistical model checking, for various values of the number N of processes.

Keywords: concurrency; anonymous shared memory; mutual exclusion; atomic/non-atomic registers; timed automata; Uppaal; model checking; statistical model checking

1. Introduction

The work described in this paper develops a methodology for formal modeling and verification of mutual exclusion algorithms [1–4]. The context is that of shared memory concurrent/parallel and distributed systems. Mutual exclusion is the well-known fundamental problem of concurrent programming [2,5], whose solution has to guarantee that multiple executing processes that share a common data resource (e.g., a file in the context of an operating system) are constrained to coordinate each other, according to a suitable protocol, to prevent the resource from being accessed simultaneously by multiple processes. Concurrent access and modification of the resource would impair its correct use and state evolution (think about a bank account balance with unconstrained deposit and withdrawal operations). The instructions executed by a process for accessing the shared resource constitute its *critical section*. The mutual exclusion protocol controls the critical section executions. Starting from the basic solution for two processes proposed by Dekker [6–8], several algorithms for $N = 2$ and $N > 2$ processes have been proposed (see, e.g., [4,9–12]), each distinguished from one another based on the simplicity, elegance, and sharpness of the protocol design, which is based on, hopefully few, *shared communication variables* (registers). Besides the often ingenious protocol design, to be correct, a mutual exclusion algorithm has to fulfill the following main properties: (a) only one process at a time is allowed to use the resource (mutual exclusion property); (b) absence of deadlock (the protocol never implies that the processes enter a lockout situation); and (c) absence of starvation (a competing process eventually enters its critical section). An implicit further assumption of classical mutual exclusion algorithms is that processes are assumed to be *memoryless*: a process that tries to enter its critical section does not remember any information about its previous attempts.

Correctness of a mutual exclusion algorithm should be checked both when *atomic registers* are used [11,12] (read/write operations on the same memory register are mutually exclusive) and, more in general, when *non-atomic registers* [13–16] are used (multiple read/write operations on the same register can occur simultaneously). Non-atomic registers exist when concurrent/parallel processes are allowed to execute in low-cost hardware devices like phone-cells and similar [17], that is, without hardware control to deny simultaneous access to the registers of a multi-port memory [18].

As a significant variation of the classical shared memory model, the so-called *shared anonymous memory* [19,20] emerged in the last few years as a new challenging scenario for designing and studying mutual exclusion algorithms [21,22]. The fundamental difference from the classical shared model consists of adopting anonymous registers to control process competition and coordination. Anonymous registers mean that each process possesses its own names for the same set of shared registers. A register named *A* for a given process can correspond to what another process refers to as register *C*, and so forth. Stated in other terms, every process uses its *permutation* of the global available register names. In addition, processes are assumed to be *symmetric*, that is, they execute the same code and have unknown identifiers.

A representative mutual exclusion algorithm for $N = 2$ processes was recently proposed by Taubenfeld in [22]. The author referred to the use of $m \geq 7$ anonymous registers, with m odd, as a necessary condition for the correctness of the algorithm. In [22], the author presented some possibility and impossibility results. Among the possibility results, the case $N = 2$ was proved to be correct (including absence of deadlocks and starvation) for m not less than 7. An impossibility result concerns the fact that the algorithm should not be starvation-free for $N > 2$ processes. The case $N > 2$ still appears open and should be further investigated.

The anonymous memory scenario has a twofold motivation. First of all, as stated in [22], it is of interest from a computing theoretical point of view, that is, studying what can/cannot be done by concurrent processes under this new memory model. The second motivation is practical and concerns the development of bioinspired applications, particularly based on the ideas of molecular biology. In [23], in fact, it was observed that in the process of genome-wide epigenetic modifications, cells utilize their DNA as a shared anonymous memory system.

In the literature, the property assessment of mutual exclusion algorithms has often been based on informal mathematical reasoning and the use of atomic registers only. For example, the algorithm in [22] was studied this way. However, it is widely recognized that intuitive reasoning can be unable, except in simple cases, to detect all the properties of a concurrent system due to the intrinsic difficulty of completely predicting the non-deterministic order of actions (*interleaving*) that characterizes the process executions. Formal modeling and exhaustive verification are currently recognized as an important contribution to the correctness assessment of a mutual exclusion solution. Two significant formal methods have proven to be useful in many practical cases: the assertional method (e.g., [24]) and the model checking method (e.g., [25,26]). The assertional method builds a transition system from a mathematical specification of the mutual exclusion solution. Assertions are, then, formulated and systematically analyzed with the services of a theorem prover. Model checking methods rely on a formal model of the mutual exclusion algorithm, e.g., based on the use of timed automata [27,28] or process algebra [15,26]. A (hopefully finite) state graph of the model can then be derived, containing all of the possible execution states and the execution paths admitted in the model. The state graph can, finally, be systematically consulted with the help of queries expressed in a temporal logic language (e.g., the Temporal Computational Tree Logic -TCTL- in [28]). The assertional method can be hard to use in complex mutual exclusion algorithms. On the other hand, model checking can be limited by an infinite state graph generated by state explosions [29] when the model relies on many data variables or it has a high degree of non-determinism (partial-order or number of possible exit transitions in the state graph nodes) in the navigation of the execution paths.

This paper is based on timed automata and the Uppaal model checker [28,30–33]. The approach is motivated by the fact that graphical models of time-sensitive process automata can be more intuitive to understand than, e.g., process algebra descriptions as in mCRL2 [26]. In addition, the Uppaal toolbox, continually improved, is characterized by a state graph memory representation that uses compact data structures and efficient navigation algorithms, as well as an easy-to-exploit TCTL subset of queries for property checking.

The possible use of the time dimension in a Uppaal model deserves further discussion. It has been observed that the correctness of certain mutual exclusion algorithms (see also later in this paper) may depend on subtle time assumptions that are unexpected from the intuitive description of the algorithm. On the other hand, theorem provers may be unable to model or reason about timing issues.

Finally, the Uppaal toolbox was also chosen for its support of a statistical model checker (SMC) [34–36]. The SMC does not create the model state graph, requires a reduced amount of memory footprint, and uses stochastic simulations to estimate model properties. Although simulation estimates cannot replace the precise, exhaustive model checking results, they can be helpful to observe properties of a mutual exclusion model for non-trivial values of the number N of processes.

This paper is devoted to the formal investigation of Taubenfeld's mutual exclusion algorithm [20] based on anonymous memory, using a transformation of the algorithm in Uppaal models. A refinement of the formal modeling and verifying methodology based on Uppaal, introduced in [11,12], is applied to Taubenfeld's algorithm. The paper first shows that the algorithm is time-sensitive. It is required that a process exiting the critical section should not immediately start a new competition. Model checking the case $N = 2$ confirms that the model satisfies all the correctness properties with $m = 7$ registers, as informally assessed in [22]. For a smaller number of registers, e.g., $m = 5$, although ensuring that the model continues to be deadlock-free, it violates the mutual exclusion. For $N = 3$, model checking proves that the algorithm misses the fundamental property of mutual exclusion, regardless of the starvation-free requirement. The paper continues by showing that Taubenfeld's algorithm, even for $N = 2$, is incorrect when anonymous and non-atomic registers are used. As a further original contribution of this paper, Taubenfeld's algorithm for $N = 2$ is then used as the *arbitration unit* in a standard, state-of-the-art tournament tree (TT) [12,37,38] structure. This organization makes the basic algorithm correct for $N \geq 2$ processes, using $m = 7$ anonymous registers at each arbitration stage of the TT. This new result seems to overcome the impossibility result stated in [22] about the non-existence of a symmetric solution for $N \geq 3$, whatever the number m of employed registers. The TT model is thoroughly checked for different values of N by adapting it with time to be usable by both the model checker (MC) and the statistical model checker (SMC) of Uppaal.

The paper is structured as follows: Section 2 describes the operation of Taubenfeld's mutual exclusion algorithm, and its possibility and impossibility results. Section 3 summarizes the modeling and verification features of Uppaal. The model checker, its TCTL queries, and the statistical model checker with some supported Metric Interval Temporal Language (MITL) queries are highlighted. Section 4 proposes and applies some reduction rules for transforming Taubenfeld's algorithm into Uppaal models based on timed automata. The basic model is separately verified using atomic and non-atomic registers. Then, a timed version model is introduced, which lends itself to being investigated by exhaustive model checking and stochastic simulations. Section 5 proposes and verifies the tournament binary tree–based solution that generalizes the basic algorithm to $N \geq 2$ processes. Finally, conclusions are presented together with some indications of further work.

2. Taubenfeld's mutual exclusion algorithm

In the following, concurrent processes participating in a mutual exclusion problem regulated by a suitable protocol are assumed to be non-terminating, fault-free, and executing, e.g., on distinct cores of a multiprocessor architecture. After exiting its critical section, a process re-enters, thus starting a new competition. The overall structure of a process is shown in Algorithm 1.

Algorithm 1. Structure of a concurrent process involved in a mutual exclusion problem.

```
shared communication variables //with proper initialization
process(i) //i is the process identifier
    local variables
    loop
    NCS // Non Critical Section
    Entry part of the protocol
    CS // Critical Section
    Exit part of the protocol
    end-loop
end-process
```

In the NCS section code, the process is not interested in entering the critical section. In the Entry part of the protocol, the process competes with its peers to gain permission to enter the critical section. Toward this, busy-waiting or spinlock loops can be executed. During a busy waiting period, the process actively checks, although by wasting core cycles, shared variables until a condition is detected that permits it to go further. In the Exit part of the protocol, the process can be required to reset global variables, possibly by busy-waiting, and local variables before reaching NCS.

Each particular mutual exclusion algorithm is characterized by the set of shared communication variables (registers). An ingenious design of a mutual exclusion solution uses a minimal number of shared registers. Some shared variables can be output or exterior [37] registers. An output register is written by only one process (its owner) but can be read/consulted by all the remaining processes, possibly concurrently. In the more general and challenging case, non-output shared registers can be read/written by any process, also simultaneously.

In a memoryless algorithm, neither the Entry nor the Exit part depends on saved information from previous process attempts.

Taubenfeld's algorithm [22] rests on m shared and anonymous registers $reg[]$, e.g., selected by the indexes from 1 to m. All these registers are non-output variables, and the most general pattern of Multiple Writers Multiple Readers (MWMR) applies, because every register is allowed to be read/written concurrently. Neither a process knows the identifiers of partners, nor can it make any assumption about the state of a partner. In this paper, it will be assumed that processes have the identifiers from 1 to N, and all of them execute the same code (*symmetric* processes). The values of an anonymous register are the integers in the range $[0..\mathrm{W}]$ where $\mathrm{W} = \mathrm{N} + 1$. Value 0 denotes a free register. Value W indicates there is a waiting process. Values from 1 to N mirror that the register is currently occupied by the corresponding process identifier.

Each process i accesses shared registers by using a (hidden) permutation $\mathrm{perm_i}[]$ of the indexes $[1..\mathrm{m}]$, unknown to the other processes. This way, accessing, e.g., the 0-th register by process i, will result in access to the real register selected by the index in the 0-th position of the permutation of process i. To simplify the notation, the operations, which read/write the j-th register as viewed by a given process i, will be expressed by $\mathrm{r(j)}$ and $\mathrm{w(j,v)}$, where $\mathrm{r(j)} \equiv \mathrm{reg[perm_i[j]]}$ and $\mathrm{w(j,v)} \equiv \mathrm{reg}\big[\mathrm{perm_i[j]}\big] = \mathrm{v}$. It is evident that what a process considers as the j-th register can be completely different from the j-th register viewed by a different process.

The pseudo-code of Taubenfeld's algorithm executed by a generic process i is presented in Algorithm 2, adapted from [22]. For simplicity, the index permutation of the process i over global registers is not shown. Recall that all the processes are symmetric and execute the same code.

In Algorithm 2, busy-waiting loops are expressed using repeat-until structures (see line 1, lines from 6 to 8, 24 to 26, and 10 to 29). For example, in line 1, for generality, busy-waiting is supposed to be continued until a shared register, indexed by the local variable *mycounter*, is found with a value of 0. Implicitly, *mycounter* is reset to 0 at each repeat-until termination without finding a register with a 0 value.

Some intuitive notes about the behavior of Algorithm 2 are as follows. At each start of competition (beginning of the Entry part of the protocol, see Algorithm 1) process i (see lines from 1 to 9 in Algorithm 2) gives priority to waiting processes. First, it waits for a register (indexed by the local *mycounter* variable) with the value 0. If one is found, that register is tentatively occupied with the process identifier. However, if at least one other process exists that is waiting (a register with the W value), process i releases its commitment (line 5) and waits until no process exists with the waiting status. Due to the frequent checks of the global registers, each process keeps a local copy of the global registers in the *myview*[] array updated. For non-determinism, though, no guarantee is provided to the process by the content of *myview*[]. Of course, on arriving at line 10, at most one occurrence of the process identifier i can be present in the global registers.

Algorithm 2. Pseudo-code of Taubenfeld's mutual exclusion algorithm.

```
global constants:
    int m; // number of shared anonymous registers. Assumed $m \geq 7$, m odd
    int N; // number of processes, indexed from 1 to N
    int $W = N + 1$ // waiting status
global variables:
    int reg[1..m]; // all initialized to 0; admitted values are the integers from 0 to W
process(i) // i in [1..N]
  local variables:
    int myview[1..m]; // all initialized to 0
    int j, k, mycounter; // all initialized to 0
    boolean mygo; // initialized to false
  loop
0) NCS;
1) repeat mycounter=mycounter+1; until r(mycounter)==0;
2) w(mycounter,i);
3) for j=1 to m do myview[j]=r(j); end-for
4) if $\exists$ j in [1..m] : myview[j] == W then
5)      if r(mycounter)==i then w(mycounter,0); end-if
6)      repeat
7)          for j=1 to m do myview[j]=r(j); end-for
8)      until $\forall$ j in [1..m] myview[j]$\neq$W;
9) end-if
10)     repeat
11)         for k=1 to m do
12)             if r(k)==0 then
13)                 for j=1 to m do myview[j]=r(j); end-for
14)                 if i appears in less than m-2 entries of myview[1..m] then
15)                     w(k,i); end-if end-if
16)         end-for
17)         for j=1 to m do myview[j]=r(j); end-for
18)         if i appears in less than $\lceil m/2 \rceil$ entries of myview[1..m] then
19)             mycounter=0;
20)             for j=1 to m do if r(j)==i then
21)                 if mycounter==2 then w(j,0);
22)                 else w(j,W); mycounter=mycounter+1; end-if end-if
23)             end-for
24)             repeat
25)                 for j=1 to m do myview[j]=r(j); end-for
26)             until $\forall$ j in [1..m] : myview[j] in {0,W};
27)             mygo=true;
28)         end-if
29)     until i appears in m-2 entries of myview[1..m] or mygo==true;
30)     CS;
31)     if mygo==true then for j=1 to m do if r(j)==W then w(j,0); end-if end-for
32)     else for j=1 to m do if r(j)==i then w(j,0); end-if end-for
33)     end-if
34)     set all local variables to their initial values
  end-loop
end-process
```

The core of the algorithm is expressed in the repeat-until loop from line 10 to line 29. Briefly, a process is a winner and gets allowed to enter its critical section when it succeeds in occupying *m*-2 entries of the registers (as perceived by *myview*[]). The occupation attempt is carried out in the lines from 11 to 16, where free registers (which have the value 0) are set with the process identifier. If the attempt fails, process i releases its previously occupied entries except for two. In particular, the first two occupied registers are changed to the W value (i.e., the waiting status), and the remaining entries are set to 0. Despite this release, the process can still be a winner in the case that all the register entries contain either 0 or the W value, with the Boolean *mygo* variable that is set to true. A subtle point in Algorithm 2 is the fact that the occupy attempt is considered to have failed when the number of achieved entries is less than $\lceil m/2 \rceil$ (i.e., 4 when $m = 7$) and not $m - 2$ (i.e., 5 with $m = 7$). Using the latter test, two processes p and q, which occupied three entries and four entries, respectively, would both be considered losers and forced to release the occupied entries. In this case, for non-determinism, both processes could become winners because registers are found to have only the values 0 or W (with their *mygo* variables, which are set to true). Thus, both processes would enter their critical section simultaneously and violate the mutual exclusion property.

The Exit part of the protocol in Algorithm 2 (lines from 31 to 33) resets the registers according to the motivation that the process was a winner. In the case that *mygo* is found to be true, all the W entries are turned to 0. Otherwise, ($m - 2$ entries were occupied by the process) all the entries with the process id are changed to 0. Finally, the process resets all the local variables to their initial values.

Of course, from the preceding intuitive description, it is hard to fully infer that the algorithm is correct, in the sense that it always fulfills the mutual exclusion property, and it is free of both deadlock and starvation. What is difficult to assess are the subtle effects of non-determinism.

Correctness of Algorithm 2 was valuably discussed, by informal mathematical reasoning, by the author in [22], also by investigating the critical value of the minimal number m of the required registers, which should guarantee algorithm correctness. In Algorithm 2, $m \geq 7$ is assumed, with m odd. This paper aims to provide correctness arguments about all the expected properties through formal analysis and exhaustive verification (see Section 4). The analysis of the algorithm for larger values of the number N of processes can be assisted by statistical model checking [34,35] techniques.

2.1. Possibility and impossibility results

The following possibility and impossibility results were stated in [22] about Algorithm 2:
1) A symmetric memoryless deadlock-free and starvation-free mutual exclusion solution exists for $N = 2$ processes, provided a number $m \geq 7$ of anonymous registers are used, with m being an odd number.
2) There cannot exist a symmetric memoryless starvation-free mutual exclusion solution for $N \geq 3$ processes, whatever the number m of anonymous registers. In particular, for $N \geq 3$ processes, there is a separation between deadlock-free and starvation-free.

This paper will formally check these results and develop a possible workaround for the impossibility result.

3. Uppaal modeling and verification

This section first reviews the basic formal modeling and verification methodology based on Uppaal [11,12,16,28]. Then, the methodology is applied to reduce Taubenfeld's algorithm [22] (see Algorithm 2) in a Uppaal model, which is thoroughly analyzed.

3.1. Model specification

Uppaal is a popular toolbox for specifying and analyzing real-time systems (see, e.g., [36]) that supports a high-level modeling language based on timed automata (TA) [27]. Clock variables control time. A clock can be reset; then, it automatically advances to measure the relative time elapsed from its last reset. All the clocks of a model grow at the same rate. A system model consists of a network of concurrent interacting processes. Each (template) process is a parameterized timed automaton, built of *locations* and *edges*. Processes share a global data environment and can have local data declarations. Elementary data types are int, bool, clock, and channel. Structs and arrays of basic types are also admitted. C-like functions can be used, which can greatly contribute to model compactness and readability. Process interactions can be based on global data variables or can exploit the exchange of signals by channels, which carry no data. Channels can be unicast/rendezvous or broadcast. A rendezvous channel can synchronize a sender (which executes the **!** send operation) and a receiver (which executes the **?** receive operation). Only when both the sender and the receiver are ready does the communication (signal exchange) occur. Otherwise, the first process that arrives will wait for the partner to be ready. A broadcast channel has one sender and 0, 1, or multiple receivers. The sender of a broadcast channel never waits. Broadcast channels are useful for asynchronous communications.

A process automaton stays, at any moment, in a location (local state). What can change is the dwell time associated with the location. In a normal location (represented by a white circle), the automaton can remain for an arbitrary time, from 0 to infinity. To constrain the duration, a normal location can be equipped with an invariant, that is, a Boolean expression often based on a clock constraint. The automaton can remain in the location, provided the invariant remains true. Otherwise, the location has to be abandoned. There are also urgent (circle with an internal U) and committed (circle with an internal C) locations. Both have to be immediately exited, without time passage. Uppaal, though, gives precedence to committed locations, which are always exited before urgent locations.

Locations are linked to one another by arrowed edges. An edge comes with an (optional) guarded command, that is, a triple <guard, synchronization, update>. The guard is a Boolean expression, usually based on data and/or a clock constraint. A true valued guard denotes that the edge can be taken, although it is not forced to be immediately taken. An omitted guard evaluates as true by default. A synchronization expresses a channel send (**!**) or receive (**?**) operation. The update part of a command is an ordered list of variable assignments and clock resets. It is worth noting that the guarded command can be omitted. In this case, the spontaneous edge has no constraint on its execution. Any channel type can be declared to be urgent. In this case, if the guard of the command is true, the edge has to be taken without time passage.

3.2. Basic semantic issues

The Uppaal modeling language is both concurrent and time-sensitive. The evolution of a system model can be described by a Timed Transition System $\mathrm{TTS} = \{\mathrm{S}, \mathrm{S}_0, \rightarrow, \mathrm{now}\}$, where S is a set of states, S_0 is the initial state, $\rightarrow$ is the transition relation that moves the TTS from a state to the next one, and now is the current global time. A state logically consists of a data part and a time part. The data part stores the values of the model data variables and maintains a vector of locations, with one item per process storing its current location and a vector of clock valuations. The time part consists of an inequality system (firing domain or zone) that captures the clock inequalities that hold for this state to be entered.

Two kinds of transitions (atomic elementary events) can be distinguished: the action transition $\rightarrow_a$ and the delay transition $\rightarrow_d$. Action transitions are instantaneous. All the actions that can occur at a given time are concurrent. They are executed one at a time and in a non-deterministic order. Examples of action transitions are the guarded commands, in separate process automata, attached to the edges exiting from urgent locations, or involved in enabled edges (the guard is true) with an urgent synchronization, and so forth. All these events will occur non-deterministically.

When no more actions are enabled in the current state, a delay transition can occur that causes the global time of the model to be advanced by the minimal amount compliant with all the active location invariants. Such a minimal value Δ would cause no invariant to be falsified. Rather, the time increment would make one or more constrained locations become candidates to be exited, and one or multiple action transitions eligible for non-deterministic execution. Of course, all the clocks in the model will be incremented by Δ.

The number of transitions exiting from a state s represents its partial order degree. Model verification will consider, separately, what happens in the system evolution when each distinct transition is taken. In the case that the number of exiting transitions from a state of the TTS is 0, the state is a deadlocked one.

3.3. Model verification

The Uppaal model checker (MC or verifyta) [28,30] builds an in-memory (hopefully finite) compact representation (state graph) of the model TTS. The model state graph is then explored by efficient algorithms by responding to queries prepared according to a subset of the Timed Computational Tree Logic (TCTL) [28]. All the possible states and execution paths existing in the state graph are systematically investigated for proving/disproving a given property. TCTL queries can be based on the existential (E<>) or invariant (A[]) operator, applied to a given formula Φ built in terms of state predicates (the fact that a process occupies a given location) and/or data or clock constraints, combined with the usual Boolean operators, including the implication operator (imply). Basic examples of TCTL queries are the following (the state graph is normally explored starting from its initial state):

(a) $\mathrm{E} <> \Phi$
does at least one state exist in the state graph where Φ is satisfied?
(b) $\mathrm{A}[]\ \Phi$
is it always true that, invariantly, the Φ holds in every state of the model state graph?
(c) $\mathrm{E}[]\ \Phi$
does a state path exist where Φ holds in every state of the path?
(d) $\mathrm{A} <> \Phi$
is it always true that a final state can be reached where Φ holds?
(e) $\Phi \dashrightarrow \Psi$ (Φ $leads - to$ Ψ)
is it always true that starting from a state where Φ holds, it inevitably follows that a next state, also with a possible time constraint, will be reached where Ψ is satisfied?

Two abbreviations of the A[] query, practically useful, are the inf(ima/sup(rema queries. For instance:

(f) $\mathrm{sup}\{\ \Phi\ \}$: expression,
asks to find the maximum value of the expression (e.g., a single clock or a data variable) in all the states where Φ holds. It could equivalently be expressed by the repeated use of:

A[] Φ imply expression $\leq$expected-maximal-value, by tentatively changing the expected maximum value toward finding its lower bound. In a similar way, operates inf, which asks for finding the minimal value of the expression.

The default deadlock state predicate can be used for checking the existence of deadlocked states. Two equivalent and alternative queries can be used:

```
E<> deadlock
A[] ! deadlock
```

The analysis of the first, existential query, would terminate as soon as a deadlocked state is reached. The invariant version will check (and always generate) the full state graph to conclude that no deadlocked state exists.

An important feature of MC is its capability of generating a counterexample (also called a diagnostic trace), that is, a sequence of atomic events (state transitions) whose execution can demonstrate why a given property gets violated in the model, or it can be satisfied. The modeler can carefully inspect, step-by-step, the diagnostic trace in the so-called symbolic simulator [26,28], to detect a possible modeling bug to be fixed. This is important during the early stages of modeling, design, and debugging.

The critical aspect of MC is that the state graph can be practically enormous (even infinite) in its memory requirements, thus impeding, for state explosions [29], the verification activities of the model. Some provisions can, therefore, be of utmost importance in the attempt to avoid state explosions. First of all, the modeler should always try to reduce the number of data variables needed by the model. In addition, integer variable domains should purposely be reduced to the strict minimal range of values. Sometimes, even dropping a single redundant variable can improve model verification. A second cause of state explosion is directly related to the great amount of partial order in the states of the state graph. For example, when modeling a mutual exclusion algorithm, the existence of busy-waiting loops can easily generate a high degree of non-determinism by also often introducing zeno-cycles [39], that is, the creation of possibly infinite event paths, which are executed in a finite, e.g., zero time.

3.4. The statistical model checker

Nowadays, Uppaal also admits a statistical model checker (SMC) [34,35], together with the use of double variables, clocks (stopwatches)—which can be temporarily stopped (see, e.g., [36])—and a generalized version of the so-called hybrid/stochastic timed automata, where location invariants can include simple first-order differential equations. The SMC rests on simulation and probabilistic behavior for investigating model properties.

The SMC cannot replace the efficacy and precision of the conclusions achieved through the exhaustive verification of MC. However, it can be a practical help when the state graph suffers from state explosions. As an example (see also later in this paper), the Uppaal model of a mutual exclusion algorithm is normally impossible to scale in the number N of processes. The SMC can then be exploited to achieve estimations of the model properties for not-so-small values of N.

Uppaal SMC does not build the state graph. Then, its memory consumption is linear with the model size. Model explorations are then based on simulations. SMC naturally works with time-dependent models. A zeno-cycle could easily give rise to a time-lock error, where time does not advance starting from a given location. A recurrent use case of SMC is executing a given number of simulations (Monte Carlo–like simulations [34]), each lasting a certain time duration, and counting the number of simulations in which a given event occurs. All of this can estimate the occurrence probability of the event. The following are some useful Metric Interval Temporal Logic (MITL) queries [35]:

(a) simulate [<=timeEnd] {expression1 "[, expression2, …, expression]"}
launches a single simulation that lasts timeEnd time units and accumulates information about the values of the observed expressions. At the simulation end, by right-clicking the query, Uppaal SMC graphically displays the observed values of the expressions vs. time.
(b) Pr[<=timeEnd]("<>|[]" Φ)
asks to quantify the occurrence probability, with associated confidence interval (default confidence index of 95%), of the event: 1) <> Φ: "exists a state in which Φ holds", 2) []Φ: "it always happens that Φ holds".
(c) E[<=timeEnd;nr_of_simulations]("[min|max]:" expression)
executes a given number of simulations to estimate the minimum/maximum value of the specified expression. It is worth noting that the Pr(…) query implicitly and automatically infers a required number of simulation runs to estimate the event occurrence probability. For more details, the reader is referred to [35].

4. Reducing Taubenfeld's algorithm into Uppaal models

Algorithm 2 can be reduced to Uppaal by combining untimed and timed modeling features. In the following, the modeling aspects of atomic registers will be first explored. As a consequence, the read/write operations on the same register are assumed to be indivisible. Atomicity can be directly achieved by realizing an operation as part of the guarded command attached to an edge. The operation can concretely be specified in the guard or the update component of the command.

The reduction process of Algorithm 2 in the timed automata of Uppaal is guided by two conflicting goals: 1) reproducing concurrency and non-determinism among the process actions, and 2) controlling the non-determinism degree, together with a minimization of the memory footprint corresponding to the number of data variables and their possible values used in the model, to possibly improve the model checking activities.

A useful modeling abstraction assumes all the elementary process actions are instantaneous, that is, they have a duration of 0 time units. Unlike [11,12,16], the critical section is also assumed to be instantaneous. This adaptation was assumed to be more suited to reproduce the intrinsic concurrency dynamics of a mutual exclusion algorithm. Instantaneous actions are purposely realized as edge commands exiting from urgent locations. This way, the execution order of the elementary actions becomes truly concurrent and non-deterministic. However, the modeling of busy-waiting loops is critical. A cascade of urgent actions in a cycle can easily determine zeno-cycles [39] in the model. Although a busy-waiting loop wastes CPU cycles on a physical core in the concrete implementation of the algorithm, a continuous, also infinite, cycle of urgent actions can complicate the verification activities of the model checker, for the increased, yet redundant, nondeterminism. Our solution is to model a busy-waiting loop by a normal location, without the invariant, which should be abandoned as soon as there is an indication that the spinlock can be broken. A Boolean try function can purposely be introduced that checks, optimistically, if the termination condition of the busy-waiting can be true. In this case, an immediate exit from the normal location is commanded, e.g., by generating a synchronization signal over an urgent and broadcast channel that, purposely, has no receiver. The synchronization only provides urgency to the exit from the normal location. In particular, the try function can evaluate a complex expression based on multiple shared registers, without considering the atomic accesses. However, following an attempted exit, the busy-waiting condition is effectively evaluated, component by component, with full compliance with the atomic accesses. The normal location is immediately re-entered if any information emerges that the busy-waiting exit was erroneously taken. The overall construction is aimed at reducing the amount of nondeterminism that comes with the busy-waiting loops.

Because of the hypothesis, in Algorithm 2, of non-terminating and fault-free processes, the non-critical section should not be modeled as a normal location where a process can also remain for infinite time (to represent possible process termination). Practically, a normal location with a minimal duration (e.g., 1 time unit) as the invariant can be used. It is worth noting, though, that the alternative of an urgent location for the non-critical section can create problems for the model of Algorithm 2 (see later in this paper).

4.1. Global declarations

After carefully analyzing Algorithm 2, a fundamental decision was taken concerning a possible reduction of the adopted data variables. According to Algorithm 2, a process repeatedly copies, at many points, the entries of the shared registers onto a local myview[] array, which is then checked to assess a particular data configuration without the constraints of atomic accesses. Of course, for non-determinism, the copied values in myview can be changed in the global registers just before they are consulted. We then decided, with total compliance with the algorithm logic, to avoid the introduction of myview[] and test directly, by atomic accesses, the entries of the global shared registers.

Although Algorithm 2 does not depend on the particular assigned process identifiers, and each process does not assume the identifiers of partners, for modeling simplicity, the process identifiers were kept in the range from 1 to N. In addition, the m shared registers are selected by indices from 1 to m. Some global declarations of the Uppaal model of Algorithm 2 are the following:

```
const int N=2; // number of processes
typedef int[1,N] pid; // type range of process identifiers
const int m=7; // number of anonymous registers
typedef int[1,m] indices; // type range of the entry indexes of the anonymous registers
const int W=N+1; // Waiting status
typedef int[0,W] values; // possible register values: 0 (default) free status, 1..N pid, W waiting status
urgent broadcast chan synch; // broadcast channel used for busy-waiting exits
// shared communication variables
values reg[indices];
// process permutations
const indices perm[pid][indices]={
{1,2,4,3,7,5,6}
,{4,3,2,6,5,7,1}
/*
,{7,2,4,1,3,6,5}
...
*/
};
```

As one can see, each process is assigned a permutation of the indexes over the anonymous shared registers reg[]. Local declarations of each process automaton include the following (for brevity, the mycounter local variable in Algorithm 2 was simply renamed as c):

```
// local declarations of each process instance
indices j, k; // 1 initially
int[0,m] c; // 0 initially
bool mygo; // false initially
clock x; // clock variable
```

The two local functions r(j) and w(j, v), shown in Algorithms 3 and 4, respectively, access the value of the j-th entry of the global registers and write it with the v value. The j-th entry is accessed according to the view (index permutation) of the process i.

Algorithm 3. The read function of a register.

```
values r(const indices j){
    return reg[perm[i][j]];
}//r
```

Algorithm 4. The write function of a register.

```
void w(const indices j, const values v){
    reg[perm[i][j]]=v;
}//w
```

Each process template has only one parameter, const pid i. This way, at the system initialization, the following configuration line

```
system Process;
```

will automatically create N Process instances, which can be distinguished from one another as Process(1), …, Process(N).

Figure 1 shows the Uppaal model of the generic process of Algorithm 2. To facilitate model reading, the major line numbers of Algorithm 2 are recalled by using location names Lx. According to the Uppaal graphical editor conventions, guards are shown in green, channel synchronizations are in azure, and update actions are in blue.

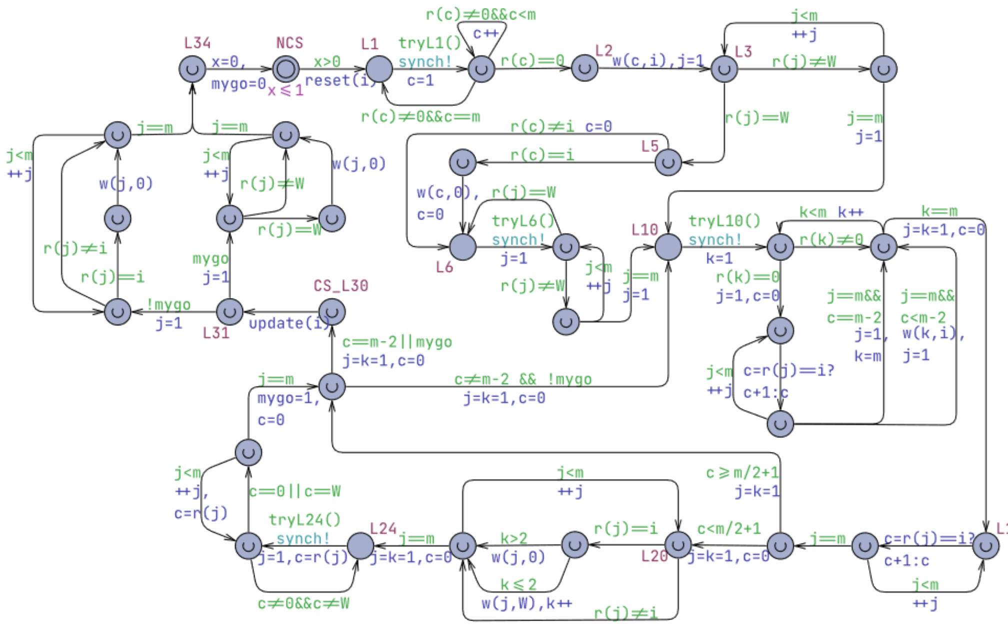


Figure 1. Uppaal Process automaton of Algorithm 2, with try functions indicated.

In Algorithm 2, four busy-waiting loops start at line numbers 1, 6, 10, and 24. Such points are indicated in the named locations L1, L6, L10, and L24 in Figure 1. The corresponding try functions are detailed in Algorithms 5 and 6.

The built-in primitive functions exists(), forall(), and sum() are used. The expression "exists(j:indices) r(j)==0;" returns true if an index exists (in the process permutation) where the corresponding register is 0. Similarly, "forall(j:indices) r(j)!=W;" returns true if all register entries have a value that differs from the W waiting status. Finally, "ci=sum(j:indices) r(j)==i;" assigns to ci the number of register entries that contain the process identifier i.

Algorithm 5. The tryL1(), tryL6() and tryL24() functions.

```
bool tryL1(){
    return exists(j:indices) r(j)==0;
}//tryL1
bool tryL6(){
    return forall(j:indices) r(j)!=W;
}//tryL6
bool tryL24(){
    return forall(j:indices) (r(j)==0||r(j)==W);
}//tryL24
```

Algorithm 6. The tryL10() function.

```
bool tryL10(){
    int ci=sum(j:indices) r(j)==i;
    int c0=sum(j:indices) r(j)==0;
    int cW=sum(j:indices) r(j)==W;
    if( ci+c0>=m/2+1 ) return true;
    ci=ci+c0; c0=0;
    if( ci>=2 ){  ci=ci-2; cW=cW+2; c0=c0+ci; }
    else if( ci==1 ){ ci=0; cW=cW+1; }
    if( c0+cW==m ) return true;
    return false;
}//tryL10
```

The tryL10() function first tests if the number of empty (0) register entries, together with those having the process identifier i, can reach the value of $\lceil m/2 \rceil = m/2 + 1$. In this case, the busy-waiting from L10 is tentatively exited (first reason to be a winner). Otherwise, the second condition to be a winner (all the entries have a 0 or W value, and variable mygo will be set to true) is checked. Following a true value of tryL10(), the exact situation of the register entries is assessed, and in the case the process was in reality a loser, $\mathrm{c} \neq \mathrm{m} - 2$ && ! mygo, the L10 location is re-entered. For brevity, the equivalences 0-false, 1-true for Booleans are used in Figure 1.

Provisions were taken in Figure 1 to detect the overtaking factor (OV), that is, the number of times a competing process is bypassed by peers that precede it in entering their critical section. A bounded value of the OV predicates about the starvation-free property of the mutual exclusion algorithm. The following further global declarations were added together with the reset(i) and update(i) functions shown in Algorithms 7 and 8:

```
const pid tp=1; // target process
```

int ov=-1, OV=-1; //default values

Algorithm 7. The reset(i) function.

```
void reset(const pid i){
    if (i==tp) ov=0;
}// reset
```

Algorithm 8. The update(i) function.

```
void update (const pid i){
    if (i!=tp){
        if (ov!=-1) ov=ov+1;
    }
    else{
        if (ov>OV) OV=ov;
        ov=-1;
    }
}//update
```

As one can see, a process instance tp (target process) is selected to be observed. The $\mathrm{reset(i)}$ function is called when the process i exits the non-critical section (location NCS) to signal (when $\mathrm{tp} = = \mathrm{i}$) the start of a new competition ($\mathrm{ov} = 0$). The $\mathrm{update(i)}$ function is invoked when a process exits its critical section (see the location $\mathrm{CS\_L30}$ in Figure 1). In the case that the target process is competing ($\mathrm{ov} \neq -1$), and a process distinct from tp is exiting its critical section, the overtaking count ov gets incremented. When the target process abandons the critical section, the maximal value of the overtaking is retained in the OV variable, and the ov is assigned the default value of -1.

4.2. Verification of the Taubenfeld basic model

The model in Figure 1 was thoroughly verified by initially using $\mathrm{N} = 2$ and $\mathrm{m} = 7$. The experiments were carried out by using Uppaal 5.0 on a Windows 11 Pro desktop platform with a Dell XPS 8940, Intel i7-10700, CPU@2.90 GHz, and 32 GB RAM. The following TCTL queries [28] assess the absence of deadlocks (1), the mutual exclusion observance (2), the overtaking bound OV (3) that verifies the starvation-freeness of the processes, and the liveness of processes (4):

$$\mathrm{A[]!\,deadlock}, \tag{1}$$

$$\mathrm{A[]\ (sum(i{:}\,pid)\ Process(i).CS\_L30) \leq 1}, \tag{2}$$

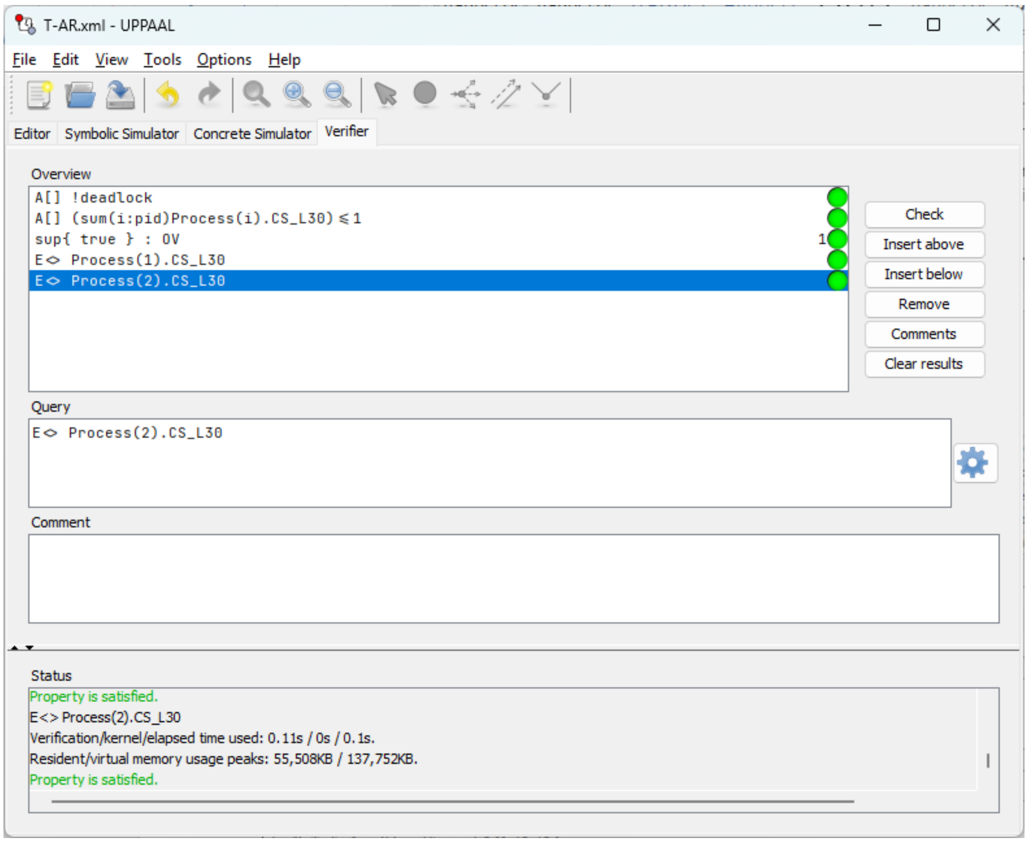
$$\mathrm{sup\{true\}\ :\ OV}, \tag{3}$$

$$\mathrm{E <>\ Process(1).CS\_L30} \text{ and so forth for the other processes.} \tag{4}$$

In particular, query (2) checks that it never happens, in the state graph, that more than one process can be found, at the same time, in its critical section. Satisfaction of each query is ensured in less than 1 s (see the snapshot in Figure 2).

A few preliminary experiments revealed that when the NCS location is turned to be urgent, or the guard $x > 0$ is dropped on the edge from NCS to $\mathrm{L1}$, meaning that the non-critical section can be abandoned in 0 time units (a process that exits its critical section immediately re-enters and starts competing), the model of Figure 1 loses the mutual exclusion property with $\mathrm{N} = 2$ processes.

Model checking also confirmed that when the number m of anonymous registers is smaller than 7, the model in Figure 1 (and Algorithm 2), while remaining deadlock-free, misses the fundamental mutual exclusion property. The model exhibited the same behavior when using m = 8 (absence of deadlocks and mutual exclusion violation) but came back to be fully correct with the same results as in Figure 2 when m = 9. Thus, the exhaustive verification confirmed the expectation formulated in [22] that for the algorithm to be correct, the number m should be odd and m ≥ 7.



```
A[] !deadlock
A[] (sum(i:pid)Process(i).CS_L30) ≤ 1
sup{ true } : OV                                   1
E<> Process(1).CS_L30
E<> Process(2).CS_L30
```

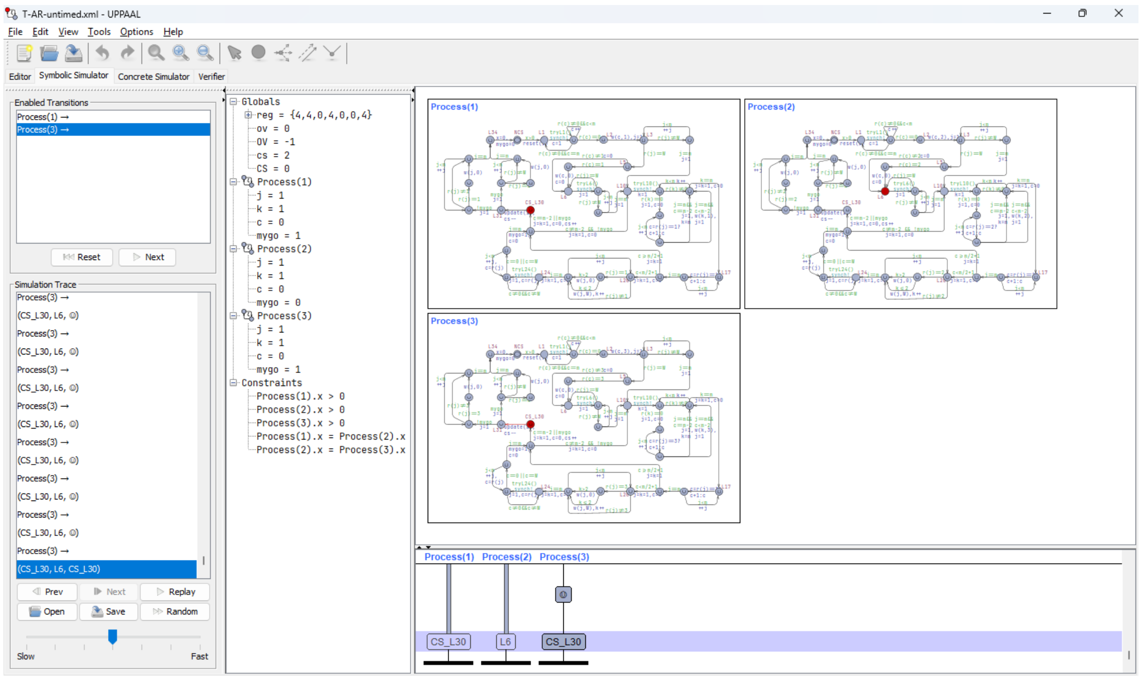
```
E<> Process(2).CS_L30
```

```
Property is satisfied.
E<> Process(2).CS_L30
Verification/kernel/elapsed time used: 0.11s / 0s / 0.1s.
Resident/virtual memory usage peaks: 55,508KB / 137,752KB.
Property is satisfied.
```

Figure 2. Snapshot of the Uppaal model checker for the model in Figure 1, with N = 2 and m = 7, which confirms that all property queries are met.

The next experiments were devoted to studying the model in Figure 1 for N = 3. Unfortunately, checking for the absence of deadlocks incurs a state explosion, and it is inconclusive. However, the query (2) about mutual exclusion terminates unsatisfactorily (multiple processes can simultaneously enter their critical section) after 559 s with a peak of memory usage of 20 GB, as witnessed by the snapshot of the symbolic simulator in Figure 3. A state can be reached where both process 1 and process 3 are simultaneously winners, with their mygo variable becoming true. This result confirms the impossibility result stated in [22] (see also Section 2.1) about the impossibility that Algorithm 2 can be starvation-free for N ≥ 3. But, as our result indicates, the problem is not related to the starvation-free property but to the mutual exclusion violation. Consequently, Algorithm 2 remains effectively correct, under atomic registers, only for N = 2 processes.

The model of Figure 1 was also used to test the effectiveness of the adopted try functions (see Algorithms 5 and 6) on the model analysis. In the case that the try functions are omitted, the nondeterminism degree is maximal, and the model for N = 3 processes is affected by state explosions that deny checking the mutual exclusion property, which was instead correctly documented in Figure 3 with the help of the try functions.



```
Enabled Transitions
Process(1) →
Process(3) →

Simulation Trace
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, ⊚)
Process(3) →
(CS_L30, L6, CS_L30)
```

```
Globals
  reg = {4,4,0,4,0,0,4}
  ov = 0
  OV = -1
  cs = 2
  CS = 0
Process(1)
  j = 1
  k = 1
  c = 0
  mygo = 1
Process(2)
  j = 1
  k = 1
  c = 0
  mygo = 0
Process(3)
  j = 1
  k = 1
  c = 0
  mygo = 1
Constraints
  Process(1).x > 0
  Process(2).x > 0
  Process(3).x > 0
  Process(1).x = Process(2).x
  Process(2).x = Process(3).x
```

Figure 3. Snapshot showing mutual exclusion is violated by the model in Figure 1 when N = 3.

4.3. Checking Taubenfeld's algorithm with non-atomic registers

Algorithm 2 was also studied when anonymous registers are non-atomic [12–16]. Since the Multiple Writers Multiple Readers (MWMR) pattern applies to m registers, both the *scrambling* effect (due to simultaneous write operations on the same register, which determine that a non-deterministic value, in the type domain of the register, is finally left in the register), and the *flickering* effect (due to one or multiple readers that read simultaneously to a write operation on the same register, with each reader that gets a non-deterministic value belonging to the register type domain) should be considered. In the following, as, e.g., in [14], only flickering is considered, and multiple write operations are supposed to be fenced and enforced to occur one at a time. Toward this, as in [16], the model of Figure 1 was decorated by adding a Boolean variable per register, whose value true (or 1) indicates the register is under writing. A read operation that finds the register is under writing returns a non-deterministic value chosen in the values type range of the register. All of this can be achieved by using the non-deterministic selection field (yellow colored in Figure 4) of a guarded command. More precisely, the following decoration variables were added to the model of Figure 1:

```
// decoration variables
bool uw[indices]; // all false/0 initially
```

and the function re(j) (Algorithm 9) was added locally to each process, which returns the effective shared register corresponding to an index j issued by process i.

A fragment of the decorated version of the model in Figure 1 is reported in Figure 4. As one can see, at the time of a write operation on a register, the corresponding uw[] variable is set to true (1). In the subsequent step, the variable is reset to 0. Similarly, during a read operation, depending on whether the register is underwriting or not, either the effective register value or the non-deterministic value v is used.

The complete model of Figure 4 turns out to have a greater degree of nondeterminism. However, the verification of the decorated model for N = 2 reported that it is deadlock-free, but it violates the mutual exclusion. In one case, both processes were found to be winners with their mygo variable set to true.

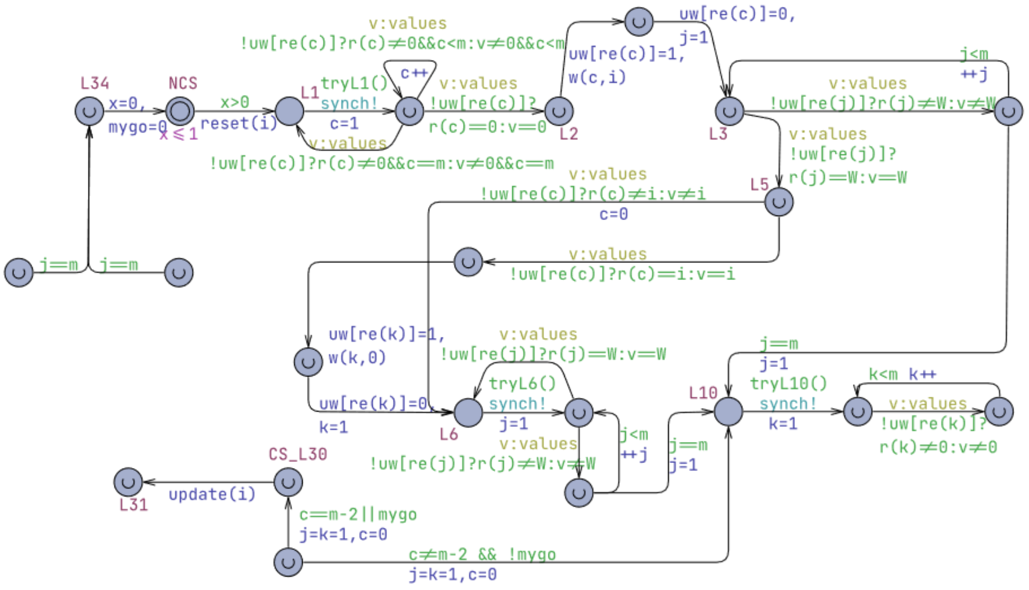


Figure 4. Fragment of the model of Figure 1 adapted to work with non-atomic registers.

Algorithm 9. Function that maps an index j issued by process i to the corresponding shared register.

```
indices re(const indices j){
   return perm[i][j];
}//re
```

4.4. A timed model

An important feature of the model in Figure 1 concerns the absence of any hypothesis about the relative speeds of the processes. A process can remain in an urgent location while another, for non-determinism, executes a complete cycle of urgent locations for checking the shared register entries. This behavior can be approximated, in a timed version of the model, by exploiting stochasticity and probability distribution functions (PDF). In the following, a time interval $[L_b..U_b]$, with $0 < L_b$, $L_b \leq U_b, U_b < \infty$, is supposed to characterize the duration of any process action (a read, write, test, and so forth, operation). Operations that occur at the same time will be executed in an arbitrary order. Following operations will be executed according to the uniform PDF associated with the $[L_b..U_b]$ time interval. The timed model can be analyzed using either the exhaustive model checker (MC), provided the bounds of the time intervals are integers, or by the statistical model checker (SMC) that can exploit doubles as the interval bounds and rests on simulations and stochasticity for property estimation (see also [36]).

Figure 5 shows a snippet of the timed model that gives a hint about how urgent locations of Figure 1 are replaced by normal locations with the invariant $x \leq U_b$ and the guard $x \geq L_b$ for enabling their exit.

In a first scenario, the timed model was studied using the model checker (MC) with L_b=1, U_b=2, that is, all elementary actions have a minimal non-zero duration. The time window uncertainty reproduces nondeterminism. The MC confirmed the results observed on the basic model of Figure 1 for N = 2, including the indications about the minimal number m = 7 of required shared registers, and the scenario N = 3, which violates the mutual exclusion.

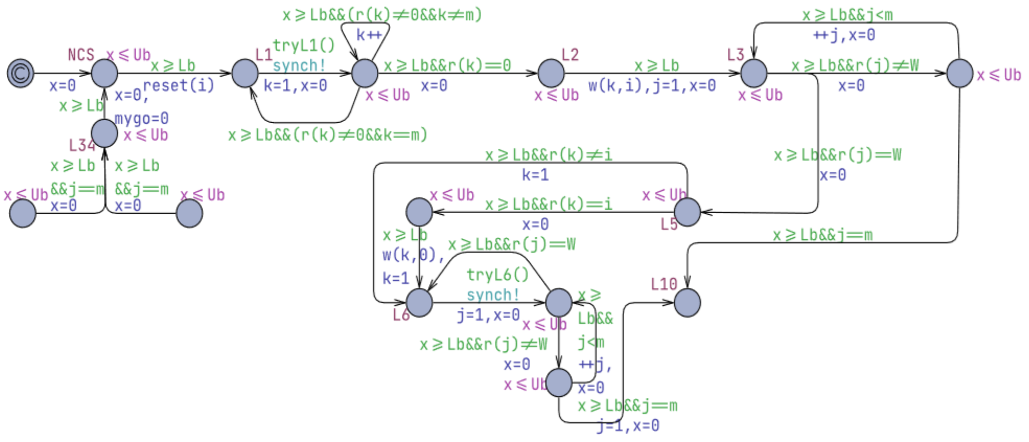


Figure 5. Fragment of the timed version of the model of Figure 1.

In a second case, the time model for N = 2 processes was analyzed through the SMC. Under SMC, nondeterminism is imitated by action stochasticity. The following MITL [35] query was issued to quantify the probability of the event: "*can a state be reached where multiple processes are found simultaneously in their critical sections?*"

Pr[<=100000](<>(sum(i:pid)Process(i).CS_L30)>1). (5)

Query (5) executes 72 simulation runs (established by Uppaal SMC), each lasting 10^5 time units, and proposes a probability that is ≤ 0.0499441, with a confidence degree CI = 95%. With the default statistics parameter settings, the suggested probability refers to a "practically" impossible event. A query like the following can check the liveness of processes:

Pr[<=10000](<>Process(1).CS_L30). (6)

Query (6), on any process, suggests a confidence interval of about 0.969349 ± 0.03 (95% CI) that qualifies an (almost) *certain event*.

Absence of deadlocks was implicitly checked by observing that simulations always run to completion. Some decoration variables were introduced to monitor and count the number of critical sections executed by each process:

double ncs[pid]; // all 0 initially

The update(i) function of Algorithm 8 was adjusted to also increment the ncs[i] variable. Then, the following query was launched that executes one simulation run lasting 10^5 time units, and generates, at the query completion, the graphical output depicted in Figure 6.

simulate [<=100000]{ncs[1], ncs[2]}. (7)

As one can see from Figure 6, the number of executed critical sections almost coincides for the two processes, and always increases until the simulation end, thus witnessing model liveness. Figure 6 also suggests the model should be starvation-free. An estimation of the overtaking bound can be achieved by the query:

E[<=10000;10](max:OV). (8)

Query (8) launches a batch of 10 simulation runs and monitors the emerging maximum value of OV. SMC suggests a maximum value of ≈ 2, thus sustaining the starvation-free property.

Subsequently, the time model was exploited for studying the case $N = 3$ processes. Query (5) now suggests a confidence interval of ≥ 0.950056 (95% CI), thus witnessing that the mutual exclusion can be violated.

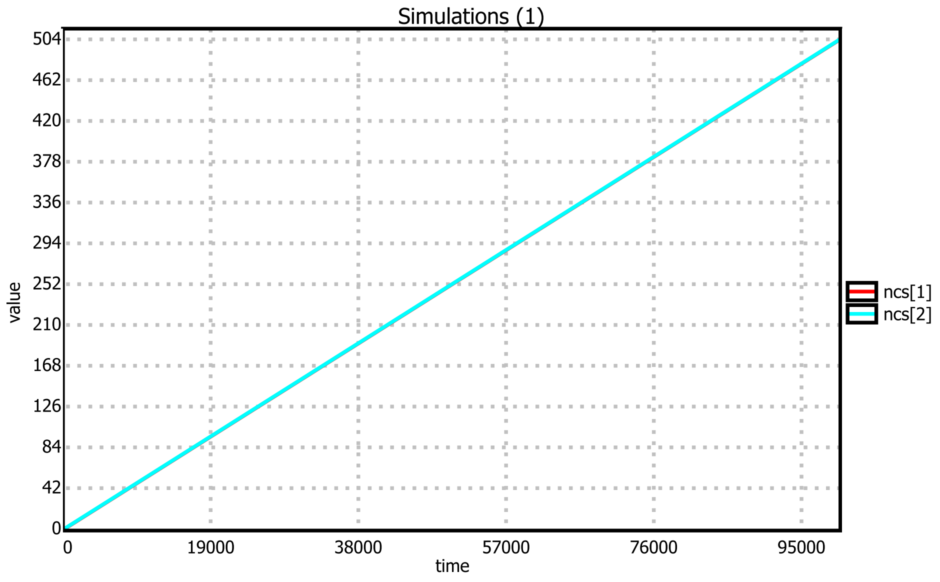


Figure 6. Observed number of critical sections executed by the timed model for N = 2 processes.

5. Embedding Taubenfeld's algorithm in a tournament tree organization

Based on the possibility and impossibility results (see Section 2.1) of Taubenfeld's mutual exclusion algorithm [22] (see Algorithm 2) operating on anonymous memory, confirmed in this paper through the Uppaal model checkers [28,35], the following proposes a novel solution that extends Algorithm 2, which is proved to be correct for $N = 2$ processes and atomic registers, to the more general case of $N \geq 2$. The solution is based on a standard, state-of-the-art tournament tree (TT) organization [12,38,39], in which the basic algorithm for $N = 2$ is responsible for managing pairs of processes at each *arbitration node* of the TT.

In this paper, TT is implemented as a binary tree (see Figure 7 for the case $N = 3$) where arrival processes enter leaf nodes (bold circles in Figure 7) of the (in general partially occupied) last level $lev = \lceil \log_2 N \rceil$ of the tree. The total number of TT nodes is $T = 2^{lev} + N - 1$, and they are numbered from 1 to T. At its arrival, a process occupies the first free leaf node. In Figure 7, process i can occupy a node from 4 to 7. This random choice of the leaf node was deemed more natural with the process anonymity assumption of Taubebfeld's algorithm, where the precise identifiers of competing processes are unimportant.

Arbitration nodes are the intermediate nodes 2, 3, and 1 in Figure 7. Sibling processes share the same ancestor node, and apply Algorithm 2 for two processes to establish the local winner, which advances, in the TT, to the ancestor node. If t is the current node occupied by the process, its ancestor node is $t / 2$. The loser will wait at its currently occupied node. The first process that overcomes all the encountered arbitration nodes and reaches and occupies the root node 1 is the overall winner and enters the critical section.

It is evident that the shared communication variables, that is, the anonymous register entries and the mygo variables, have to be replicated at each intermediate node. At its exit from the critical section, the overall winner will traverse, in the opposite direction, the path that it followed to reach node 1. During its back movements, the process will reset, in that order, the shared variables at each previously visited intermediate node. Such a reset can awaken a waiting process that then resumes its possible upward movement in the TT. Finally, the process abandons the TT, frees the previously occupied leaf node, and enters its non-critical section.

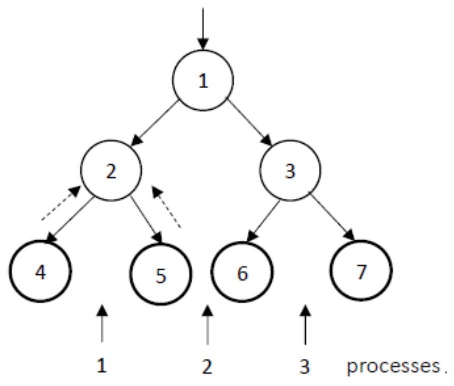


Figure 7. Tournament tree for N = 3 processes, which occupy, at their arrival, random leaf nodes.

5.1. The TT-T2 model

The following proposes a Uppaal model of Algorithm 2 for two processes, embedded in a tournament binary tree. It generalizes the model in Figure 1. The new model and algorithm will be referred to as TT-T2. First of all, the Uppaal global declarations of TT-T2 are clarified.

```
const int N=…; // number of processes
typedef int[1,N] pid; // process indexes
const int T=fint( pow(2,ceil(log2(N))) )+N-1; // total number of TT nodes
const int I=fint( pow(2,ceil(log2(N))) )-1; // number of intermediate nodes
typedef int[1,T] tindex; // TT node indexes
typedef int[1,I] iindex; // TT intermediate nodes indexes
typedef int[I+1,2*I+1] lindex; // leaf node indexes
bool leaf_node[lindex]; // all true initially
const int m=7; // number of anonymous registers
typedef int[1,m] indices; // range of indexes of anonymous registers
const int W=N+1; // Waiting value
typedef int[0,W] values; // register values: 0 free or default status, 1..N pid, W waiting status
// shared communication variables
values reg[iindex][indices]; // one instance of registers per intermediate mode
bool mygo[iindex]; // one variable per intermediate node
```

The data structures reg[][] and mygo[] represent the TT. Each process i uses a local variable t, initialized at the start of a new competition, with the leaf node assigned to i. The identity of the leaf node is returned by the local function leaf(), which also stores the leaf node index in a local variable of the process. The release_leaf() function frees the previously occupied leaf node. Algorithm 10 shows the initialize() function, which frees (assignment of the value true or 1) each leaf node of the TT. The initialize() function is invoked once by a Bootstrapper automaton (see Figure 8), which, finally, takes no further role in the model.

Algorithm 10. The initialize() function of the TT-T2 model.

```
void initialize(){
    int l;
    for (l=I+1; l<=2*I+1; ++l) leaf_node[l]=1;
}// initialize
```

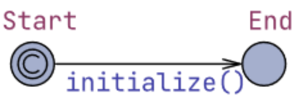


Figure 8. The Bootstrapper automaton, which is responsible for model initialization.

Figure 9 reports the TT-T2 Uppaal model. For simplicity, each process uses the same permutation at each arbitration node during the up movement toward the root node. Since each intermediate node has its copy of the registers, the local functions (see Algorithm 11), which read/write a register entry, are generalized by taking also as a parameter the identity of the node with the registers to use. This is especially important during the back movements and resets of a process.

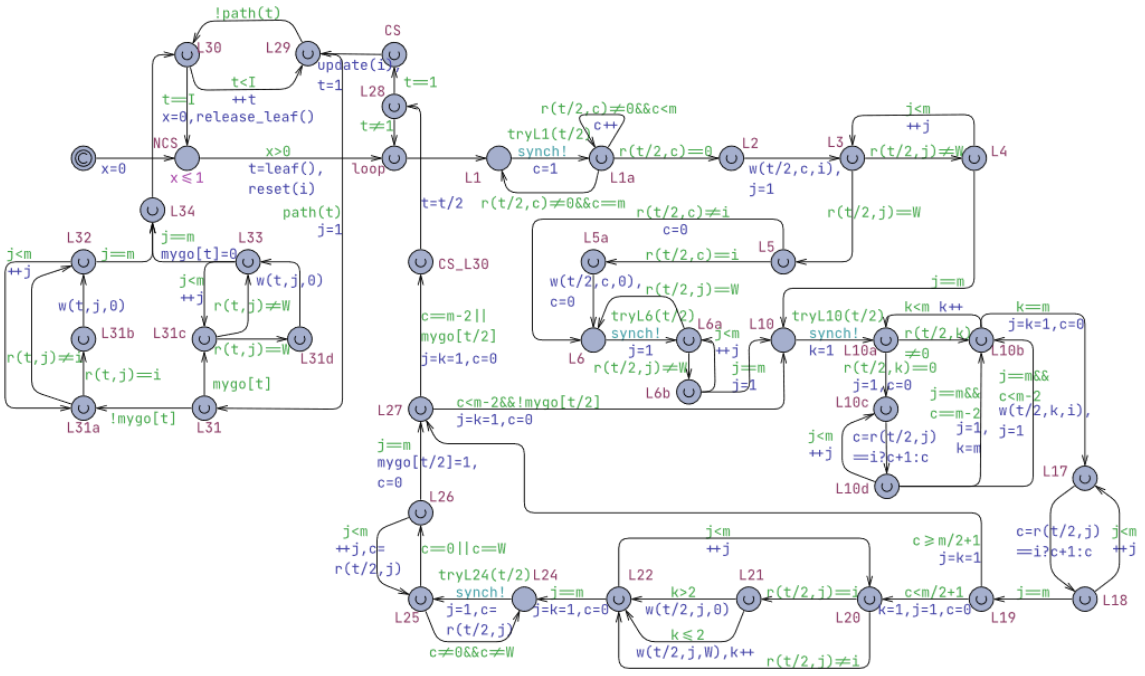


Figure 9. The TT-T2 complete Uppaal model.

Algorithm 11. The new r(t, j) and w(t, j, v) local functions of Process(i).

```
values r(const iindex t, const indices j){
    return reg[t][perm[i][j]];
}//r
void w(const iindex t, const indices j, const values v){
    reg[t][perm[i][j]]=v;
}//w
```

From the location L1 to CS_L30, one can easily retrieve, in Figure 9, the basic actions of the model of Figure 1. However, at each read/write operation of a register entry, the ancestor node $t/2$ is used. In addition, CS_L30 now represents the local critical section, which refers to the current arbitration.

When Process (i) exists from its NCS location, thus starting a new competition, its current node t is initialized with its assigned leaf node. The process then executes a loop (see the loop location in Figure 9) until t becomes equal to the root node (1).

Each time the process is a local winner, variable t is updated with its ancestor node, $t = t/2$, and the next loop iteration is possibly executed. When the process becomes a global winner, the CS location is finally entered. After that, the clearing process is launched, starting from the root node (the local variable t is set to 1). If the node t belongs to the path traversed by the process (the function $path(t)$ returns true) for reaching the root node, depending on the value of the $mygo[t]$ variable, the reset operations on the register entries of node t are carried out as in Algorithm 2. The clear actions are continued until the last intermediate node of the path is reached. Finally, the previously occupied leaf node is freed, and the NCS is entered.

The TT-T2 model was verified with $N = 2$, and the same correct results observed with the basic model of Figure 1 were retrieved. However, for $N > 2$, as expected, TT-T2, having a much more expensive state graph than the basic model of Figure 1, suffers from state explosions, which forbid model checking. As a consequence, an alternative, timed version of the TT-T2 model was prepared and experimented with as described in the next section.

5.2. The TT-T2-timed model

A TT-T2-timed model was achieved from the TT-T2 model by attaching the time interval $[L_b..U_b]$ to every process action, as discussed in Section 4.4, and exemplified in Figure 5. The new Uppaal model, not shown for brevity, was studied only with the statistical model checker (SMC), because the model checker, even in the case $N = 2$, is affected by state explosions. The particular bounds $L_b = 0.1$ and $U_b = 0.5$ were used to reproduce, in simulation, the necessary nondeterminism in the process actions. In the TT-T2-timed model, the verification of mutual exclusion was achieved by introducing a global variable (initialized to 0), cs. As soon as a process is up to enter its global critical section, cs is incremented. At the exit from the critical section, cs is decremented. By declaring cs of the bounded type, int[0,1] cs, any executed query implicitly also checks mutual exclusion by having that in no case a value greater than 1 is attempted to be assigned to cs.

5.2.1. Simulation experiments

The TT-T2-timed model was simulated by varying N from 2 to 16 as powers of 2, which guarantees the TT is always well-balanced. In all the cases, the event of mutual exclusion violation was found to be “practically” impossible to occur. Tables 1 and 2 collect the estimated values of the overtaking bound OV and the required elapsed real-time in seconds, when the try functions are enabled and when they are disabled, respectively. In particular, the following MITL query was used:

E[<=tEnd;30] (max:OV).

The query executes 30 simulations each lasting $tEnd$ time units and proposes a confidence interval for the OV value.

In Table 2 (try functions disabled), the case $\mathrm{N} = 16$ was impossible to handle due to the increased model nondeterminism for the active busy-waiting loops. All of this causes heap space problems during data accumulation in the statistical model checker (SMC).

Table 1. Experimental results of simulating the TT-T2-timed model with $\mathrm{L_b} = 0.1,\ \mathrm{U_b} = 0.5$, with the try functions enabled, tEnd=$1.5x10^4$.

N	OV (95% CI)	ET(s)
2	1.83 ± 0.2	3.45
4	4.23 ± 0.5	5.63
8	8.1 ± 1.4	12.17
16	14.7 ± 2.9	25.61

Table 2. Experimental results of simulating the TT-T2-timed model with $\mathrm{L_b} = 0.1, \mathrm{U_b} = 0.5$, with the try functions disabled, tEnd=$1.5x10^4$.

N	OV (95% CI)	ET(s)
2	≈ 2	4.81
4	5.1 ± 0.2	11.67
8	10.7 ± 0.4	28.91
16	$-$	$-$

The experiments confirmed the usefulness of the try functions (see Algorithms 5 and 6) for model analysis. The positive impact on the results' careful estimation and lower computational time should be noted. As witnessed by the values of Table 1, as N grows, OV tends to the $\mathrm{N} - 1$ bound, thus guaranteeing satisfaction of the starvation-free property.

In the light of the SMC experiments, it emerged that the tournament tree-based solution of Algorithm 2 appears as a viable and correct generalization, for $\mathrm{N} \geq 3$ processes, of the basic correct algorithm handling $\mathrm{N} = 2$ processes.

6. Conclusions

Studying concurrent systems in anonymous shared memory [20–22] is challenging but very interesting and promising because it can directly support some bioinspired applications, particularly at the molecular level, as in the process of genome-wide epigenetic modifications [23]. A specific mutual exclusion algorithm for concurrent/parallel processes communicating to one another by anonymous shared memory registers was proposed recently by Taubenfeld in [22], together with some possibility and impossibility results. The fundamental possibility result concerns the existence of a correct mutual exclusion solution (with the absence of starvation and deadlocks) for $\mathrm{N} = 2$ processes, when $\mathrm{m} \geq 7$ (m odd) registers are used. The impossibility results regard the non-existence of a starvation-free mutual exclusion solution for $\mathrm{N} \geq 3$ processes, whatever the number m of used anonymous registers. These results were stated in [22] based on informal mathematical reasoning.

The original contribution of this paper consisted of formal modeling and verification of the mutual exclusion algorithm and results described in [22], using timed automata [27] and the Uppaal toolbox [28,35], which provides both an exhaustive model checker (MC) and a simulation-based statistical model checker (SMC) [34]. This paper first detected a time-dependency in the algorithm in [22], and proved, by model checking, the correctness of the basic solution for $\mathrm{N} = 2$ processes with the minimal number of $\mathrm{m} = 7$ anonymous and atomic registers. The paper has also shown that when non-atomic registers [13–16] are used, the basic algorithm ceases to be correct. Our paper demonstrated, also by model checking, that for N = 3, the algorithm in [22] is incorrect not because it is no longer starvation-free but because it loses the fundamental mutual exclusion property.

As another contribution, this paper proposed an embedding of the basic solution for N = 2 of [20] as the arbitration algorithm for pairs of processes, in the context of a general, state-of-the-art tournament tree (TT) organization [12,37,38], thus opening it to N ≥ 2 processes. Since the TT-based model is affected by state explosions even with N = 3, a timed version of the TT model is considered where the elementary actions can occur, realistically, in a given time interval. The TT timed model was then verified by MC when the actions have all the same minimal duration (1 time unit) and by the SMC when actions have a stochastic duration provided by a uniform probability distribution.

Future work will focus on the following points: First, the developed Uppaal models will be ported to a high-performance computer platform equipped with, e.g., 128 GB of RAM, in order to better exploit the verification capabilities of the model checker. Second, experiments will be conducted with Taubenfeld's algorithm implemented in Java on top of the efficient Theatre actor system [12,40] to improve model scalability. The idea is to map each process instance to a distinct actor allocated to a different core. The Theatre's abilities will then be exploited to enable untimed or timed (simulated or real-time) parallel execution, supervised by a suitable control machine. Third, the verification of the Uppaal models presented in this paper, based on timed automata, will be compared with that of the model checker mCRL2 [15,26], which relies on process-algebra models.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Conflict of interest

The authors declare no conflict of interest.

References

1. L. Lamport, The mutual exclusion problem: part I, In: *Concurrency: the works of Leslie Lamport*, New York: Association for Computing Machinery, 2019, 227–245. https://doi.org/10.1145/3335772.3335937
2. L. Lamport, The mutual exclusion problem: part II, In: *Concurrency: the works of Leslie Lamport*, New York: Association for Computing Machinery, 2019, 247–276. https://doi.org/10.1145/3335772.3335938
3. M. Raynal, D. Beeson, *Algorithms for mutual exclusion problem*, Cambridge: The MIT Press, 1986.
4. M. Raynal, G. Taubenfeld, A visit to mutual exclusion in seven dates, *Theor. Comput. Sci.*, **919** (2022), 47–65. https://doi.org/10.1016/j.tcs.2022.03.030
5. M. Raynal, *Concurrent programming: algorithms, principles, and foundations*, Berlin: Springer-Verlag, 2013. https://doi.org/10.1007/978-3-642-32027-9
6. E. Dijkstra, About the sequentiality of process descriptions, University of Texas at Austin, 1962.
7. E. Dijkstra, Co-operating sequential processes, In: *The origin of concurrent programming*, New York: Springer, 1968, 65–138. https://doi.org/10.1007/978-1-4757-3472-0_2

8. T. Dekker, History of Dekker's algorithm for mutual exclusion, In: *Tales of electrologica: computers, software and people*, Cham: Springer Nature, 2022, 111–120. https://doi.org/10.1007/978-3-031-13033-5_6
9. B. Szymanski, A simple solution to Lamport's concurrent programming problem with linear wait, *Proceedings of the 2nd international conference on Supercomputing*, 1988, 621–626. https://doi.org/10.1145/55364.5542
10. B. Szymanski, Mutual exclusion revisited, *Proceedings of the 5th Jerusalem Conference on Information Technology*, 1990, 110–117. https://doi.org/10.1109/JCIT.1990.128275
11. L. Nigro, F. Cicirelli, F. Pupo, Modeling and analysis of Dekker-based mutual exclusion algorithms, *Computers*, **13** (2024), 133. https://doi.org/10.3390/computers13060133
12. L. Nigro, F. Cicirelli, Property assessment of Peterson's mutual exclusion algorithms, *Appl. Comput. Intell.*, **4** (2024), 66–92. https://doi.org/10.3934/aci.2024005
13. L. Lamport, Concurrent reading and writing, *Commun. ACM*, **20** (1977), 806–811. https://doi.org/10.1145/359863.359878
14. P. Buhr, D. Dice, W. Hesselink, High-performance N-thread software solutions for mutual exclusion, *Concurr. Comp.-Pract. E.*, **27** (2015), 651–701. https://doi.org/10.1002/cpe.3263
15. M. Spronck, B. Luttik, Process-algebraic models of multi-writer multi-reader non-atomic registers, *Proceedings of the 34th International Conference on Concurrency Theory (CONCUR)*, 2023, 1–29.
16. L. Nigro, Verifying mutual exclusion algorithms with non-atomic registers, *Algorithms*, **17** (2024), 536. https://doi.org/10.3390/a17120536
17. *L. Frenzel, Dual-port SRAM accelerates smart-phone development*, Electronic Design, 2004. Available from: https://www.electronicdesign.com/technologies/industrial/boards/article/21774041/dual-port-sram-accelerates-smart-phone-development.
18. Z. Wang, Q. Zuo, J. Li, An intelligent multi-port memory, *Proceedings of International Symposium on Intelligent Information Technology Application Workshops*, 2008, 251–254. https://doi.org/10.1109/IITA.Workshops.2008.231
19. M. Raynal, G. Taubenfeld, Fully anonymous shared memory algorithms, arXiv: 1909.05576. https://doi.org/10.48550/arXiv.1909.05576
20. G. Taubenfeld, Anonymous shared memory, *J. ACM*, **69** (2022), 24. https://doi.org/10.1145/3529752
21. Z. Aghazadeh, D. Imbs, M. Raynal, G. Taubenfeld, P. Woelfel, Optimal memory-anonymous symmetric deadlock-free mutual exclusion, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, 157–166. https://doi.org/10.1145/3293611.3331594
22. G. Taubenfeld, Memory-anonymous starvation-free mutual exclusion: possibility and impossibility results, arXiv: 2309.11337. https://doi.org/10.48550/arXiv.2309.11337
23. S. Rashid, G. Taubenfeld, Z. Bar-Joseph, The epigenetic consensus problem, In: *Structural information and communication complexity*, Cham: Springer, 2021, 146–163. https://doi.org/10.1007/978-3-030-79527-6_9
24. W. Hesselink, Verifying a simplification of mutual exclusion by Lycklama-Hadzilacos, *Acta Inform.*, **50** (2013), 199–228. https://doi.org/10.1007/s00236-013-0178-2
25. C. Baier, J. Katoen, *Principles of model checking*, Cambridge: MIT Press, 2008.
26. J. Groote, J. Keiren, B. Luttik, E. de Vink, T. Willemse, Modelling and analyzing software in mCRL2, In: *Formal aspects of component software*, Cham: Springer, 2020, 25–48. https://doi.org/10.1007/978-3-030-40914-2_2

27. R. Alur, D. Dill, A theory of timed automata, *Theor. Comput. Sci.*, **126** (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8
28. G. Behrmann, A. David, K. Larsen, A tutorial on UPPAAL, In: *Formal methods for the design of real-time systems*, Berlin: Springer, 2004, 200–236. https://doi.org/10.1007/978-3-540-30080-9_7
29. E. Clarke, W. Klieber, M. Nováček, P. Zuliani, Model checking and the state explosion problem, In: *Tools for practical software verification*, Berlin: Springer, 2012, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1
30. *Uppsala University, Aalborg University*, *Uppaal on-line*, Uppaal tool, 2025. Available from: https://uppaal.org.
31. Uppaal Tutorial, 1999. Available from: https://www.cis.upenn.edu/~lee/09cis480/lec-part-3-uppaal-inside.pdf.
32. W. Zhou, Y. Zhao, Y. Zhang, Y. Wang, M. Yin, A comprehensive survey of UPPAAL-assisted formal modeling and verification, *Softw. Pract. Exp.*, **55** (2025), 272–297.
33. I. Grobelna, K. Gajewski, A. Karatkevich, Systematic review on the applications of Uppaal, *Sensors*, **25** (2025), 3484. https://doi.org/10.3390/s25113484
34. G. Agha, K. Palmskog, A survey of statistical model checking, *ACM Trans. Model. Comput. Simul.*, **28** (2018), 6. https://doi.org/10.1145/3158668
35. A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, Uppaal SMC tutorial, *Int. J. Softw. Tools Technol. Transf.*, **17** (2015), 397–415. https://doi.org/10.1007/s10009-014-0361-y
36. L. Nigro, F. Cicirelli, Formal modeling and verification of embedded real-time systems: an approach and practical tool based on Constraint Time Petri Nets, *Mathematics*, **12** (2024), 812. https://doi.org/10.3390/math12060812
37. J. Kessels, Arbitration without common modifiable variables, *Acta Inform.*, **17** (1982), 135–141. https://doi.org/10.1007/BF00288966
38. W. Hesselink, Tournaments for mutual exclusion: verification and concurrent complexity, *Form. Asp. Comp.*, **29** (2017), 833–852. https://doi.org/10.1007/s00165-016-0407-x
39. H. Bowman, R. Gomez, L. Su, A tool for the syntactic detection of zeno-timelocks in timed automata, *Electronic Notes in Theoretical Computer Science*, **139** (2005), 25–47. https://doi.org/10.1016/j.entcs.2005.09.006
40. L. Nigro, Parallel theatre: an actor framework in Java for high performance computing, *Simul. Model. Pract. Th.*, **106** (2021), 102189. https://doi.org/10.1016/j.simpat.2020.102189