Applied Computing and Intelligence
4 (1): 66–92.
DOI: 10.3934/aci.2024005
Received: 22 July 2024
Revised: 16 August 2024
Accepted: 21 August 2024
Published: 28 August 2024

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

Research article

Property assessment of Peterson's mutual exclusion algorithms

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: The goal of this work was to experiment with the formal modeling and automated reasoning of concurrent/parallel systems, mainly focusing on mutual exclusion algorithms. A concrete method is presented, which depends on timed automata and the model checker provided by the popular Uppaal toolbox. The method can be used for a thorough assessment of the properties of mutual exclusion algorithms. The paper argues that the proposed approach can be effective in moving beyond informal and intuitive reasoning about concurrency, which can be challenging due to the non-determinism and interleaving of the action execution order of the involved processes. The approach is described and applied to an in-depth analysis of Peterson's algorithms for $N = 2$ and $N > 2$ processes. The analysis allows the reconciliation of different evaluations reported in the literature, particularly for the overtaking bound, and also reveals new results not previously disclosed. The general case of $N > 2$ was handled within the context of the state-of-art, standard, and efficient tournament binary-tree organization, which uses the solution for two processes to arbitrate between pairs of processes at different stages of their upward movement along the tree. All mutual exclusion algorithms are investigated under both atomic and non-atomic memory models.

Keywords: mutual exclusion problem; safety and liveness properties; memory models; formal modeling; model checking; timed automata; Uppaal; Java

1. Introduction

This paper develops a method for formal modeling and automated reasoning of concurrent/parallel and distributed systems. It is well-known that informal, intuitive reasoning of concurrency and its properties can be very challenging, in general due to the non-determinism and interleaving that characterize the action execution order of the processes that compose the system. As a concrete application, pure software-based mutual exclusion algorithms [1,2] that are not based on special hardware mechanisms like *test-and-set* instructions are chosen, and their properties are assessed by model checking. Such algorithms are the bases upon which high-level mechanisms such as semaphores and monitors depend [3].

Mutual exclusion addresses the predictability of a concurrent system when a shared data resource can be accessed simultaneously by $N \geq 2$ processes. In these cases, it is mandatory that only one process at a time can access and modify the resource. The code fragment of a process when it is allowed to use the resource is called its *critical section*. On the other hand, other competing processes, which have signaled their interest in using the resource, have to wait, and the waiting time should be bounded (*absence of starvation*) to ensure that each process eventually enters its critical section.

The first mutual exclusion algorithm for $N = 2$ processes was attributed to Dekker by Dijkstra in 1966 [1]. A solution for $N > 2$ processes was proposed by Dijkstra in [4], although without a guarantee of bounded waiting for competing processes. Improved solutions for $N = 2$ and $N > 2$ were proposed, e.g., by Peterson in [5]. Such solutions will be studied in depth in this paper using our approach based on model checking [6–10]. Emerging results reconciliate with similar results reported in the literature [2,11,12]. Many other mutual exclusion algorithms investigated with the chosen approach were discussed in recent papers [13–15]. As an example, in [15], the method was applied to property checking of FCFS Lycklama and Hadzilacos's (LH) algorithm [16,17] and Burns and Lamport's solution [18,19], which is used as a component in LH.

The modeling and verification approach adopted in this paper is based on *timed automata* (TA) [20] and the Uppaal toolbox [8,21]. Being based on exhaustive model checking [9,10], the approach is not scalable in the number N of the involved processes. This is due to the well-known state explosion problems [22]. However, in a practical case, a significant number of processes can often be handled, e.g., $N = 5$ or a slightly greater value, sufficient for observing the rate with which the so-called *overtaking* factor varies vs. N. The overtaking factor represents the maximum number of times a competing process p will be by-passed by other competing processes before p eventually enters its critical section. Of course, the overtaking factor is directly related to the (hopefully) bounded waiting time of a competing process.

A key contribution of this paper consists in the possibility to adapt the formal TA-based model of a mutual exclusion algorithm for it to be studied under both the classic strong memory model, with the atomic read/write operations on the same memory cell, and the weak memory model (e.g., [23]) where, e.g., multiple read operations can occur simultaneously with a write operation, thus generating the *flickering* phenomenon. A reader process affected by flickering can achieve a non-deterministic value of a variable, which can influence its behavior. Considering the diffusion of multi-port memories [24] in widespread personal devices (e.g., cell phones), it is important to verify that a mutual exclusion solution remains RW-safe under weak memory.

Another contribution of this paper is an embedding of Peterson's solution for $N = 2$ processes in the context of the state-of-art and efficient *tournament tree* (TT) organization [25–27], where pairs of processes compete locally to establish the process that can move upward along a path of the binary tree. The TT solutions will also be studied under the strong and the weak memory model.

The rest of this paper is organized as follows. Section 2 describes the process model of a mutual exclusion algorithm. Section 3 discusses the fundamental properties of a mutual exclusion solution. Section 4 presents the adopted timed automata modeling language of Uppaal. Section 5 proposes a mapping of a mutual exclusion algorithm into a Uppaal model and discusses the complexity of model checking of an achieved model. Section 6 motivates some basic timed computational tree logic

(TCTL) queries of Uppaal for checking the correctness of the mutual exclusion model. In Section 7, the adaptations of a mutual exclusion model, so as to work with different memory models, are illustrated. Section 8 presents different Peterson's mutual exclusion algorithms, their formal modeling, and the assessment of their properties. For completeness, the mutual exclusion algorithm of Peterson and Fisher for two processes is also studied in depth. Section 9 is devoted to the modeling and analysis of mutual exclusion algorithms for $N > 2$ processes, according to a tournament tree organization, where Peterson's solution for two processes is used to arbitrate pairs of processes. Finally, Section 10 concludes the paper with an indication of ongoing and future work.

2. Process model

A mutual exclusion algorithm introduces a protocol (see Algorithm 1) that establishes the operations the processes have to execute when entering and competing (Entry) for the achievement of the permission to use the shared resource and for exiting (Exit) after finishing with the critical section. The protocol rests on a (hopefully) few shared communication variables whose initialization and management often mirrors an "ingenious design", which can be difficult to master by intuitive reasoning. As usual, a process is assumed to execute on a distinct processor (core) with a never-ending behavior. The process executes the *non-critical section* (NCS) when it has no interest in using the shared resource. For generality, the duration of the NCS code can also be infinite to model process termination. The Entry or competing part, and possibly also the Exit part, can introduce busy-waiting, unproductive or CPU-wasting cycles, when the conditions exist for waiting for some other processes to modify the shared communication variables. In Algorithm 1, unique process IDs are assumed to be from 1 to N, with $N \geq 2$.

Algorithm 1. Process model.

```
shared communication variables
Process(i):
local variables of process i
repeat
    NCS;
    Entry-part;
    CS;
    Exit-part;
forever
```

Many mutual exclusion algorithms purposely depend on shared communication variables, of which each one is assigned to a distinct process. Each process is responsible for changing the value of its own variable. Partner processes, though, can check all the protocol variables to regulate their behavior in the Entry/Exit parts. Such variables are called *exterior variables* by Kessels in [26]. However, other algorithms can also introduce some shared variables, which can be both read/written by all the involved processes.

3. Properties of a mutual exclusion algorithm

To be correct, a mutual exclusion algorithm has to fulfill both safety and liveness properties. A *safety property* aims to guarantee that some bad/hazardous execution state is never reached in the algorithm evolution. A *liveness property* wants to ensure a good state is eventually reached. A *bounded-liveness property*, in addition, is a liveness one where the good state is finally reached after a finite or bounded time.

The following properties characterize a correct mutual exclusion algorithm:

1) (*Safety*) Only one process at a time can enter its critical section (CS).
2) (*Liveness*) All processes must be ensured to eventually enter their CS.
3) (*Bounded liveness*) A competing process should get permission to enter its CS after a bounded time. The property is also referred to as the *absence of starvation*.
4) (*Safety*) All processes should never suffer any *deadlock* or *fatal lockout*.
5) (*Liveness*) A process in its NCS should not forbid another process to enter its CS.
6) (*Liveness*) No hypothesis should be made on the relative speed of processes.

Proving the properties of a mutual exclusion algorithm can be hard to accomplish by informal/intuitive reasoning. The difficulty is tied to the non-determinism or interleaved order of execution of the actions of the processes. Formal methods are normally advocated to establish the correctness of a mutual exclusion solution. In the literature, both *assertional methods* based on a *theorem prover* [27–29] or methods based on exhaustive *model checking* [13,14,30] are often used.

Assertional methods formalize a mutual exclusion algorithm and its data, deriving a state transition system from the algorithm. Assertions are then formulated about each state and the transitions to the next state. The usual informal reasoning of mathematics, with the help of a theorem prover as a proof-assistant, is then used to derive properties. The assertional methods with theorem provers have been successfully exploited in many cases. However, such methods are usually unable to deal with the timing issue, which can be important for property assessment.

Model checking methods formalize a mutual exclusion solution, e.g., by timed automata (TA) [20]. TA have an intuitive graphical version that facilitates the modeler's understanding. From the algorithm/model, the underlying state transition system (often called the model state graph) is automatically derived. The verifier navigates the state graph for property checking. All the execution paths originating from the initial state and corresponding to the non-determinism and partial order/interleaving of process actions are systematically analyzed. In addition, the use of TA makes it possible to study the influence of time on the behavior of an algorithm. The use of a model-checking tool is practically limited by the *state explosion problem* [22]. The problem naturally arises when the number N of the processes and the corresponding (global and local) data variables increase. However, the behavior of an algorithm can often be studied for non-trivial values of N, sufficient for extracting the *overtaking bound*. Such a bound is the number of times a competing process is bypassed by other competing processes until the former process finally enters its critical section (see also point 3 in the list of properties).

This paper adopts the model checking method and proposes an approach based on the TA of the Uppaal toolbox [7]. Uppaal supports a subset of the TCTL temporal logic queries. Uppaal was chosen because it offers not only the exhaustive verifier but also a symbolic simulator. A mutual exclusion model can be animated in the symbolic simulator for debugging purposes, thus contributing to its understanding. In addition, a diagnostic trace (counter-example), e.g., generated by a not-satisfied query, can be carefully examined in the simulator, to infer the causes for a model’s bad behavior.

4. Modeling language

The following considers the basic features of the Uppaal concurrent modeling language [8,21], based on a high-level version of the timed automata (TA) [20]. A model is a network of process instances (TA), communicating one to another by unicast or broadcast channels. Processes share global data declarations but can also have local data. Integers, Booleans, and arrays of these primitive data are supported. Clock variables can be introduced for dealing with time. A clock can be reset. A clock measures the (relative) time elapsed from its last rest. All the clocks of a model grow at the same rate, which mirrors the advancement of global time. Process behavior is expressed by a finite automaton made of *locations* (local states denoted by circles) and edges (location transitions). Edges are annotated by *guarded commands*.

A command has three basic optional attributes: a *guard* (boolean expression, green-colored in a graphical model as in Figure 5), a *channel synchronization* (azure-colored), and an *update* (blue-colored). All these attributes are lacking in a *spontaneous* edge. The guard (true if not specified) must be true for the edge to be taken. But, as in Petri nets [10,31], an enabled edge can be taken but it is not forced to be taken soon.

A channel synchronization is represented by a sending (indicated by the symbol!) or receiving (?) operation on a channel. Unicast channel synchronization expresses a *rendezvous*: both the sender and the receiver processes must be ready for them to synchronize. Otherwise, the first arriving process will wait for the partner. After the rendezvous, both processes resume their concurrent evolution. A broadcast channel is useful for asynchronous communications. The sender can synchronize with a group of 0, 1, or multiple receivers. In no case, though, the sender gets blocked; it immediately prosecutes even when no receiver is ready to accept the synchronization.

The update of a command is an ordered list of variable assignments and clock resets. Uppaal locations can be *normal* or *urgent*. In a normal location, the process can remain for an arbitrary (also infinite) time. Model progress can be improved by attaching an *invariant* (boolean expression, often based on a clock constraint) to a normal location. The process can stay in the location provided the invariant continues to be true. Otherwise, the location has to be abandoned. An urgent location (see location with a U in Figure 3) has to be exited without time passage. However, among a group of urgent locations, in multiple processes, their exit follows a non-deterministic order. A particular case of an urgent location is the committed location, which has also to be abandoned without time passage. Uppaal, though, gives priority to committed locations over urgent locations. Also, in a group of committed locations, their exit occurs non-deterministically.

Another way to force the progress of a Uppaal model rests on the usage of *urgent channels*. An enabled rendezvous on an urgent unicast channel has to be taken immediately, otherwise a deadlock occurs. Similarly, an enabled and urgent broadcast synchronization always occurs immediately, independently of the number of receivers. Channel synchronizations do not carry any data. However, the transmission of parameters from the sender to the receiver(s), when needed, can be assisted by some global data variables.

The concurrent language of Uppaal recognizes the guarded command as the fundamental *atomic action*. Model evolution is ensured by executing actions (or joint actions when a channel synchronization is involved) *one-at-a-time*, possibly at the same time. This way, the order of action execution can naturally express the partial order and interleaving of a typical concurrent system.

Uppaal offers the use of the symbolic simulator for a preliminary investigation of the model behavior. After that, the model state graph, hopefully finite, can be built, whose nodes reflect all the possible execution states of the modeled system. Table 1 reports the basic TCTL (timed computational tree logic) queries [8], which can be issued for state graph navigation. Each query relies on an operator and a formula (denoted by φ, ψ in Table 1), which can be a state predicate (the fact that a process is in a particular location) or be based on data and clock constraints. A built-in predicate is the *deadlock* keyword. It can be used to check if there are deadlocked states in the state graph.

Table 1. Basic TCTL queries of Uppaal [8].

TCTL query	Meaning
E <> φ	*Possibly* φ, i.e., a state exists in the state graph where φ holds.
A[] φ	*Invariantly* φ, equivalent to: not E <> not φ.
E[] φ	*Potentially always* φ, i.e., a state path exists over which φ holds in every path state.
A <> φ	*Always eventually* φ, equivalent to: not E[] not φ.
φ − −> ψ	φ *always leads-to* ψ, equivalent to: A[] (φ imply A <> ψ).

Uppaal distinguishes from similar tools for the compact data structures adopted for representing the state graph in memory and the efficient algorithms implemented for the state graph navigation. For further details, the reader is referred to [21].

5. Transforming a mutual exclusion algorithm into a Uppaal model

The process model of Algorithm 1 can be mapped onto Uppaal by introducing a parameterized Process automaton, with the only parameter *const pid i*, where *i* is the unique identifier assigned to the process instance. *pid* is the type range of all the possible process IDs. The following are some basic global declarations for the processes:

```
const int N = 2; //example
typedef int[1, N] pid;
clock x[pid]; //one clock per Process instance
const pid tp=1; //target process example
```

5.1. Basic transformation rules

The Process automaton associates locations to each action/instruction of the algorithm. NCS (see Figure 1) is modeled as a normal location, in which the process can stay an arbitrary time, also infinite. NCS is also the initial location, and it is reached after executing the Exit part of the process. The exit edge from NCS is a spontaneous edge.

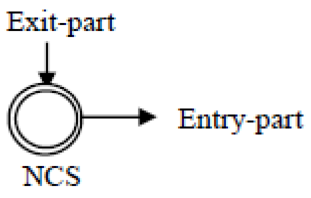


Figure 1. Modeling the non-critical section (NCS).

CS (see Figure 2) is modeled as a location where the automaton can remain exactly for 1 time unit, measured by the process clock $x[i]$. This provision is key to determine the overtaking factor (see below for details).

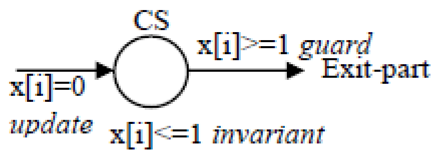


Figure 2. Modeling the critical section (CS).

Any other action can be modeled by an urgent location (see Figure 3); thus, it consumes no time. The use of urgent locations allows us to reproduce, in a natural way, the non-deterministic order of execution of the process actions. However, to improve the model checker operation (and thus to reduce the partial order degree of the nodes of the state graph), a busy-waiting situation (see the location BW in Figure 3) is realized by a normal location. An immediate exit is commanded from a BW location as soon as the awake condition is satisfied. In other terms, whereas in the physical case of a process running on a separate processor/core, the active busy-waiting, although wasting corresponding CPU cycles, can be tolerated, for model checking it is convenient to keep a process in a normal location for the duration of the busy-waiting. To achieve the immediate abandoning of the busy-waiting location, an urgent and broadcast channel *synch* is used. Provided the awake condition (*try*() guard) holds, it is sufficient to command an output synchronization *synch*! where no receiver is expected to capture the synchronization.

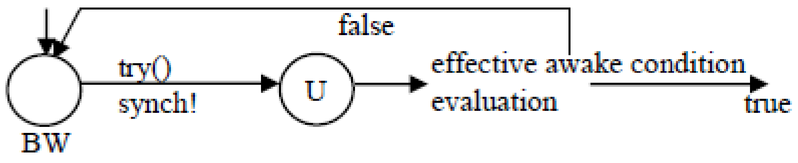


Figure 3. Modeling a busy-waiting situation by a BW location.

There is a subtle point in Figure 3 that deserves further discussion. Whatever the adopted memory model (see Section 7), it is clear that a process cannot simultaneously access multiple shared communication variables. Therefore, the *try*() function, which checks the awake condition, except for very simple cases, cannot be evaluated in a single step (a Uppaal guard). Here, the modeler is assumed to check optimistically that the awake condition holds (*try*() returns true) just as a signal to possibly exit the BW. But after exiting BW, the process has to evaluate the effective and whole awake condition by accessing and logically composing the single pieces of the condition. Would it emerge, at any stage during the evaluation, that the awake condition does not hold, the BW is immediately re-entered. It is worth noting that evaluating the single components of a complex awake condition is naturally affected by the non-determinism that occurs in the physical realization of the mutual exclusion algorithm. In the case of simple awake conditions, the *try*() function can be avoided (see Figure 5).

5.2. Predicting the overtaking factor

The following discusses some aspects of the overtaking factor prediction. Considering that, normally, all the processes of a system are identical from the point of view of the protocol of mutual exclusion, one process can be chosen upon which the overtaking bound is detected. Such a process is indicated as the *target process* tp and can be any process (by default, $tp = 1$). The clock $x[tp]$ is reset as soon as tp starts competing, that is, it exits from NCS and begins executing the Entry part of the protocol. Of course, as other competing processes precede the competing tp and enter their CS, the clock $x[tp]$ gets incremented by 1. As a consequence, when tp achieves, finally, the permission to enter its CS, the maximal value of $x[tp]$ can be checked on a location immediately preceding the CS, which provides the overtaking bound. As a final remark, it is useful to note that the time spent by a process into NCS does not alter the evaluation of the overtaking factor. In fact, if such a bound is Δ and dw is the dwell time in NCS, in the case $dw < \Delta$, the process actively participates in the current competition, which contributes to the definition of the Δ amount. In the case dw $\geq \Delta$, the arrival of the process will be part of a subsequent competition.

5.3. Model checking complexity of a mutual exclusion model

The complexity (see also [21]) of a mutual exclusion model directly depends on the amount of data variables (global and local) required by the process models and their internal structure (number of locations, degree of non-determinism). In addition, the complexity is exponential in the number of clocks used. In reality, it is the number of *active clocks*, at any time, that is important—not the absolute number of introduced clocks, which coincides with the number N of processes. An active clock is one that is reset and then actively used to constrain an invariant and/or a guard (see Figure 2). From this point of view, if the algorithm ensures, as expected, the mutual exclusion property, only one process at a time can be in its CS and thus have its clock active. All of this, paired with the competing target process, permits us to conclude that the effective number of active clocks, at any moment, is 2 and not N. Another important factor affecting the complexity is the so-called *partial order* (or non-determinism) of the nodes of the state graph, that is, the number of exiting transitions from each node. Modeling busy-waiting by normal locations from which an exit occurs only when the data variables are potentially changed in such a way as to terminate the busy-waiting (see the $try()$ function in Figure 3) was chosen to reduce the amount of the partial order in the state graph nodes. However, the partial order increases as the flickering phenomenon is planned in a model as a consequence of the adoption of the weak memory model (see Section 7).

6. Checking the correctness of a mutual exclusion algorithm

The correctness of a mutual exclusion algorithm transformed into a Uppaal model can be assessed by TCTL queries corresponding to the properties discussed in Section 3, as in Table 2, where the Uppaal predefined functions exist $(\ldots)$ and sum $(\ldots)$ are also used.

The query of property 1 in Table 2 checks that in all the states of the state graph, the number of processes simultaneously found in their CS is always less than or equal to 1. A different but equivalent way to check the same property would be the following:

A[] exists(i:pid)Process(i).CS imply !exists(j:pid)Process(j).CS && i!=j.

The sup (suprema) query for property 3 is a shortcut for the repeated use of the following query with tentative values for the $overtaking\_bound$:

A[] Process(tp).L imply x[tp]<=$overtaking\_bound$.

The $overtaking\_bound$ is expected to converge to the lowest upper bound, which satisfies the A[] query. In other terms, the query should not be satisfied by the $overtaking\_bound - 1$. In the query 5 of Table 2, the target process tp can be replaced by any other process id.

Table 2. TCTL queries for correctness assessment of a mutual exclusion algorithm.

Property	TCTL query	Expected result
(1). (*safety*) Only one process at a time can enter its critical section (CS).	A[] (sum(i:pid)Process(i).CS)<=1	satisfied
(2). (*liveness*) All the processes must be ensured to eventually enter their CS.	for each i of type pid: E<> Process(i).CS	satisfied
(3). (*bounded liveness*) A competing process should get permission to enter its CS after a bounded time.	let tp be the observed target process and L be a location immediately preceding CS: sup{ Process(tp).L } : x[tp]	finite integer bound
(4). (*safety*) All the processes should never suffer from any *deadlock* or *fatal lockout*.	A[] !deadlock	satisfied
(5). (*liveness*) A process in its NCS should not forbid another process to enter its CS.	let tp be the observed target process: E<> Process(tp).NCS && exists(j:pid) Process(j).CS	satisfied

7. The memory models

This paper considers only two memory models—the classic strong memory model and the today widespread weak memory model. In the strong model, the read/write operations on the same memory cell are atomic or indivisible. Reflecting the strong memory model in a mutual exclusion algorithm transformed into a Uppaal model is straightforward due to the Uppaal concurrency model where guarded commands (actions) are intrinsically atomic. It is worth recalling that no unicast channel is used and the broadcast and urgent channel $synch$ always has no waiting receiver.

More challenging is the weak memory model, where read/write operations can occur simultaneously. Two situations can be distinguished. The first one relates to multiple read operations that occur during one write operation. In these cases, the *flickering* phenomenon occurs [23]. Therefore, each reader process achieves a non-deterministic value due to flickering. Whereas in [25] it was assumed that a reader could achieve either the old or the new value, more recently [23], it was pointed out that flickering can really return a value non-deterministically belonging to the type of the variable. Of course, a read that follows the write operation will return the correct new value.

Modeling the flickering as assumed in [25] is directly supported by Uppaal modeling. In fact, due to non-determinism, the read can be issued before or after the write and the two possible results are ensured. To model general flickering, the schema suggested in Figure 4 can be used. First, var is assigned a value non-deterministically chosen in the set of values of the type of var. Then, the effective value is assigned to var. As a consequence, for non-determinism, a reader can achieve the flickered value instead of the true value.

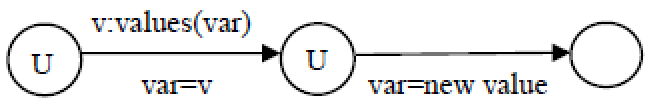


Figure 4. Modeling flickering before a write operation on var.

The flickering schema suggested in Figure 4 can be realized in a Uppaal model by using the fourth attribute of a guarded command, which allows to specify a *non-deterministic selection* (yellow-colored) as indicated in Figure 4. The proposed schema introduces a temporary variable like v, whose lifetime is the belonging command, and to assign to var the non-deterministic value selected for v.

Obviously, the introduction of flickering augments the degree of non-determinism in the Uppaal model, which tends to complicate the behavior and verification of a mutual exclusion algorithm. The algorithm could also lose its properties. An algorithm that keeps its properties despite flickering is said to be RW-safe.

A more complex case occurs when multiple write operations are permitted to execute, simultaneously, on the same variable. In this case, the resultant value of the written variable gets *scrambled*, that is, there is no guarantee that the value belongs to the type of the variable. As in [23], in this paper scrambling is not admitted, that is, each write is assumed to occur in isolation with the help of some underlying hardware mechanism (*fencing*). Anyway, scrambling will be signaled in a model location where it can logically occur. From the point of view of Uppaal modeling, scrambling cannot occur because the write operations are necessarily executed one at a time and in any order.

8. Peterson's mutual exclusion algorithms

The Peterson's algorithm for $N = 2$ and $N > 2$ processes in [5] were proposed as an improvement of previous solutions [1,4]. These algorithms will be analyzed by first transforming them into Uppaal models. The stable Uppaal version 5.0 [21] is used for the experiments on a Win11 Pro desktop platform, Dell XPS 8940, Intel i7-10700 (8 physical cores), CPU@2.90 GHz, and 32GB RAM.

8.1. Peterson's solution for $N = 2$ processes

An elegant and simpler algorithm than Dekker's solution [1] was proposed by Peterson in [5]. Here, the algorithm is reproduced in Algorithm 2, with a simple renaming of the shared variables, also adopted in subsequent solutions. The algorithm depends on the following shared communication variables:

bool flag[pid]={false,false}, pid turn=*immaterial initialization*.

Variables $flag[1]$ and $flag[2]$ are exterior [26], $turn$ not. The $const\ pid\ j = 3 - i$ identifies the partner of the process i. When it starts competing, process i assigns true to *its* variable $flag[i]$. Then, it gives the $turn$ to the partner j. Process i remains in busy-waiting until either the partner is not interested in the CS ($flag[j] = false$) or the $turn$ is assigned to i ($turn! = j$). After the CS, the process i resets its $flag[i]$.

The critical situation in Algorithm 2 is when both processes ask simultaneously to access their CS (see also Figure 6). In this case, the process i that last writes a value on to $turn$ is decisive and permits the partner process j to avoid the busy-waiting and gain the critical section. Process i, instead, enters its busy-waiting. The write to $turn$ needs to be fenced in a practical implementation. Figure 5 shows a Uppaal model corresponding to Algorithm 2. The $reset(i)$ helper function resets the clock $x[i]$ when i is the target process tp.

Algorithm 2. Peterson’s solution for 2 processes.

```
shared variables: bool flag[pid], pid turn
Process(i):
local variables: const pid j=3-i;
repeat
    NCS;
    flag[i]=true;
    turn=j;
    wait-until (!flag[j] or turn!=j); //busy-waiting
    CS;
    flag[i]=false;
forever
```

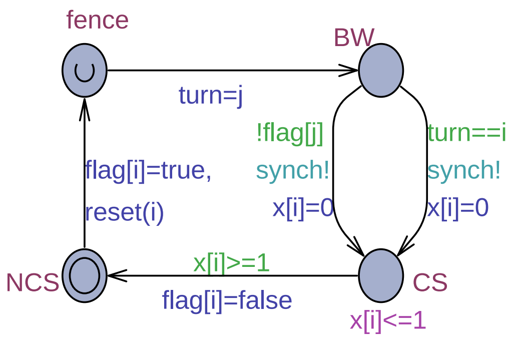
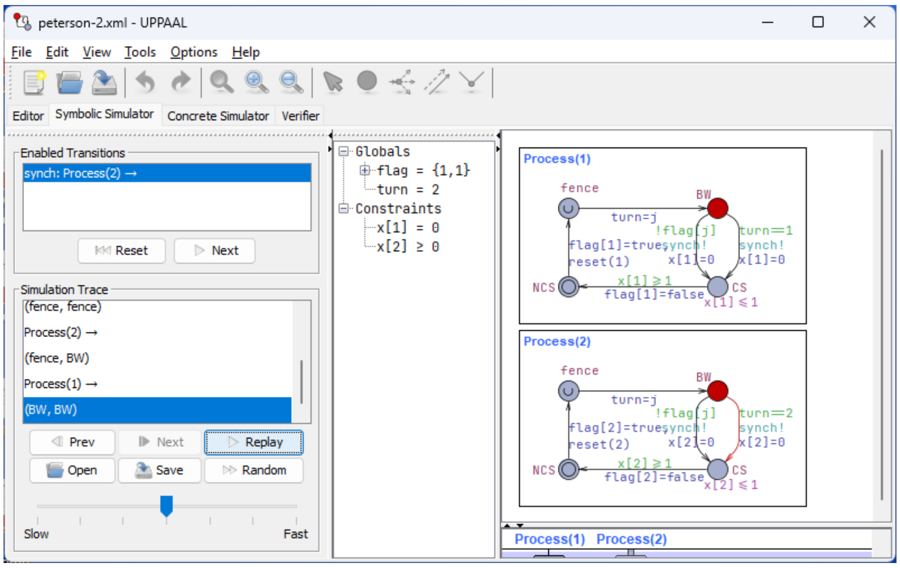


Figure 5. A Uppaal model for Algorithm 2.

As one can see in Figure 5, no need exists for a $try()$ function (see Section 5) because of the simple disjunction of the awake condition.

The behavior of the model in Figure 5 can be animated in the symbolic simulator (see a snapshot in Figure 6), where the modeler can explicitly examine the critical behavior when both processes require access to the resource. Uppaal fences automatically the writing to the $turn$ variable. Other simple scenarios, when only one process asks to enter its CS, can be studied as well, thus confirming intuitively the *correct* behavior of Peterson’s algorithm for $N = 2$ processes.



synch: Process(2) →

Simulation Trace:
(fence, fence)
Process(2) →
(fence, BW)
Process(1) →
(BW, BW)

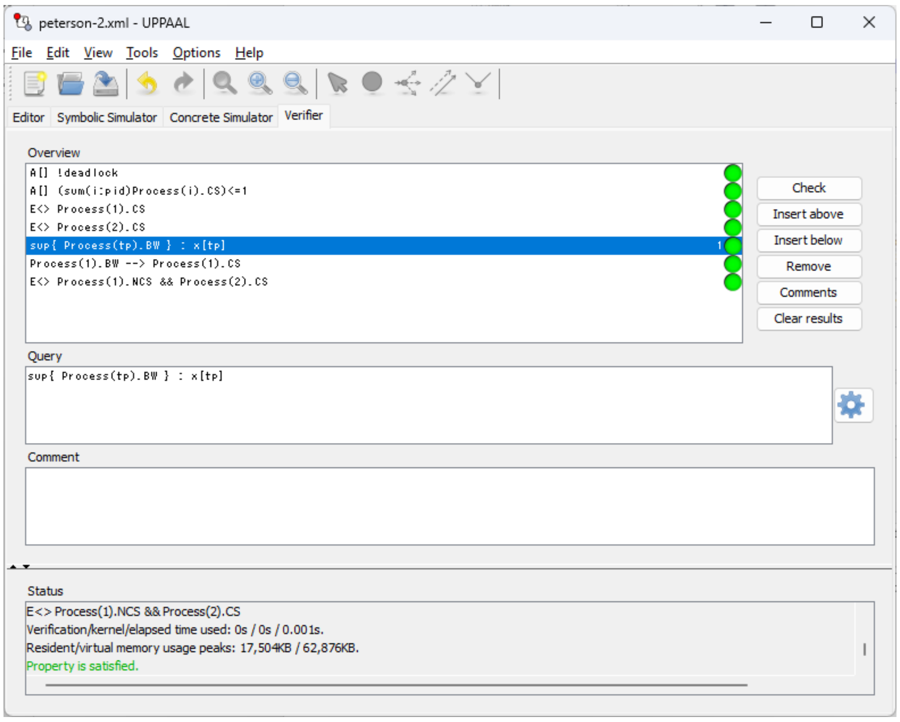
```
Globals
  flag = {1,1}
  turn = 2
Constraints
  x[1] = 0
  x[2] ≥ 0
```

Process(1): fence, BW, turn=j, !flag[j], synch!, x[1]=0, turn==1, synch!, x[1]=0, flag[1]=true, reset(1), NCS, x[1] ≥ 1, flag[1]=false, CS, x[1] ≤ 1

Process(2): fence, BW, turn=j, !flag[j], synch!, x[2]=0, turn==2, synch!, x[2]=0, flag[2]=true, reset(2), NCS, x[2] ≥ 1, flag[2]=false, CS, x[2] ≤ 1

Figure 6. A snapshot of the model in Figure 5 in the Uppaal symbolic simulator.

Formal correctness is ensured by model checking the model in Figure 5 using the queries of Table 2, which are all satisfied (see Figure 7). In particular, query 3 about the overtaking factor suggests the bound is 1. That is, a competing process waits, at most, for the partner to exit its CS; after that, it can enter its own CS.



```
A[] !deadlock                                    ●
A[] (sum(i:pid)Process(i).CS)<=1                 ●
E<> Process(1).CS                                ●
E<> Process(2).CS                                ●
sup{ Process(tp).BW } : x[tp]                   1●
Process(1).BW --> Process(1).CS                  ●
E<> Process(1).NCS && Process(2).CS              ●
```

```
sup{ Process(tp).BW } : x[tp]
```

```
E<> Process(1).NCS && Process(2).CS
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 17,504KB / 62,876KB.
Property is satisfied.
```

Figure 7. A snapshot of the model in Figure 5 showing that the overtaking factor is 1.

8.2. Peterson's solution for $N = 2$ *processes under the weak memory model*

The model in Figure 5 assumes, by default, the strong or atomic memory model, where the read/write operations on a variable are atomic. Figure 8 shows the model adapted for working with the weak memory model.

Exterior variables $flag[i]$ are written by the owner process and consulted by the partner process. The $turn$ variable, that is not exterior, can be written by one process and checked by the other, or possibly it can be written simultaneously by the two processes. Flickering is modeled by a non-deterministic selection of a value belonging to the variable type (yellow colored in Figure 8). When a process is in the fence location, the $turn$ variable can be affected by simultaneous read/write operations. To protect the turn from simultaneous write operations, it needs to be fenced (see Section 7) as signaled in Figure 8.

Model-checking the model in Figure 8 confirmed it possesses exactly the same properties as the basic model in Figure 5, with the same overtaking bound of 1. Therefore, Peterson's solution for two processes emerged to be RW-safe also under flickering and weak memory.

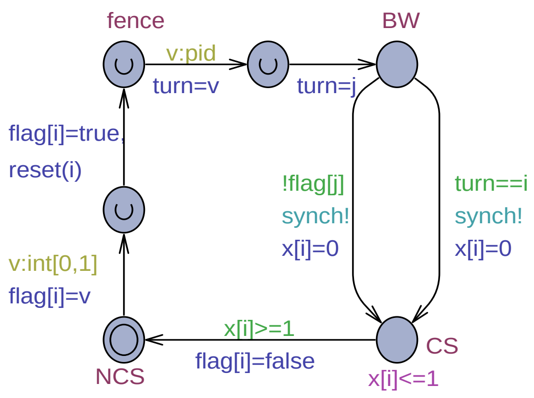


Figure 8. Peterson's model of Figure 5 adapted for working with weak memory.

8.3. Peterson's solution for $N > 2$ processes

Peterson's solution for 2 processes was generalized in [5] to $N > 2$ processes using the metaphor of "climbing a ladder". The ladder admits $L = N - 1$ steps or levels, denoted by the numbers from 1 to L. The process that first arrives at the last level gets permission to enter its critical section. A not-competing process is, by convention, at level 0 (out of the ladder). A competing process residing at the step j, will move to the next step provided all the remaining processes are at levels less than j. The process also moves next when a competitor reaches the same level j. The new algorithm is reported in Algorithm 3. It relies on the following shared communication variables:

int[0,L] flag[pid]; pid turn[1..L],

whose initialization is immaterial and can exploit default values. The $flag[]$ variables are exterior variables, controlled separately by the processes. The $turn[]$ variables are not exterior. Variable $turn[j]$ stores the process id who is occupying the level j.

Algorithm 3. Peterson's solution for $N > 2$ processes.

```
shared communication variables: flag[], turn[]
Process(i):
local variable int j;
repeat
    NCS;
    for j=1 step 1 until N-1 {
        flag[i]=j; //process i moves to step j
        turn[j]=i; //step j is occupied by process i
        wait-until( (∀ k!=i: flag[k]<j)) or turn[j]!=i );
    }
    CS;
    flag[i]=0;
forever
```

Algorithm 3 is transformed into Uppaal, as shown in Figure 9. The write operation to $turn[j]$ needs to be fenced under weak memory. In Figure 9, to improve the model checking, the local variables j and k are reset to their default value as soon as they cease to be actively used.

The $try()$ function used in the model in Figure 9 is reported in Algorithm 4. It checks optimistically if the process in the busy-waiting location BW can possibly enter its CS. The function assumes to check simultaneously multiple shared communication variables. As a consequence, in the case $try()$ returns true, BW is tentatively abandoned, and the verification of the awake condition is carefully evaluated by inspecting the involved shared variables one at a time. As soon as the condition is found to be false, the BW location is re-entered. Would the entire condition emerge to be true, either the next iteration of the FOR loop is started or the CS is entered.

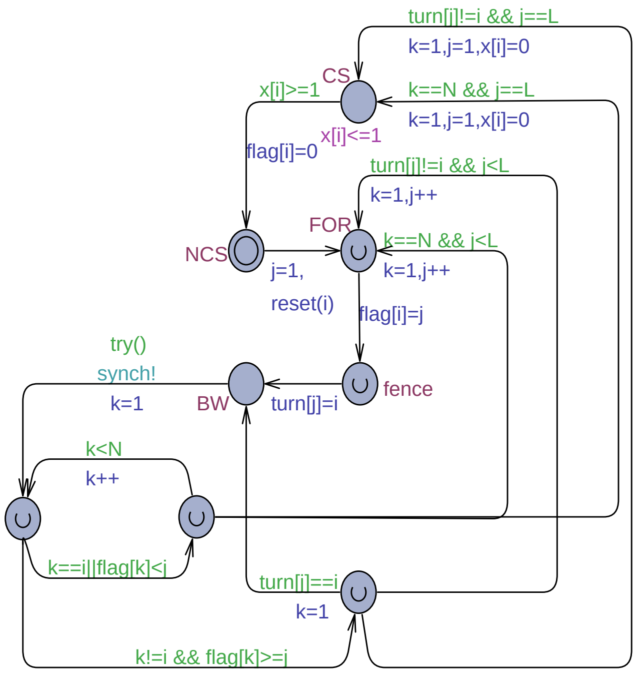


Figure 9. Uppaal model for Peterson's algorithm of Algorithm 3.

Algorithm 4. The try() function of the model in Figure 9.

```
bool try(){
    return forall(k:pid)(k==i||flag[k]<j) || turn[j]!=i;
}//try
```

The model of Figure 9 satisfies all the queries of Table 2, and its overtaking factor is bounded. Therefore, Algorithm 3 is a correct mutual exclusion algorithm. Also, the following liveness property was found satisfied, for any process:

Process(1).BW --> Process(1).CS.

The query assesses that a process in the location BW inevitably will reach its CS. Satisfaction of this query excludes starvation for a competing process. The property also holds from the FOR location. The overtaking factor was observed for various values of N starting from 2. Results are collected in Table 3.

With $N = 6$, the state graph explodes. With $N = 5$, the sup query (third property of Table 2) terminates after 242 s with a memory usage of about 4.5 GB. Table 3 clearly shows that $ov = \frac{1}{2}N * (N - 1)$, that is, ov has a quadratic dependency on the number N of the processes. This experimental result complies with the result reported in [2]. However, papers [11,12] suggested a linear dependency: $ov = N - 1$.

Table 3. The overtaking bound (ov) vs. N for the generalized Peterson's solution.

N	ov
2	1
3	3
4	6
5	10

In order to (possibly) justify the two kinds of results for the overtaking factor, another series of experiments was carried out on the model of Figure 9. In Figure 9, the exiting from NCS was correctly modeled to occur at any time, from 0 to ∞. Having assessed that the algorithm/model behaves correctly even when the process terminates within the NCS, it was decided to explore the model when NCS is turned into an urgent location. In this case, a process exiting from its CS immediately re-enters and starts competing. Table 4 shows the collected overtaking bound in this new situation, where it emerges: $ov = N - 1$. It is important to note that all the other properties of a mutual exclusion algorithm continue to be satisfied.

The use of NCS as an urgent location reduces the amount of the execution paths to be verified by the model checker. The sup query for the case $N = 5$ now terminates after 22 s with a memory usage of 477 MB. But for $N = 6$, the state graph again explodes.

Table 4. The overtaking bound (ov) vs. N when NCS is made urgent in Figure 9.

N	ov
2	1
3	2
4	3
5	4

The experience with the generalized Peterson's algorithm confirmed the usefulness of studying the effects of timing on a mutual exclusion solution. The timing of the NCS location directly influences the liveness of processes, of which the overtaking bound depends. The proposed approach based on Uppaal is unique in demonstrating these subtle behavioral aspects. Timing and process liveness will also be observed in subsequent models in this paper.

8.4. Peterson's solution for $N > 2$ processes under weak memory

The new model with flickering and fencing is shown in Figure 10.

Model-checking the new model confirms it is RW-safe, that is, it is correct from the point of view of all the properties of mutual exclusion exactly as in the case of strong memory and exhibits the same overtaking bound in the two scenarios of NCS. However, the adapted model of Figure 10 is more difficult for exhaustive verification. In fact, the state graph now explodes earlier at $N = 5$. Using the NCS with arbitrary dwell time as in Figure 9, the sup query for $N = 4$ terminates after 630 s with a memory usage of about 10 GB. Using NCS urgent instead, always in the case of $N = 4$, the sup query terminates after 77 s with a memory usage of 1.5 GB.

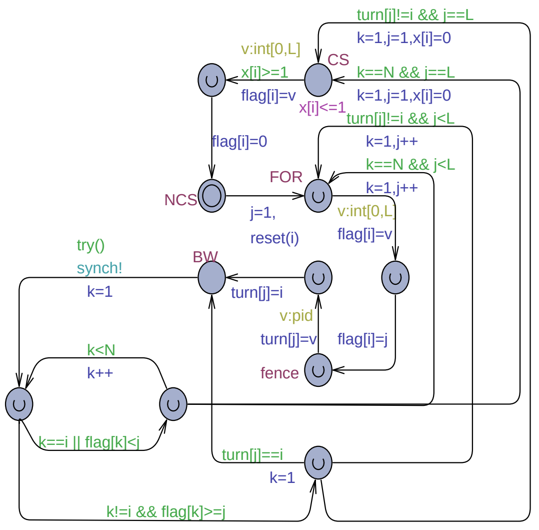


Figure 10. The Peterson's algorithm for N processes adapted for weak memory.

8.5. Peterson and Fisher's solution for $N = 2$ *processes*

This algorithm was proposed in [25]. Our interest in it was motivated by the fact that the solution relies exclusively on exterior variables [26]. Each process writes on its own variable, which is then consulted by the partner. The algorithm represents a not symmetric solution, because processes 1 and 2 do not execute the same Entry/Exit code (see Algorithm 5). In terms of Uppaal declarations, the global data can be defined as follows:

```
const int N=2;
typedef int[1,N] pid;
const int U=-1, F=0, T=1;
typedef int[U,T] value;
value flag[pid]={U,U}; //shared communication variables.
```

The exterior variables $flag[]$ are initialized with the U (undefined) value. To simplify the expression of the process code, Algorithm 5 uses the local functions $step1()$, $step2()$ and $go()$, whose behavior depends on the process's identifier i. Considering that only a global variable at a time can be inspected/modified, a local variable q is introduced, which is assigned the $flag[j]$ value of the partner process just before a step. Uppaal definitions of the local functions are clarified in Algorithm 6. The $try()$ function is identical to the $go()$ function and optimistically checks both $flag[i]$ and $flag[j]$.

Peterson and Fisher's algorithm is difficult to grasp intuitively. Authors in [25] have analyzed the algorithm in an informal way. In this work, though, the Uppaal toolbox is used for the automated reasoning. Figure 11 shows a model corresponding to Algorithm 5. It was first intuitively animated through the symbolic simulator, receiving positive feedback that it *can* be a correct mutual exclusion solution. For example, no lockout occurred during the animation, and both processes can enter, one at a time, their CS.

Algorithm 5. Peterson & Fisher process's abstract code.

```
shared communication variables: value flag[pid]
Process(i):
local variables: const int j=3-i; value q;
local functions step1(), step(2), go();
repeat
    NCS;
    q=flag[j];
    step1();
    q=flag[j];
    step2();
    await-until( go() );
    CS;
    flag[i]=U;
forever
```

Algorithm 6. Definition of functions used in Algorithm 5.

```
void step1(){                 void step2(){
  if( i==1 ){                   if( i==1 ){
    if( q!=F ) flag[i]=T;         if( q!=U ) flag[i]=q;     bool go(){
    else flag[i]=F;             }                             if( i==1 )
  }                             else{                           return q!=flag[i];
  else{                           if( q==T ) flag[i]=F;       return q==U || q==flag[i];
    if( q==U || q==F )            else if( q==F )           }//go
      flag[i]=T;                    flag[i]=T;
    else flag[i]=F;             }
  }                           }//step2
}//step1
```

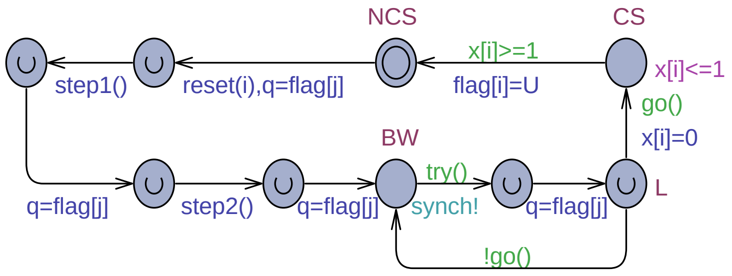


Figure 11. Uppaal model of Peterson and Fisher's algorithm for $N = 2$ processes.

Then, the model was exhaustively verified through the model checker. All the mutual exclusion properties were found to be satisfied, with an overtaking bound of 1.

The Uppaal model in Figure 11 automatically meets the simplified flickering assumption stated in [25], where reading a shared variable returns either the old or the new value. Figure 12 represents an adaptation of Figure 11 according to the more complex flickering (see [23]) phenomenon adopted in this paper. No fencing is required. Basically, since both the $step1()$ and $step2()$ redefine $flag[i]$

of process i, before $step1()$ and before $step2()$, flickering is used to affect non-deterministically the value of $flag[i]$. Only in the case of process 1 and $q == U$, flickering is avoided because $flag[i]$ gets not changed.

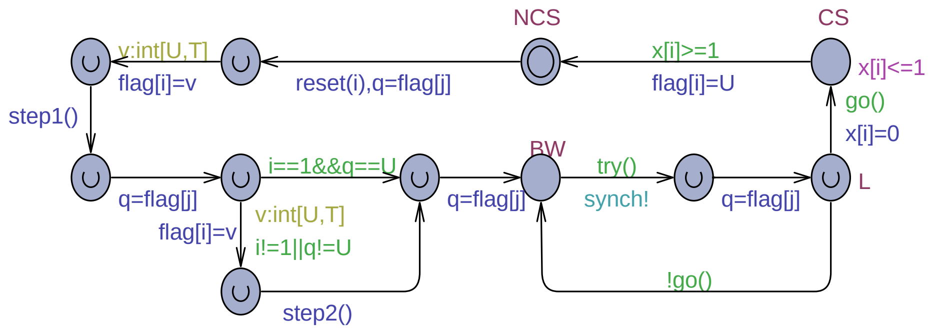
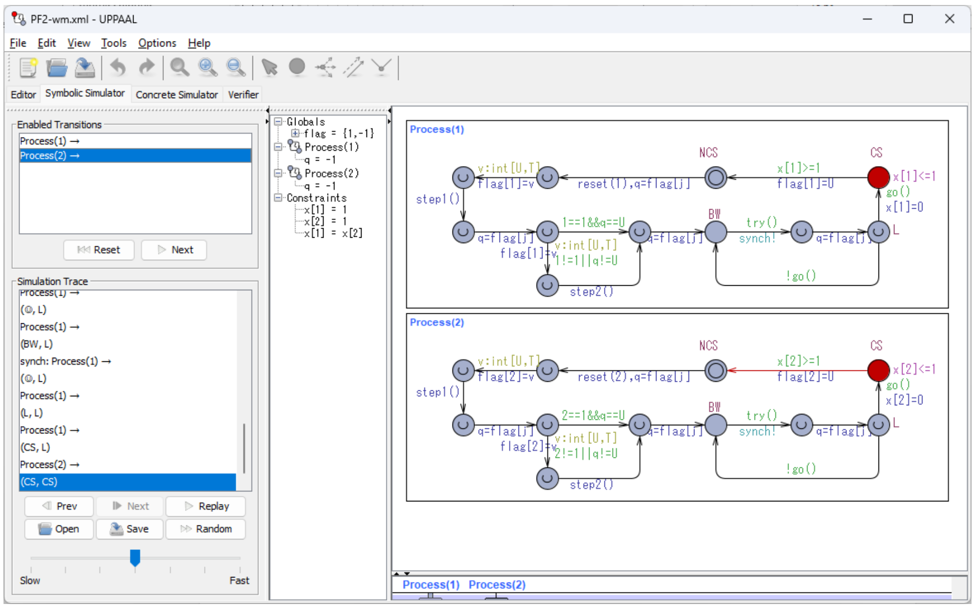


Figure 12. Peterson and Fisher's model adapted to flickering and weak memory.

The model in Figure 12 is without deadlocks and satisfies some liveness properties. Unfortunately, though, the new algorithm does not forbid the two processes to be simultaneously in their CS (!) (see Figure 13). On the left of Figure 13, the symbolic simulator was fed by a diagnostic trace, that is, a sequence of events that brings the two processes in their CS at the same time. Basically, due to flickering and non-determinism, the two processes can arrive at their L location with the local variable $q = U$ and with $flag[1] = T$ and $flag[2] = U$. In this situation, the $go()$ function (see Algorithm 5) will enable both processes to enter the CS location. Flickering here implies that the fundamental mutual exclusion property is lost.



Enabled Transitions
Process(1) →
Process(2) →

Simulation Trace
```
Process(1) →
(◎, L)
Process(1) →
(BW, L)
synch: Process(1) →
(◎, L)
Process(1) →
(L, L)
Process(1) →
(CS, L)
Process(2) →
(CS, CS)
```

```
Globals
  flag = {1,-1}
Process(1)
  q = -1
Process(2)
  q = -1
Constraints
  x[1] = 1
  x[2] = 1
  x[1] = x[2]
```

Figure 13. A snapshot showing the model of Figure 12 no longer satisfies mutual exclusion.

9. State-of-art solutions based on a tournament tree

As discussed in [27], nowadays there is an interest in embedding a mutual exclusion solution for two processes as the arbitration algorithm, which manages couples of processes (siblings) at different nodes of a binary tournament tree (TT). The winning rocess of an arbitration moves up to the ancestor node, where a new arbitration can take place and so forth. The process that first arrives at the root node has permission to enter its critical section. It is assumed that processes will occupy leaf nodes at their arrival (start of competitions). Then, concurrent arbitrations permit processes to move upward along a path toward the root. The interest in developing TT-based mutual exclusion algorithms for $N \geq 2$ processes rests on the fact that such solutions are state-of-the-art, that is, they are standard and efficient. Unfortunately, some TT-based solutions [27,13], especially when the weak memory model is considered, can have an unbounded overtaking factor (e.g., [15]) or the resultant algorithm can lose its fundamental mutual exclusion property.

9.1. Implementation framework

In this work, the binary TT is realized on a linear array in the same way the tree is used for supporting the heap sorting. In the proposed realization, the leaves reside only on the (possibly partially occupied) last level of the tree. They are used as entry points for processes. All the remaining intermediate nodes, including the root node, behave as arbitration nodes. The implementation deterministically associates process IDs to leaf nodes. To clarify the organization of the shared global variables, the following considers a TT definition that embeds Peterson's solution for $N = 2$ processes (see Section 8.1) as the basic arbitration strategy.

The bool $flag[]$ array represents all the TT nodes, numbered from 1 (the root) to $\left(2^{\lceil \log_2 N \rceil} - 1\right) + N$. Leaf nodes have the indexes from $2^{\lceil \log_2 N \rceil}$ to $2^{\lceil \log_2 N \rceil} + N - 1$. When a process is in or it has passed through the node $flag[j]$, $flag[j]$ remembers the passage with its value being set to true. To support Peterson's solution for 2 processes, a second array $turn[]$ is introduced, which has one slot for each intermediate node, including the root. At its arrival, process i enters *its* leaf node $\left(2^{\lceil \log_2 N \rceil} - 1\right) + i$. As the process moves along its path, the process is effectively represented by its latest position j (node) occupied in the $flag[]$ array. At an arbitration node, process j competes with its sibling, which is the process, if there is one, that occupies the $flag[]$ in a position s such that j and s have the same ancestor ($j/2 = s/2$). This ancestor node provides the $turn[j/2]$ variable needed by Peterson's algorithm. At the exit from its critical section, a process will reset, exactly in the opposite direction, all its positions in the $flag[]$ array that were previously occupied during the upward movement, including the leaf node. After that, the process enters its NCS and the story repeats.

The array $flag[]$ is initialized to all false values. The array $turn[]$ initialization is immaterial.

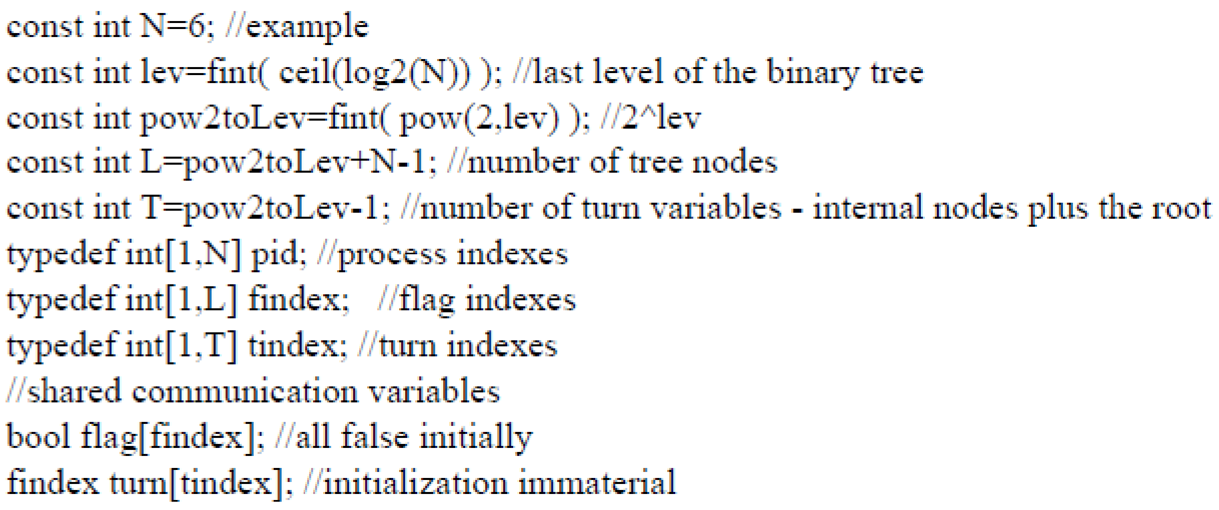
9.2. TT_P2: a TT-based mutual exclusion algorithm that exploits Peterson's solution for 2 processes

A TT-based mutual exclusion algorithm (TT_P2) for $N \geq 2$ processes, depending on Peterson's algorithm for 2 processes as the arbitration strategy, was developed.

Basically, a process i enters its leaf node j (identified by the helper function $leaf(i)$) and then competes at its next arbitration node and moves, when it is the winner of the arbitration, to its ancestor node, ready to execute its next loop from the node $j = j/2$. When j becomes 1 (the root node), the process enters its CS. After that, the process clears, in the reverse order, all the previously occupied positions in the array $flag[]$. Finally, it re-enters its NCS. The clearing actions can awake the processes that can resume their movement, after arbitration, along the TT.

Algorithm 7. The TT_P2 mutual exclusion algorithm.

```
shared communication variables: bool flag[], int turn[]
Process(i):
local variables: int j, sibling
repeat
    j=leaf(i); flag[j]=true;
    while( j>1 ){
      sibling=s(j);
      if( j!=sibling ){
            turn[j/2]=sibling;
            await-until( !flag[sibling] or turn[j/2]!=sibling );
      }
      local_critical_section;
      j=j/2; flag[j]=true;
    }
    CS;
    for( j=1; j<=T; ++j )
      if( path(j) ) flag[j]=false;
forever
```



```
const int N=6; //example
const int lev=fint( ceil(log2(N)) ); //last level of the binary tree
const int pow2toLev=fint( pow(2,lev) ); //2^lev
const int L=pow2toLev+N-1; //number of tree nodes
const int T=pow2toLev-1; //number of turn variables - internal nodes plus the root
typedef int[1,N] pid; //process indexes
typedef int[1,L] findex;   //flag indexes
typedef int[1,T] tindex; //turn indexes
//shared communication variables
bool flag[findex]; //all false initially
findex turn[tindex]; //initialization immaterial
```

Figure 14. Some Uppaal global declarations for the TT_P2 algorithm.

An informal version of the TT_P2 algorithm is reported in Algorithm 7. The function $s()$ returns the sibling of the process in position j, or j itself if the sibling does not exist. The function $path()$ returns true if the current value of j belongs to the path of the process that exited its CS.

A Uppaal model corresponding to the TT_P2 algorithm in Algorithm 7 is portrayed in Figure 15. Its basic global declarations are highlighted in Figure 14.

To reduce the number of data variables, the model in Figure 15 determines the sibling of a process through the $s()$ function, local to the Process's automaton. This way, a local variable to hold the identity of the sibling process is avoided. As shown in Figure 15 (and also in Algorithm 7), the model handles the cases when the sibling process does not exist (a leaf node that is the unique left-linked process of its ancestor node). In these cases, $s()$ returns j itself. Obviously, when the sibling process does not exist, the arbitration immediately ends and the j process progresses to its ancestor node.

The situation of a non-existing sibling is distinguished from the case where the sibling exists but now is not interested in the CS because $flag[s()]$ is false. The locations P2BW and P2CS represent, respectively, the busy-waiting during the arbitration of two processes and the entering to the local critical section of the arbitration. P2CS is realized as an urgent location because the duration of 1 time unit will be spent in the CS of the whole TT algorithm. The $loop$ location was renamed $loop\_fence$ to remember that, under the weak memory, the write operation $turn[j/2] = s()$ has to be fenced for it to be executed in isolation. Such writing, in fact, could be executed simultaneously by the process j and its sibling $s()$, which arrive at the same time at the arbitration point. During the clearing phase, the j variable starts from 1 (the root node) and then increases until T, which is the upper bound of the $turn[]$ array (see also the global declarations in Figure 14).

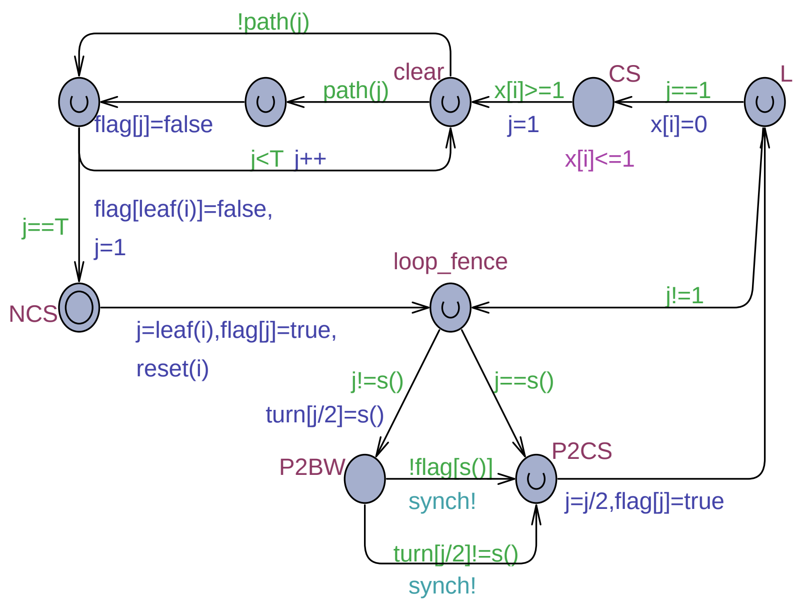


Figure 15. A Uppaal model for the TT_P2 algorithm in Algorithm 7.

The model in Figure 15 was exhaustively model-checked, starting from $N = 2$. All the basic mutual exclusion properties (see the queries in Table 2) were found to be satisfied. In addition, the liveness property that, from the $loop\_fence$ location, any process eventually reaches the CS location, was found satisfied too:

Process(tp).loop_fence --> Process(tp).CS.

In particular, the overtaking bound ov vs. N was found to be as reported in Table 5. For $N = 8$ the state graph explodes. With $N = 7$, the sup query of Table 2 terminates after 354 s with a memory usage of about 4 GB.

As one can see from Table 5, the dependency $ov = 2^{\lceil \log_2 N \rceil} - 1$ is issued, where the right-hand size is the number of intermediate nodes in the TT. When the last level of TT is fully occupied, such a dependency becomes $ov = N - 1$.

By comparing this result with that of the generalized Peterson's algorithm for $N > 2$ processes, it follows that TT_P2, in the general case of an arbitrary dwell-time in NCS, is both more efficient

($ov = N - 1$ vs. $ov = \frac{1}{2}N(N-1)$) and scalable, because it was possible to determine the overtaking bound until $N = 7$.

Table 5. The overtaking bound ov vs. N for the TT_P2 algorithm.

N	ov
2	1
3	3
4	3
5	7
6	7
7	7

A subsequent series of experiments was executed on the model of Figure 15 with NCS made urgent (zero dwell-time in NCS). The new experiments confirmed that all the mutual exclusion properties remain satisfied, and the overtaking bound follows the same behavior as in Table 5. Scalability was found to be a little improved. In the case $N = 8$, an overtaking bound of 7 was proposed, as expected, with the sup query that finishes after 112 s with a memory usage of 1.2 GB. With $N = 9$, the state graph explodes.

To further study the scalability of the TT_P2's algorithm, a Java program equivalent to the model in Figure 15 with NCS urgent was realized using the Theatre actor system [32,33]. A process is implemented as an actor, and each action is modeled by a message that the actor receives and processes. To avoid Zeno-cycles in simulation, instead of actions with zero duration, a minimal duration, e.g., of 0.01 time units, was assumed, whereas the CS always consumes exactly 1 time unit. Busy-waiting loops are programmed by re-sending to itself the received message, tagged with the minimal duration. The number of critical sections executed by other processes when the target process is competing was counted. In addition, all the N processes/actors were mapped onto just one theatre, and the simulation control structure was used. The actor program was executed with a time limit of 10,000 time units (that coincides with the number of executed CS), using different values for N. For example, using $N = 128$, the measured ov was confirmed to be 127 as expected, after a wall-clock-time (WCT) of about 15 sec.

9.3. The TT_P2's algorithm under weak memory

Figure 16 shows the Uppaal model of Figure 15 adapted to work with flickering and weak memory (TT_P2_wm). According to all the queries of Table 2, the new model was found to be correct and RW-safe, exactly as the basic model in Figure 15, although with a more limited scalability. Now, with $N = 5$, the model, either with NCS location normal or urgent, is affected by state explosion. Until $N = 4$, however, the linear character of the overtaking bound was confirmed.

For higher values of N, the Java actor program mentioned in the previous section, adapted for working with weak memory, was exploited. Due to the stochastic behavior introduced by the flickering operations, and the fact that simulation can only observe particular execution paths, a time limit of 10^6 time units was chosen. Table 6 collects the observed $\boldsymbol{s}$imulated $\boldsymbol{ov}$ertaking (SOV) vs. N, for some power of two values of N, along with the $\boldsymbol{e}$xpected $\boldsymbol{ov}$ (EOV) and the wall-clock-time (WCT), in seconds, required by the simulation program to terminate.

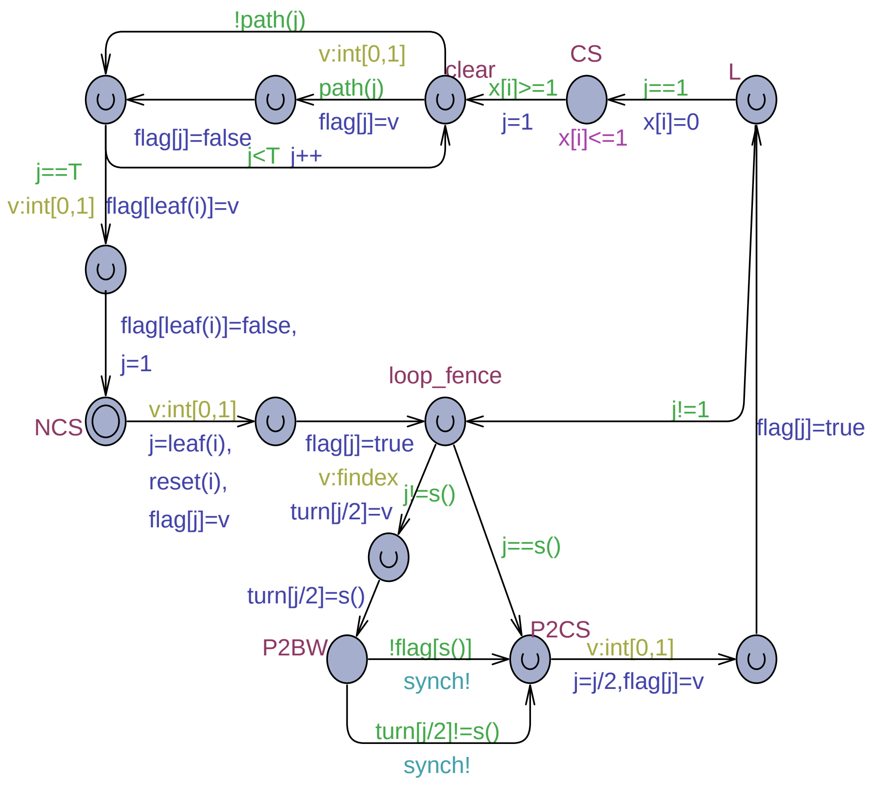


Figure 16. The TT_P2_wm Uppaal model working with memory flickering.

Table 6. Results of the simulated overtaking bound (SOV) using the TT_P2_wm actor program.

N	SOV	EOV	WCT (sec)
8	7	7	46
16	14	15	122
32	30	31	302
64	61	63	702
128	124	127	1508

Although necessarily approximated, with an approximation that gets worse as N augments, the simulation results in Table 6 are coherent with the linear expected trend of ov vs. N.

9.4. TT_PF2: a TT-based algorithm with the Peterson and Fisher solution for 2 processes for the arbitration

Figure 17 shows the TT-based Uppaal model embedding Peterson and Fisher's solution for 2 processes (TT_PF2). For simplicity, the informal textual version of the algorithm is omitted, although it can easily be extracted from Figure 17. The solution relies only on the bool $flag[]$ shared data. The $turn[]$ array is not needed. In addition, no fencing of write operations is required.

The model in Figure 17 satisfies all the mutual exclusion properties (queries in Table 2), with either the NCS location normal or urgent. The overtaking bound was confirmed to be the expected linear in N: $ov = 2^{\lceil \log_2 N \rceil} - 1$. With the NCS normal (as in Figure 17), in the case $N = 5$, the sup query terminates with $ov = 7$ after 817 s with a memory usage of 12.5 GB. As in previous models, the case with NCS urgent proved to be more scalable. For $N = 5$, the sup query terminates with $ov = 7$ after 42 s and with a memory usage of 686 MB.

It is worth noting that the model in Figure 17 follows the hypothesis of the atomic memory model and complies with the simple flickering assumption stated in [25].

Figure 18 reports the TT_FP2 model adapted to work with weak memory and the general flickering mechanism adopted in this paper. As was the case for the basic Peterson and Fisher's solution for 2 processes (see Section 8.5), the model in Figure 18 was also found to be no longer correct from the fundamental mutual exclusion property (only one process can enter its critical section at a time) thus proving it is not RW-safe.

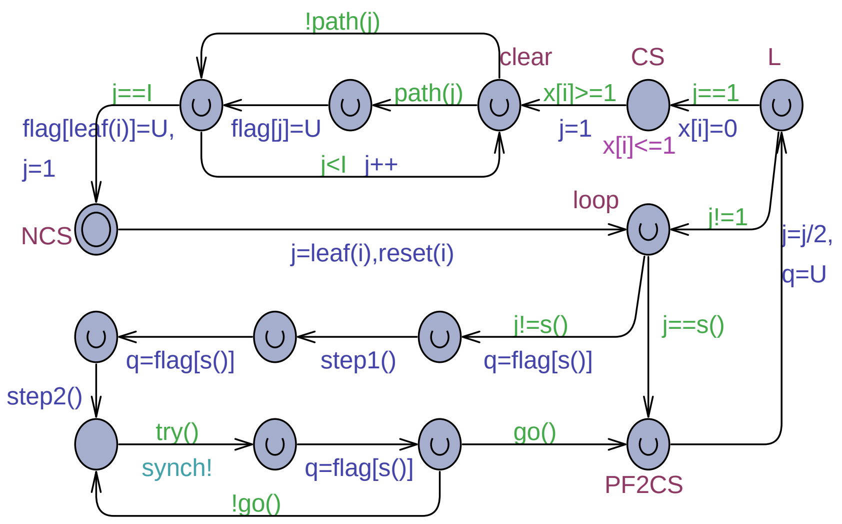


Figure 17. Uppaal model for the TT_PF2 algorithm.

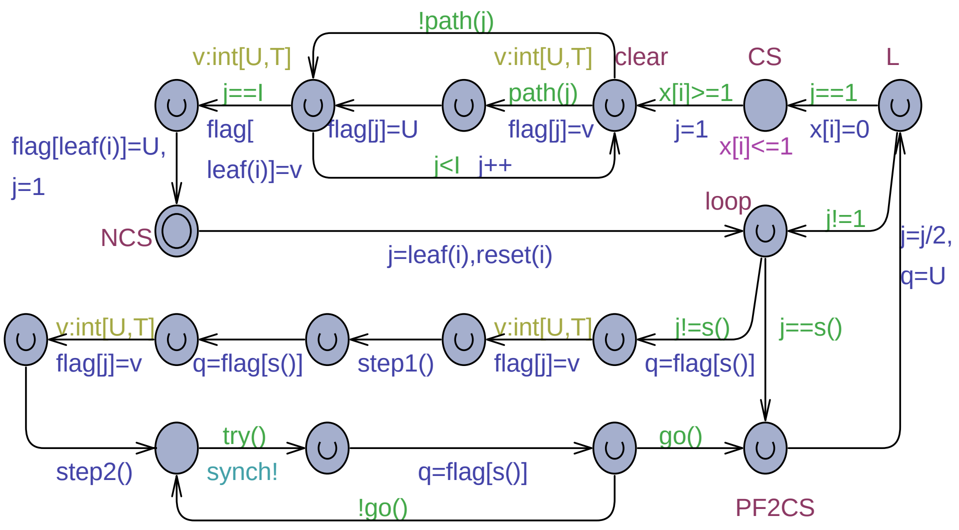


Figure 18. Uppaal model for the TT_PF2 algorithm adapted for flickering.

10. Conclusions

This paper presents a method for formal modeling and analysis by exhaustive model checking of mutual exclusion algorithms. The method is based on the popular Uppaal toolbox [8,21]. As a specific contribution, Peterson's algorithms [5] for $N = 2$ and $N > 2$ processes and Peterson and Fisher's algorithm [25] for $N = 2$ processes were thoroughly investigated under both the classic memory model with atomic read/write operations and the nowadays widespread weak memory model where, e.g., multiple read operations can occur simultaneously with a write operation.

Different results reported in the literature about the overtaking bound of Peterson's solution for $N > 2$ processes were reproduced and related to the hypothesis of process liveness. From this point of view, the possibility offered by the proposed approach of studying a mutual exclusion solution under the influence of the timing dimension was particularly revealing. New properties were disclosed too. For instance, the tournament tree-based [13,26,27] version of mutual exclusion, which relies on Peterson's solution for 2 processes as the arbitration strategy, was found both efficient (linear overtaking bound) and independent from the rate of arrivals of processes for competition.

The prosecution of the research work will be directed at the following points. First, apply the approach to other mutual exclusion algorithms. Second, to improve the Java actor program based on Theatre [32,33] for estimating, through simulations, the properties of a mutual exclusion algorithm for high values of the number N of processes. Third, compare the developed approach to other formal tools, like the assertional methods, with a theorem prover as a proof-assistant [29].

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. E. W. 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
2. M. Raynal, D. Beeson, *Algorithms for mutual exclusion*, Cambridge: MIT Press, 1986.
3. A. Silbershatz, P. N. Galvin, G. Gagne, *Operating system concepts*, 10 Eds., New Jersey: John Wiley & Sons, Inc., 2018.
4. E. W. Dijkstra, Solution of a problem in concurrent programming control, *Commun. ACM*, **8** (1965), 569. https://doi.org/10.1145/365559.365617
5. G. L. Peterson, Myths about the mutual exclusion problem, *Inform. Process. Lett.*, **12** (1981), 115–116. https://doi.org/10.1016/0020-0190(81)90106-X
6. E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, H. Veith, *Model checking*, Cambridge: MIT Press, 2018.
7. C. Baier, J. P. Katoen, *Principles of model checking*, Cambridge: MIT Press, 2008.
8. G. Behrmann, A. David, K. G. Larsen, A tutorial on UPPAAL, In: *M. Bernardo, F. Corradini (Eds.), 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

9. F. Cicirelli, A. Furfaro, L. Nigro, Model checking time-dependent system specifications using time stream Petri nets and Uppaal, *Appl. Math. Comput.*, **218** (2012), 8160–8186. https://doi.org/10.1016/j.amc.2012.02.018
10. 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.
11. M. Hofri, Proof of a mutual exclusion algorithm – a 'Class'ic example, *ACM SIGOPS Operating Systems Review*, **24** (1990), 18–22. https://doi.org/10.1145/90994.9100
12. T. Kowalttowski, A. Palma, Another solution of the mutual exclusion problem, *Inform. Process. Lett.*, **19** (1984), 145–146. https://doi.org/10.1016/0020-0190(84)90093-0
13. 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
14. L. Nigro, F. Cicirelli, Correctness verification of mutual exclusion algorithms by model checking, *Modelling*, **5** (2024), 694–719. https://doi.org/10.3390/modelling5030037
15. L. Nigro, Formal modelling and verification of Lycklama and Hadzilacos's mutual exclusion algorithm, *Mathematics*, **12** (2024), 2443. https://doi.org/10.3390/math12162443
16. E. A. Lycklama, V. Hadzilacos, A first-come-first-served mutual-exclusion algorithm with small communication variables, *ACM Trans. Progr. Lang. Sys.*, **13** (1991), 558–576. https://doi.org/10.1145/115372.115370
17. A. A. Aravind, Simple, space-efficient, and fairness improved FCFS mutual exclusion algorithms, *J. Parallel Distr. Com.*, **73** (2013), 1029–1038. https://doi.org/10.1016/j.jpdc.2013.03.009
18. J. Burns, Complexity of communication among asynchronous parallel processes, Ph. D. Thesis, Georgia Institute of Technology, 1981.
19. L. Lamport, The mutual exclusion problem: part II: statement and solutions, *JACM*, **33** (1986), 313–348. https://doi.org/10.1145/5383.5385
20. R. Alur, D. L. Dill, A theory of timed automata, *Theor. Comput. Sci.*, **126** (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8
21. *Uppaal on-line*, Uppsala University and Aalborg University, June 2024. Available from: https://uppaal.org.
22. E. M. Clarke, W. Klieber, M. Nováček, P. Zuliani, Model checking and the state explosion problem, In: *Tools for practical software verification*, Berlin: Springer, 2011, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1
23. P. A. Buhr, D. Dice, W. H. Hesselink, Dekker's mutual exclusion algorithm made RW-safe, *Concurr. Comp.-Pract. E.*, **28** (2016), 144–165. https://doi.org/10.1002/cpe.3659
24. L. E. Frenzel, Dual-port SRAM accelerates smart-phone development, *Electron. Des.*, **4 (**2004), 35.
25. G. L. Peterson, M. J. Fischer, Economical solutions for the critical section problem in a distributed system, *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, 91–97. https://doi.org/10.1145/800105.803398
26. J. L. W. Kessels, Arbitration without common modifiable variables, *Acta Inform.*, **17** (1982), 135–141. https://doi.org/10.1007/BF00288966
27. W. H. 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

28. X. Ji, L. Song, Mutual exclusion verification of Peterson's solution in Isabelle/HOL, *Proceedings of Third International Conference on Trustworthy Systems and their Applications*, 2016, 81–86. https://doi.org/10.1109/TSA.2016.22
29. W. Hesselink, Correctness and concurrent complexity of the Black-White Bakery algorithm, *Form. Asp. Comp.*, **28** (2016), 325–341. https://doi.org/10.1007/s00165-016-0364-4
30. Y. Hafidi, J. J. Keiren, J. F. Groote, Fair mutual exclusion for N processes, In: *Tools and methods of program analysis*, Cham: Springer, 2021, 149–160. https://doi.org/10.1007/978-3-031-50423-5_14
31. T. Murata, Petri nets: properties, analysis and applications, *Proc. IEEE*, **77** (1989), 541–580. https://doi.org/10.1109/5.24143
32. 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
33. L. Nigro, F. Cicirelli, P. Fränti, Parallel random swap: an efficient and reliable clustering algorithm in Java, *Simul. Model. Pract. Th.*, **124** (2023), 102712. https://doi.org/10.1016/j.simpat.2022.102712