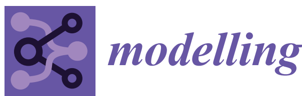

Article

Correctness Verification of Mutual Exclusion Algorithms by Model Checking

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

[1] DIMES—Engineering Department of Informatics Modelling Electronics and Systems Science, University of Calabria, 87036 Rende, Italy
[2] CNR—National Research Council of Italy, Institute for High Performance Computing and Networking (ICAR), 87036 Rende, Italy; f.cicirelli@icar.cnr.it
* Correspondence: libero.nigro@unical.it

Abstract: Mutual exclusion algorithms are at the heart of concurrent/parallel and distributed systems. It is well known that such algorithms are very difficult to analyze, and in the literature, different conjectures about starvation freedom and the number of by-passes (also called the overtaking factor) exist. The overtaking factor affects the (hopefully) bounded waiting time that a process competing for entering the critical section has to suffer before accessing the shared resource. This paper proposes a novel modeling approach based on Timed Automata and the Uppaal toolset, which proves effective for studying all the properties of a mutual exclusion algorithm for $N \geq 2$ processes, by exhaustive model checking. Although the approach, as already confirmed by similar experiments reported in the literature, is not scalable due to state explosion problems and can be practically applied until $N \leq 5$, it is of great value for revealing the true properties of analyzed algorithms. For dimensions $N > 5$, the Statistical Model Checker of Uppaal can be used, which, although based on simulations, can confirm properties by estimations and probabilities. This paper describes the proposed modeling and verification method and applies it to several mutual exclusion algorithms, thus retrieving known properties but also showing new results about properties often studied by informal reasoning.

Keywords: mutual exclusion algorithms; correctness analysis; automated reasoning; model checking; statistical model checking; Timed Automata; Uppaal

Citation: Nigro, L.; Cicirelli, F. Correctness Verification of Mutual Exclusion Algorithms by Model Checking. *Modelling* **2024**, *5*, 694–719. https://doi.org/10.3390/modelling5030037

Academic Editor: Mauro Iacono

Received: 2 April 2024
Revised: 14 June 2024
Accepted: 24 June 2024
Published: 28 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Mutual exclusion (ME) [1–5] is a fundamental problem in the concurrent/parallel, distributed, embedded real-time programming domains [6–10]. In its basic form, it can be stated as follows. We have a software system with $N \geq 2$ processes which compete for the use of a shared resource R. For predictability and deterministic evolution of the system behavior, only one process should be allowed at a time to use the resource. The code executed by the process when it is using the resource constitutes its atomic *critical section*. Other properties of a ME algorithm include the following: (a) *progress*, that is, a process competing for accessing the resource eventually obtains the permission to use the resource; (b) an absence of *deadlocks*; and (c) no process should experience *starvation*, in other terms, a competing process should not suffer from an unbounded waiting or an unbounded number of by-passes/overtakings from other competing processes.

ME solutions typically can be provided by a high-level or low-level abstraction context. High-level ME mechanisms can directly be provided by a programming language (e.g., Java) and accompanying library and include semaphores, locks or monitor constructs [11]. Such solutions, though, ultimately depend on the mechanisms implemented in the underlying Operating System. Primitive ME solutions are required to give support to high-level mechanisms. Primitive ME solutions can be based on specialized hardware mechanisms, e.g., machine instructions like *test-and-set*, or they can be designed as *pure* software algorithms which depend on (hopefully) few shared variables and a purposely designed, often

ingenious, protocol which establishes the *Entry* and *Exit* code sections which processes have to obey, respectively, for competing and entering the resource and for releasing the resource at the end of the critical section.

This paper focuses on pure software ME primitive algorithms, whose properties can be very difficult to assess. In the literature, such algorithms are normally studied by informal reasoning [12] typical of ordinary mathematics. More structured tools are represented by assertional methods based on global invariants together with a theorem prover like PVS as a proof assistant [13–18]. The correctness argument of an assertional method proceeds by first transforming an ME algorithm into a transition system. Assertions are about the state and the next state relation. It is worth noting, though, that the use of theorem provers not only can require a not easy formal representation of the algorithms and their data variables, but they are also unable to reason on the time dimension of an ME solution.

This paper proposes an original modeling and verification approach based on the Timed Automata [19] and the Uppaal toolbox [20] which enable an automated assessment of the properties of an ME algorithm through exhaustive model checking [21]. Although the approach necessarily is not scalable due to the *state explosion problem* which inevitably intervenes when considering a number of processes $N > 5$, it is of great practical value for detecting the true properties of ME solutions, which can go beyond some indications formulated in the literature, particularly about the number of by-passes a process can be affected by when competing together with its peers.

This paper significantly extends the preliminary work and results of the authors as reported in two conference papers [22,23]. Differences from the previous authors' work are as follows.

- The modeling and verification method is completely re-designed and its semantics and transformation on to the Timed Automata of Uppaal are clarified.
- The novel approach is applied to several ME algorithms proposed in the literature, thus retrieving known results and properties but also, sometimes, discovering new features about specific ME solutions.
- Both exhaustive model checking (MC) and statistical model checking (SMC) [24,25] can be exploited for property checking. Although SMC is based on simulations, the properties of an ME algorithm for $N > 5$ processes can be extrapolated by estimations and probability, thus furnishing further arguments to the indications emerged from the exhaustive verification work accomplished until $N = 5$ processes.

This paper is structured as follows. Section 2 presents an overview of the basic concepts of Timed Automata and Uppaal modeling. Section 3 describes the proposed modeling and verification method for mutual exclusion algorithms and its semantic transformation on to Uppaal. Section 4 demonstrates via several examples the application and the experimental verification work carried out by using the proposed approach on representative mutual exclusion algorithms. Both the atomic and the weak memory models are considered. The interplay between the exhaustive model checking and the statistical model checking activities is clarified. Finally, Section 5 concludes this paper and indicates some on-going and future work.

2. An Overview of Uppaal Timed Automata Modeling

Uppaal [20] is a popular modeling and verification toolbox for concurrent/distributed real-time systems. The power and flexibility of Uppaal derive from the adoption of a high-level version of Timed Automata (TA) [19] which are formal concurrent entities with an easy-to-understand graphical formulation.

2.1. Specification Aspects

A model consists of a network of interacting TA. Interactions are based on unicast and broadcast channels and on global data of primitive types like integers and bools, plus structs and arrays. Unicast channels ensure two-way synchronizations in couples of TA (also called template processes or parameterizable processes). The sender process of a

unicast channel transmits (!) a signal toward a single receiver. The sender, though, becomes blocked until the receiver is ready to receive (?) the signal and vice versa (*rendezvous*). The synchronization does not carry any data or parameters which can easily be provided through the mediation of some global data variables. After the synchronization, both the sender and receiver resume their concurrent execution. Broadcast channels, instead, enable one-way synchronizations, where a single sender can synchronize with zero, one or multiple receivers. In other terms, the sender of a broadcast channel never blocks. Uppaal models are time-sensitive. The passage of time is controlled by *clocks*. A clock can be reset. A clock measures the time units elapsed from its last reset. All the clocks of a model grow according to the same rate. Time is assumed to be dense. A process automaton consists of *locations* linked by directed *edges* annotated by *guarded commands*. A location denotes a local state where the automaton can remain for a certain amount of time (including zero or possibly infinite time units). A normal location (see the NCS location in Figure 1) denotes a state where the automaton can stay for an arbitrary time. Uppaal provides the special concept of an *invariant*, that is, a logical condition, e.g., based on a clock constraint, attached to a normal location, to restrict the permanence of the automaton the location. In this case, the location has to be abandoned before its invariant is up to become false. An urgent location (with an internal U) has to be exited immediately, without time passage. A committed location (see the initial location with the internal C in Figure 2) is similar to an urgent location. However, Uppaal gives priority to committed locations which will be exited before the urgent ones. Among the same group of committed or urgent locations, the exit order is non-deterministic. A guarded command is composed of three optional attributes: a *guard* (a logical condition based on clock and/or data constraints, drawn in green, see Figure 3) which is true by default when it is missing; a *channel synchronization* operation (? or !, drawn in azure, see Figure 3); and an *update* component (drawn in blue, see Figure 3), that is, a comma separated sequence of variable assignments and clock resets. A void command denotes a spontaneous edge. A spontaneous edge originating from a normal location can be taken at any time, also after an infinite amount of time, that is, the edge can possibly not be taken at all. A guarded command represents the *unit of concurrency* of the Uppaal modeling language. When a channel synchronization involves two or more locations in different automata, a joint instantaneous exiting of the corresponding locations will occur. The semantics of Uppaal prescribes, in a joint synchronization in multiple automata, that the update operations in the sender precede the updates of the receivers (the order of the receiver updates is defined in the system declaration section).

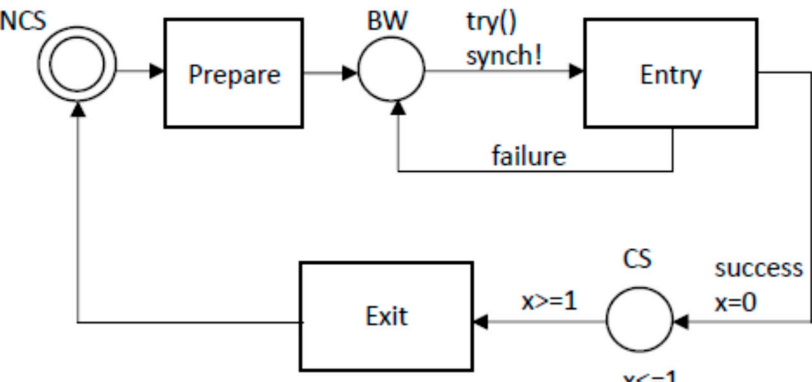

Figure 1. Proposed abstract Uppaal model for a mutual exclusion process.

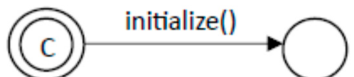

Figure 2. The Bootstrap automaton.

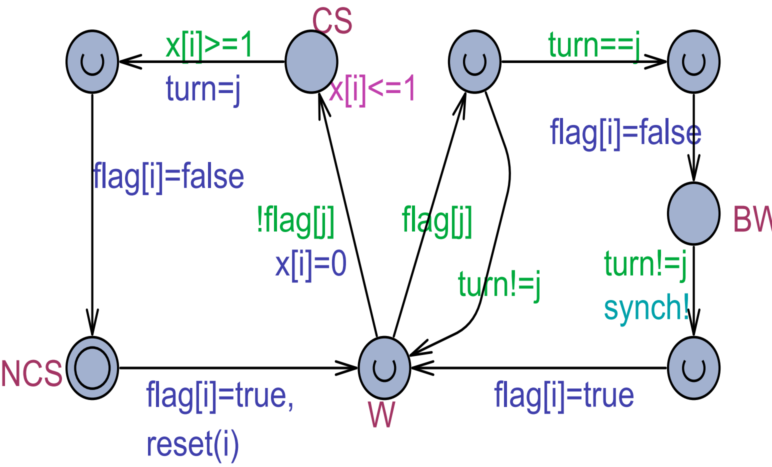

Figure 3. Uppaal model of Dekker's Process(i).

Unicast and broadcast channels can be declared to be urgent. In this case, an enabled command involving an urgent synchronization, will be executed before any time passage. However, the order of exiting from multiple simultaneous urgent locations and normal locations with an exiting edge annotated with an enabled guard and an urgent synchronization is non-deterministic.

A distinguishing feature of the Uppaal modeling language which contributes to the creation of compact models is the possibility of exploiting C-like functions (see Algorithm 5 for an example) for defining the guard and/or the update components of commands.

2.2. Semantics and Verification Aspects

The formal semantics of a Uppaal model can be defined by a Timed Transition System (TTS) defined as $(S, s0, \rightarrow)$ where S is a set of execution states of the model, $s0$ is the initial state and $\rightarrow$ denotes the *state relation* which moves the current state to the next one. Two basic relations exist: the *delay* transition and the *action* transition. An action transition represents an instantaneous state change, e.g., tied to a joint synchronization in multiple automata. All the action transitions enabled in the current system state will be executed before time advancement. When no more actions exist to be executed at the current time, the system time can be increased by quantity δ whose amount is constrained by the fact of not falsifying any invariant in the whole model. The amount δ is added to all the model clock valuations. A particular model execution is represented by a sequence of delay/action transitions.

The Uppaal model checker (here referred to as U-MC), accepts a query formula expressed according to a subset of Timed Computational Tree Logic (TCTL) [20], builds the TTS of the model, referred to as the model *state graph*, and navigates the graph to prove/disprove the query. A query captures a property to be checked. Properties can be of the following types: *safety* (checking that a bad state is never reached), *liveness* (checking that a good state is eventually reached) and *bounded liveness* (liveness property ensured within a given time-bound) nature.

The basic TCTL supported queries are the following:

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

where φ and ψ are state properties (formulas), e.g., clock constraints or boolean expressions built in terms of predicates on locations and so forth.

A counterexample (said a diagnostic trace) can be built by U-MC to reproduce a sequence of transitions (events) which the modeler can analyze, in the symbolic simulator,

to detect why a given formula was satisfied or not satisfied by the state graph. Nodes (execution states) of the state graph are made up of two parts: a *data* part and a *time zone*. The time zone represents, by a system of clock inequalities, all the possible (also infinite) time instants for reaching the given state. A node of the state graph is actually an equivalence class: it represents all the states having the same data and whose occurrence times are solutions of the time zone inequality system. Each arc exiting from a state node and linking to the next state node is labelled with the delay or action transition which causes it.

The complexity of the state graph is exponential in the number of clocks. However, it is not the absolute number of clocks used in the model which matters, but rather the maximum number of simultaneous active clocks, that is, the maximum parallelism degree of the model. An active clock is one which is currently used in an invariant or in a constraint in a guard. The number of active clocks affects the complexity of the state graph. In addition, the memory demand critically increases as more data variables are introduced by the modeler. Another measure of the state graph complexity is represented by the degree of action interleaving (partial order), mirrored by the number of exiting arcs from a state node. The Uppaal model checker (U-MC) navigates the state graph by exploring all the possible execution paths and then the action interleavings, which originate from the initial state node.

In the last few years, Uppaal was also equipped with a Statistical Model Checker (U-SMC) [24,25] which does not build the state graph but uses instead simulations for estimating properties. The memory consumption of U-SMC is linear with the model requirements. U-SMC gives the modeler the possibility of interpreting the basic TA as Stochastic Timed Automata. To a normal location, the rate of a negative exponential probability distribution can be attached, which is sampled to furnish the dwell time for remaining in the location. In the case a normal location is provided with an invariant on a clock x of the type $x \leq b$, where b is the maximal time the automaton can stay in the location, and an exiting edge with a guard like $x \geq a$, with $a \leq b$, under U-MC, the automaton can exit the location at any time non-deterministically chosen in the dense interval $[a, b]$. Under U-SMC, a delay is sampled by a uniform probability distribution in the same interval, which constitutes the dwell time before abandoning the location. U-SMC supports queries written in Metric Interval Temporal Logic (MITL) [25] (see Table 2 for examples). As a simple example, U-SMC can estimate the probability of an event occurrence by executing a certain number of simulations (whose amount can be automatically inferred), each one lasting after an assigned time limit, and counting how many times the event occurs in the various runs, divided by the total number of runs. As a further benefit, U-SMC can accumulate data from a simulation so as to display them graphically, thus favoring a better understanding of a dynamic behavior.

Uppaal Timed Automata are used in this work to model and analyze mutual exclusion algorithms by model checking. A key feature offered by Uppaal is the possibility of investigating the effects of time on the behavior of such algorithms.

3. A Modeling Method for Mutual Exclusion Algorithms

Mutual exclusion algorithms can be studied according to two memory models: the common strong memory model (SMM) and the weak memory model (WMM) [9,26]. According to SMM, the write and read operations issued by processes on a memory cell are atomic. In the WMM model, though, a memory cell under writing can be simultaneously read by other processes and a non-deterministic value can be achieved (*flickering*). In this paper, both models will be considered.

In the following, N processes identified by unique numbers from 1 to N, compete for the use of a shared resource. The abstract code structure of the generic Process(i) is assumed to be a never-ending loop as in Algorithm 1.

Algorithm 1. General Code Structure of a Process Involved in Mutual Exclusion

```
global shared communication variables
Process(i):
local variables of the process
loop
    NCS //Non Critical Section
    Entry
    CS //Critical Section
    Exit
end-loop
```

In the NCS section, the process does something unrelated from the shared resource. In particular, the duration of NCS is arbitrary and can include 0 time units (the NCS is immediately abandoned) and infinite time units (in the NCS, the process can block or terminate). Process i follows a specific protocol for accessing the resource. The Entry part is what is called the *competing* part. The process signals its interest in accessing the resource. However, it has to wait until the process obtains the grant to actually access the resource. The Exit part consists of the final operations which the process executes to let partners be informed it has finished with this access to the resource.

Both the Entry and the Exit code sections depend on and make use of the global and shared communication variables. On the other hand, each process owns some private local variables, inaccessible to other processes. The original design of a mutual exclusion algorithm concerns devising a minimal number of shared communication variables whose use is capable of ensuring the mutual exclusion and the related properties.

A correct mutual exclusion algorithm should guarantee the following properties are fulfilled (e.g., [1–4], see Section 2.2 for the meaning of safety and (bounded) liveness property).

- (*Safety*) Absence of deadlocks in the processes attempting to use the shared resource.
- (*Safety*) At any time, only one process should be allowed to enter its critical section.
- (*Bounded liveness*) A competing process eventually enters its critical section. In other terms, the waiting time for a competing process should be finite and hopefully small. Equivalently, the number of *by-passes* or the *overtaking factor* which a waiting process experiments from other competing processes should be bounded. This property also expresses, for a competing process, the *absence of starvation*.
- (*Liveness*) A process in its non-critical section should never impede a competing process to enter its critical section.

3.1. Uppaal Modeling Method

Mutual exclusion (ME) algorithms normally do not make any assumption about the time duration of the operations in the Entry, the Exit and the CS sections (see Algorithm 1). In the proposed Uppaal method for a generic process (see Figure 1), all the elementary operations are modeled by *urgent locations*, thus having a 0 duration. As a consequence, the interleaved, non-deterministic execution order of process actions is naturally expressed. In addition, the use of urgent actions enables *weak fairness* semantics (e.g., [14,27]), which ensures progress or liveness of process executions: any process continually enabled to perform a step will eventually perform it. However, a high degree of interleaving or partial order in the nodes of the state graph, particularly when the number N of processes increases, complicates the generation and the navigation of the state graph during model checking. These considerations suggested a refinement of the ME modeling method as follows. First, the execution of the Entry actions, although continuously executed (*busy-waiting*) in the physical process, perhaps running on a separate processor/core, in the Uppaal model can be conveniently postponed (lazy modeling) until there is an "evidence" that changes in the shared communication variables occurred which require the process to actually check if it really can prosecute toward the critical section or not. All of this in no case introduces

changes in the logic of the ME algorithm, but it can reduce the burden of an excessive interleaving degree in the model.

In Figure 1, the ME model-dependent try() boolean function is introduced, which returns true if something is changed in the shared communication variables which motivates the process to execute the Entry actions. If the guess expressed by try(), for non-determinism and concurrent execution with partner processes, would result in a failure, that is, the process still has to continue waiting in the competing state, the BW location will be re-entered. If instead the Entry actions end with success, the process continues by entering its critical section. It is worth noting that BW is immediately exited at any time the optimistic try() function returns true. Toward this, a (fictitious) signal sent over the urgent and broadcast channel synch is used. Note that no receiver exists for this asynchronous message.

The CS section (see Algorithm 1) is purposely modeled as a normal location with a dwell time of exactly 1 time unit (see the invariant $x \leq 1$ on CS and the guard $x \geq 1$ on the edge exiting the location CS). The use of a unitary duration for any critical section makes the model time-sensitive, facilitates the prediction of the overtaking factor (see later in this paper) and does not affect the mutual exclusion algorithm evolution. On the other hand, the CS behavior can naturally break possible Zeno-cycles [28], that is, those infinite sequences of events executed in 0 time, which could arise in a model with a lot of urgent actions.

The non-critical section NCS is instead modeled as a normal location with a spontaneous exiting edge. As a consequence, NCS can be abandoned in 0 time units, or the process automaton can remain in it also an infinite number of time units. The last situation models the process which can be stopped into NCS.

Special care is required in the modeling of the actions of both the Entry and the Exit sections. Under both the strong memory model (SMM) and the weak memory model (WMM), it is fundamental to realize that individual shared communication variables must be set (written) or obtained (read) in separated edges of the Uppaal process model. All of this mirrors the fact that each single shared communication variable has to be accessed from a different memory location. As a consequence, a condition based on multiple shared variables has to be split, and its result calculated by separately evaluating its basic components. This way, the evaluation of the condition can be correctly affected by the non-determinism and interleaving of the various processes. The situation is different for local variables of a process. Such variables are supposed to be held in separate registers so that multiple local variables can be operated in a single edge command without errors.

The above mentioned constraints on the use of shared communication variables do not apply in the formulation of the try() function body, where the goal is to “rapidly” predict, e.g., that the process can possibly enter its critical section. It has to be stressed, though, that a true result of try() expresses only an optimistic prediction which can be falsified in the actual non-deterministic execution of Entry actions.

As one can see from Figure 1, the initial location of a process model is represented by NCS. The overall model with N process instances can be initialized by a Bootstrap automaton (see Figure 2) whose initial location is marked as committed so as to be sure, through the initialize() function, that all the global shared data are initialized before any process starts executing. After the execution of the initialize() function, the Bootstrap moves to a final normal location without exits, thus disconnecting from the model.

3.2. Predicting the Overtaking Factor

In the model of Figure 1, time can advance not only because the CS location has one time unit duration but also because of the unpredictable staying of the process into the NCS. The interplay of CS/NCS timing is a key factor for predicting the number of overtakings. Since all the processes are identical, one of them can be elected for measuring the overtakings. Let it be denoted by tp (target process, by default the process 1). Some common global declarations of a mutual exclusion Uppaal model are the following:

```
const int N=...; //the number of processes
typedef int [1,N] pid; //the type of process unique identifiers
const pid tp=1; //target process example
clock x[pid]; //one clock per process
```

The clock x[tp] is reset as soon as the target process starts competing. The following reset(i) function can be used for this purpose:

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

After its reset, x[tp] grows as other competing processes enter their critical sections. Due to the unitary duration of the location CS (see Figure 1), the maximal value reached by x[tp] measured before entering its CS, can furnish the number of overtakings. Such a number coincides with the number of CSs executed by competing processes before the tp process is allowed to enter its CS.

Semantic correctness of this approach, despite the timing introduced by NCS, can be stated as in the following. Of course, by design, a mutual exclusion algorithm is expected to ensure only one process can be in its critical section at a time, and that a finite number of overtakings can be suffered by any competing process and also by the tp process. Let Δ_{tp} be the finite amount of time that the (competing) tp process has to wait before entering its CS. Let tc be a competing process which obtains its CS before tp and then enters its NCS which is finally abandoned after a time Δ_{tc}, when a new competition for tc is started. In the case $\Delta_{tc} \geq \Delta_{tp}$ it necessarily follows that the process tp eventually enters its CS, and a new competition phase is then launched. If instead $\Delta_{tc} < \Delta_{tp}$, this means that the time spent by tc into its NCS naturally occurs, by concurrency/parallelism, as part of the time spent for waiting by tp, mirrored by the clock valuation x[tp]. Therefore, the time behavior of processes into NCS does not forbit the correct detection of the maximal number of overtakings by observing the clock x[tp].

An important consequence of the mutual exclusion property and of the choice of the target process tp for observing the number of overtakings is that, at any moment of the model verification, despite the number N of the processes, at most *two* clocks can be active: that of the process tp and one of the competing processes which is allowed to enter its critical section.

3.3. Model Checking and TCTL Queries

A basic set of TCTL [20] queries (see also Section 2.2), which can be used to study a mutual exclusion algorithm expressed in Uppaal according to the general process model of Figure 1, is suggested in Table 1. The use of the built-in functions sum(...) and exists(...) should be self-explanatory.

Table 1. Some TCTL queries for checking a modeled mutual exclusion algorithm.

#	TCTL query	Property
1	A[] !deadlock	Absence of deadlocks.
2	A[] (sum(i:pid)Process(i).CS)<=1	At most one process can be in its critical section (CS).
3	Process(tp).BW −− >Process(tp).CS	A process in BW eventually enters the critical section.
4	E<> Process(1).NCS && exists(j:pid) Process(j).CS	A process in NCS does not forbid another process from entering the CS.
5	sup{ Process(tp).BW } : x[tp]	The suprema value of the overtaking factor.

The first query of Table 1 asks if it is always true (A[]), or invariantly, that in all the states of the state graph, there is no deadlock. The second query checks if, invariantly, the

number of processes that are in the CS location is always less than or equal to one (the fundamental mutual exclusion property). The third query, based on the $leads - to$ operator $- - >$, states that starting from the BW location, it inevitably follows that the process enters its CS. The fourth existential query (E<>) verifies that a process in the NCS does not forbid another process to be in the CS. The fifth query, finally, asks to quantify the maximal number of overtakings suffered by the chosen target process. Different locations, not necessarily the BW, can be chosen for observing the suprema value of x[tp]. For example, a (urgent) location just before entering the CS could be adopted as well. The sup query is equivalent to the following basic query:

A[] Process(tp).BW imply x[tp]<=*ov-bound*

where *ov-bound* is the expected lower bound of the overtaking factor.

3.4. Statistical Model Checking and MITL Queries

The basic process model in Figure 1 can easily be adapted for it to be also exploited under the Uppaal Statistical Model Checker (U-SMC) [25]. Basically, the possibility of time not advancing in the NCS location can be avoided by attaching to NCS the rate of an exponential probability distribution function which can conveniently be established as $mu = 1.0/EOF$, where EOF is the expected overtaking factor.

Some MITL queries useful for checking the model of a mutual exclusion algorithm through simulations are proposed in Table 2. It is to be anticipated, though, that it can be difficult, in simulation, to observe the effective number of overtakings. This is because U-SMC, during a simulation, follows a particular execution path of the actions interleaving, whereas the model checker analyzes *all* the possible paths.

Table 2. Some MITL queries for monitoring the overtaking factor.

#	MITL query	Property
1	Pr[<=10000](<>(sum(i:pid)Process(i).CS)>1)	Probability that more than one process can be in CS.
2	simulate[<=100]{sum(i:pid)Process(i).CS}	Monitoring the number of processes simultaneously in CS in a simulation of 100 time units.
3	Pr[<=1000](<>Process(tp).CS)	Probability that a process can actually enter its CS.
4	Pr[<=10000](<>Process(tp).BW && x[tp]>*expected-value*)	Probability that the overtaking factor (x[tp]) can be found greater than the *expected-value*.
5	Pr[<=10000]([] Process(tp).NCS || x[tp]<=*expected-value*)	Probability that x[tp] is always found less than or equal its *expected-value*.

Query 1 of Table 2 asks U-SMC to estimate, using an inferred number of simulation runs each lasting 10,000 time units, the occurrence probability of the event "*is there any state in which the number of processes in CS is greater than 1?*". U-SMC proposes a confidence interval (by default with a confidence degree 95%) for the event probability. Of course, the confidence interval should indicate an almost impossible event. With the default parameters setting of U-SMC [25], a typical "impossible" event is witnessed by the interval $[0, 0.0981446]$. Similarly, an almost "certain" event is denoted by a confidence interval of $[0.901855, 1]$. The second query monitors the number of processes simultaneously in CS in a simulation, e.g., lasting 100 time units. Following the query execution, U-SMC can show the accumulated data graphically. The third query quantifies the probability of a process, e.g., the tp, to effectively enter its critical section. Such a query is expected to suggest an almost sure event. The fourth and fifth queries ask U-SMC to quantify the confidence interval of the probability that the target process experiences a number of overtakings, respectively, greater than the expected bound (an event which should be impossible) or that such a number is always found to be less than the expected bound (an event which should be certain).

4. Modeling and Analysis of Mutual Exclusion Algorithms

The following reports several modeling examples and their analysis results. The presented models are examples of *scientific models* [29], in the sense that they are aimed at permitting reasoning on and semantic interpretation of the selected mutual exclusion algorithms. They differ from *engineering models* where a formal system model is preliminarily prepared and analyzed toward a physical implementation. All the presented experiments were carried out using the latest version Uppaal 5.0.0, 64 bit, running on a Win11 Pro desktop platform, Dell XPS 8940, Intel i7-10700 (8 physical cores), CPU@2.90 GHz, 32GB RAM.

4.1. Solutions for Two Processes

The mutual exclusion problem for two processes was solved in the sixties of the previous century by T.J. Dekker [6]. This algorithm was improved by G.L. Peterson in [30].

4.1.1. Dekker's Algorithm

Dekker's algorithm is based on the following shared communication variables:

bool flag [2], int [1,2] turn

The flag array is initialized to all false. The initial value of turn can be either 1 or 2. The generic process i (1 or 2) is shown in Algorithm 2, where the Entry and Exit code sections are clearly recognizable. The local constant j denotes the partner process. Each process sets the value of its own flag[], which can be checked by the partner (flag[.] variables are said *exterior* variables in [31]). When a process wants to compete for the use of the resource, it sets its flag[] variable to true. The value of turn ultimately decides which process actually enters its critical section. When a process exits from its CS, it gives the turn to the partner and puts its flag[] to false to state it is, at the moment, not interested in the resource. Despite its apparent simplicity, understanding the Entry section can be difficult. Non-determinism, in fact, is at the heart of the algorithm behavior. The Uppaal model of Process(i) is depicted in Figure 3, where the try() function (see Figure 1) was directly expanded. The Process automaton has only one parameter: const pid i. The use of the pid typedef enables N instances (here $N = 2$) of the Process automaton to be created at the system Bootstrap time, and the various instances are distinguished by Process(1) and Process(2).

The behavior of the algorithm can be preliminarily visually inspected by animating the Uppaal model in the symbolic simulator, which is the typical tool for debugging model errors. Effectively, one "can see" that the two processes never enter their CS simultaneously. In addition, should the partner be in the busy-waiting location BW, when the current process exits its CS, the waiting process can be unblocked by receiving its turn. Things, though, can be more complicated with respect to what intuitively emerges from the animation.

Property verification can be assessed through the model checker. Effectively, the Dekker model is deadlock-free (query 1 of Table 1). In addition, the fundamental mutual exclusion property (query 2 of Table 1) is satisfied. This property was accompanied by the assessment that both Process(1) and Process(2) can effectively enter the CS with the two existential queries which are satisfied:

E<> Process(1).CS
E<> Process(2).CS

Also, the liveness property checked by query 4 of Table 1 is satisfied. However, query 3 of Table 1 as well as the query

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

are not satisfied. In reality, this property is tied to the fact that the model in Figure 3 has a Zeno-cycle (see the loop from W to itself: flag[j] followed by turn!=j, which can be executed an infinite number of times and in 0 time). Query 5 of Table 1 furnishes, whatever the target process is, an overtaking factor of 1.

Algorithm 2. Dekker's Process(i)

```
Process(i): local constant: int j=3-i;
while(true){
    NCS;
    flag[i]=true;
    while( flag[j] ){
        if( turn==j ){
            flag[i]=false;
            wait-until( turn==i );
            flag[i]=true;
        }
    }
    CS;
    turn=j;
    flag[i]=false;
}
```

As a final remark, it should be observed that Zeno-cycles (equivalent to a mechanical perpetual motion) can exist in the model but not in physical reality where operations have, necessarily, a non-zero execution time, provided an adequate time resolution level is adopted.

4.1.2. Peterson's Algorithm for Two Processes

A more elegant and compact solution to the mutual exclusion problem for two processes was proposed by G.L. Peterson in [30]. Using the same shared data as in Dekker's algorithm and the same initializations, the pseudo-code of the algorithm and its Uppaal model are shown, respectively, in Algorithm 3 and Figure 4.

Algorithm 3. Peterson's Process(i)

```
Process(i):
local constant: int j=3-i;
while(true){
    NCS;
    flag[i]=true;
    turn=j;
    wait-until( !flag[j] || turn==i );
    CS;
    flag[i]=false;
}
```

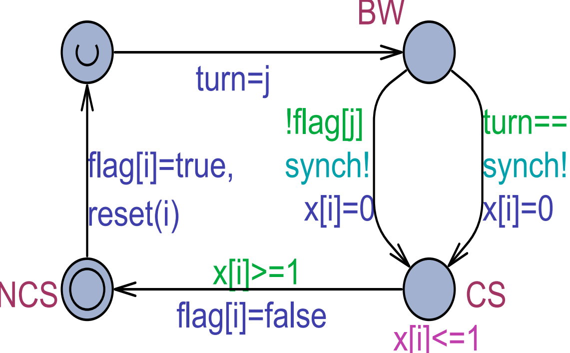

Figure 4. Uppaal model of Peterson's Process(i).

The Peterson algorithm was found to be fully correct from all the queries of Table 1, with an overtaking factor of 1. What is interesting, in the Peterson model, is that it does

not have any Zeno-cycle by construction. In the algorithm, a process i in busy-waiting finishes waiting as soon as the partner process j is not interested in the use of the resource or the turn is assigned to i. Any one of these conditions commands the exiting of process i from BW. On the other hand, when a process exits its CS, in the case it immediately starts competing again, not only signals this by setting its flag[] to true but also assigns the turn to the partner which, if in BW, immediately becomes unblocked while the first process certainly waits in its BW.

4.2. Solutions for $N \geq 2$ Processes

Several algorithms are reported in the literature. In the following, a subset of representative algorithms is considered, and their properties are thoroughly studied using the modeling and verification method proposed in this paper.

4.2.1. Dijkstra's Algorithm

The first solution to the mutual exclusion problem for $N \geq 2$ processes, numbered from 1 to N, was proposed by E.W. Dijkstra in [32]. The algorithm is based on the following shared variables:

bool [1..N] b, c; int k

The initial value of k, $1 \leq k \leq N$, is immaterial. If Process(i) enters its CS, k holds the value i. Process(i) modifies the values b[i] and c[i] which are inspected by the other processes. All the elements of the arrays b[] and c[] are initialized to the true value. The algorithm, reproduced as in [32], is shown in Algorithm 4. Informal arguments about its (not trivial) correctness were discussed in [32].

The Uppaal model of Algorithm 4 is depicted in Figure 5, whereas the adopted try() function is shown in Algorithm 5. As one can see, try() returns true if, optimistically, Process(i) could proceed toward its CS. After a true value of try(), the exact actions of the Entry section of Algorithm 4 are executed. Since in Li3 the value of b[k] has to be checked, the value of k is first read into the (at the moment free) local variable j; then, the value of b[j] is consulted. From Li3, the model always comes back to BW. To reduce the burden of the model checker operation, each time BW or NCS is re-entered, the local value of j is reset to 1. From Li4, if all the c[j], $1 \leq j \leq N$, $j! = i$, are true, the process moves to its CS.

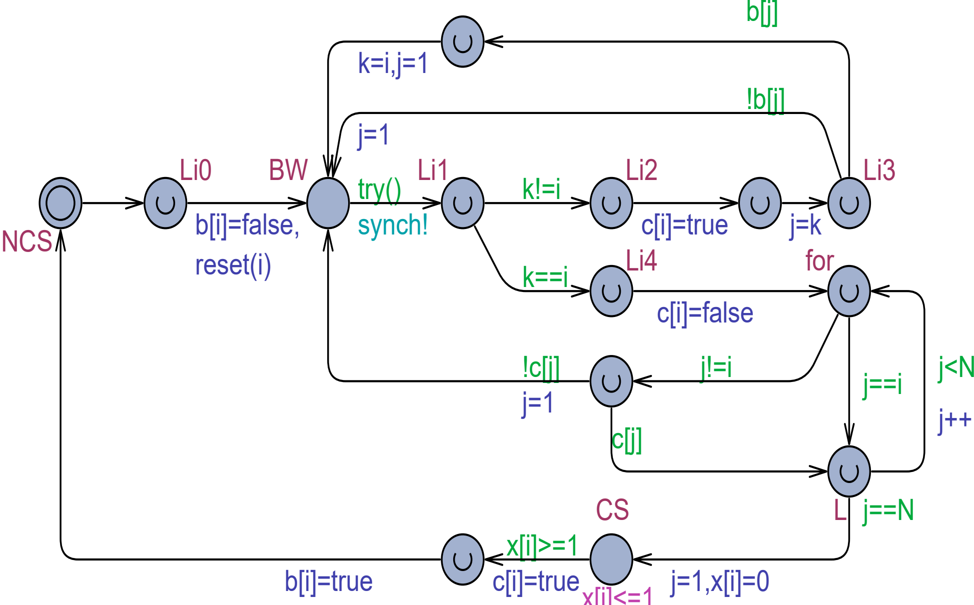

Figure 5. Uppaal model for Dijkstra's algorithm.

The model in Figure 5 was investigated for N varying from two to five. It emerged that the model satisfies the properties of absence of deadlocks and that of mutual exclusion (queries 1 and 2 of Table 1). In addition, a different behavior emerged from Process(1) from the other processes. Process(1) satisfies query 3 but not the other processes. Query 4 was also not verified whatever the target process was. The overtaking factor for $N = 3$ is unbounded (undefined) for processes i>1. This result, mentioned, e.g., in [33], complies with the implicit assumption in the design of Algorithm 4 that processes are expected to compete rather infrequently, as it may occur in some applications.

Algorithm 4. The E.W. Dijkstra Algorithm for the $N \geq 2$ Processes

```
Process(i):
local int j;
Li0:   b[i] := false;
Lil:   if k # i then
Li2:       begin c[i] := true;
Li3              if b[k] then k:=i;
                 go to Lil
             end
         else
Li4:         begin c[i] := false;
                 for j := 1 step 1 until N do
                 if j # i and not c[j] then go to Lil
             end;
           critical section;
           c[i] := true; b[i] := true;
           remainder of the cycle in which stopping is allowed;
           go to Li0
```

Algorithm 5. try() Function of Figure 5

```
bool try(){
    int j;
    if( k!=i ){ return false; }
    else{
        for( j=1; j<=N; ++j )
            if( j!=i && !c[j] )
                return false;
    }
    return true;
}//try
```

4.2.2. Knuth's Algorithm

This algorithm was proposed in [33] as an attempt to improve Dijkstra's solution described in [32]. It is based on the following shared communication variables

```
int [1,N] control; int k
```

both initialized to 0. A process i signals to partners about its status by writing a value in {0,1,2} in its control[i] variable. The value 0 mirrors that the process is not interested in the CS. The value 1 means the process wants to compete for the resource, thus starting the Entry section of Algorithm 1. Finally, the value 2 mirrors that the process is up to enter its CS. The algorithm is reproduced in Algorithm 6. The meaning of the for statement at line L1 is to first use the indices from k down to 1, by step -1, then to use the indices from N down to 1 step -1. It should be noted that initially, when k = 0, only the second part of the for from N down to 1 is executed.

Algorithm 6. Knuth's Algorithm for N ≥ 2 Processes

```
Process(i):
begin int j;
    L0:   control[i]=1;
    L1:   for j=k step −1 until 1, N step −1 until 1 do
              begin if j==i then go to L2;
                if control[j]!=0 then go to L1;
              end;
    L2:   control[i]=2;
           for j=N step −1 until 1 do
             if j!=i && (control[j]==2) then go to L0;
    L3:   k=i;
          critical_section;
          k=if i==1 then N else i−1;
    L4:   control[i]=0;
          remainder of the cycle in which stopping is allowed;
          go to L0;
end
```

The Uppaal model corresponding to Algorithm 6 is shown in Figure 6. The busy-waiting location coincides with L1. The try() function is reported in Algorithm 7 and basically executes the actions from L1 to L2 without state changes. The decrement function dec(j) is also shown in Algorithm 7.

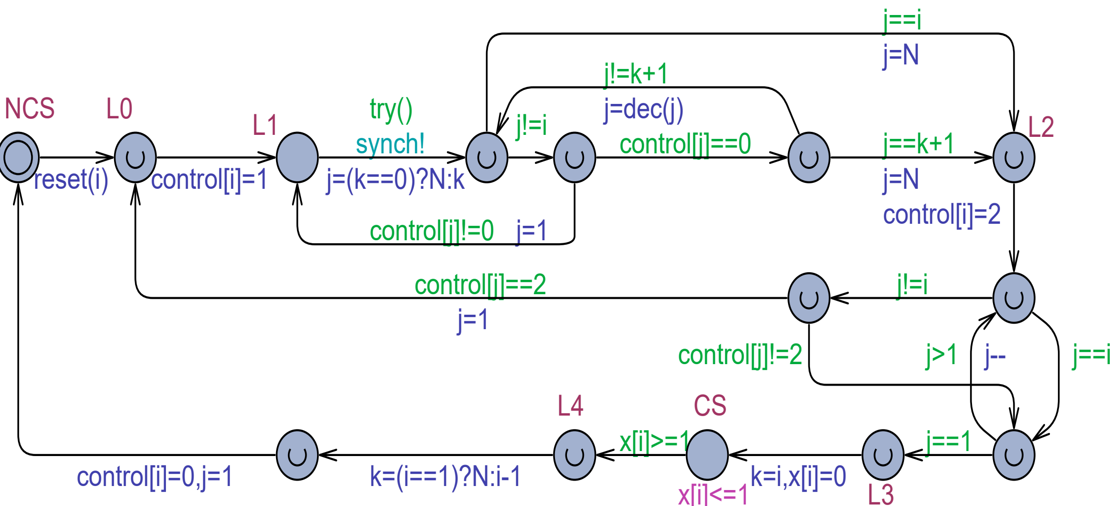

Figure 6. Uppaal model of Knuth's algorithm of Algorithm 6.

The correctness of the intricate Knuth's algorithm was informally discussed in [33], where it was predicted to have an overtaking factor of $2^{N-1} - 1$.

The Uppaal model of Figure 6 was extensively analyzed by model checking, with N ranging from 2 to 5. Queries 1, 2 and 4 of Table 1 were found satisfied. In particular, query 1 about the absence of deadlocks which requires the complete generation of the state graph, in the case $N = 5$ terminates after 265 sec with a RAM peak of 5GB. Query 3 of Table 1, stating that a competing process in L1 eventually reaches the CS location, was found to be not satisfied. This last property testifies the existence of a Zeno-cycle in the Entry part. By using our proposed time-sensitive model, it emerged that the number of observed overtakings, for varying N, is reported as in Table 3.

Algorithm 7. The try() and dec() Functions of the Uppaal Model in Figure 6

```
bool try(){
    int j;
    if( k>=i && i>=1 ){
        for( j=k; j>i; --j )
            if( control[j]!=0 ) return false;
    }
    else{ //k<i<=N
        for( j=N; j>i; --j )
            if( control[j]!=0 ) return false;
    }
    for( j=N; j>=1; --j )
        if( j!=i && control[j]==2 ) return false;
    return true;
}//try

int dec( int j ){
    if( k>0 ) return (j==1)?N:j−1;
    return j−1;
}//dec
```

Table 3. Observed overtakings vs. N for Knuth's algorithm.

N	ov
2	1
3	2
4	3
5	4

From Table 3, a linear number of overtakings $N-1$, not $2^{N-1}-1$, emerges. The same results were observed on the more expensive Knuth's model where the location L1 of Figure 6 is turned urgent and the try() function is ignored; thus, the maximum degree of interleavings is required by Algorithm 6. On the adapted model for U-SMC, query 4 of Table 2

```
Pr[<=10000](<>Process(tp).L1 && x[tp] > (N−1))
```

launched, e.g., for $N = 10$, proposes the confidence interval of an almost impossible event.

4.2.3. de Bruijn's Algorithm

N.G. de Bruijn suggested in [34] an improvement to Knuth's algorithm which consisted of a replacement of the three lines of the L3 instructions of Algorithm 6 with the following:

```
L3:   critical_section;
      if( control[k]==0 || k==i ) then k= (if k==1 then N else k−1);
```

In addition, the initial value of k is no longer 0 but can be any process identifier.

As a consequence of the above modifications, the new algorithm should ensure a bound on the overtakings of $\frac{1}{2}N(N-1)$. The Uppaal model of the de Bruijn algorithm is shown in Figure 7.

Model checking the de Bruijn model confirmed exactly the same basic properties as those of Knuth's model. The observed overtakings vs. N are shown in Table 4.

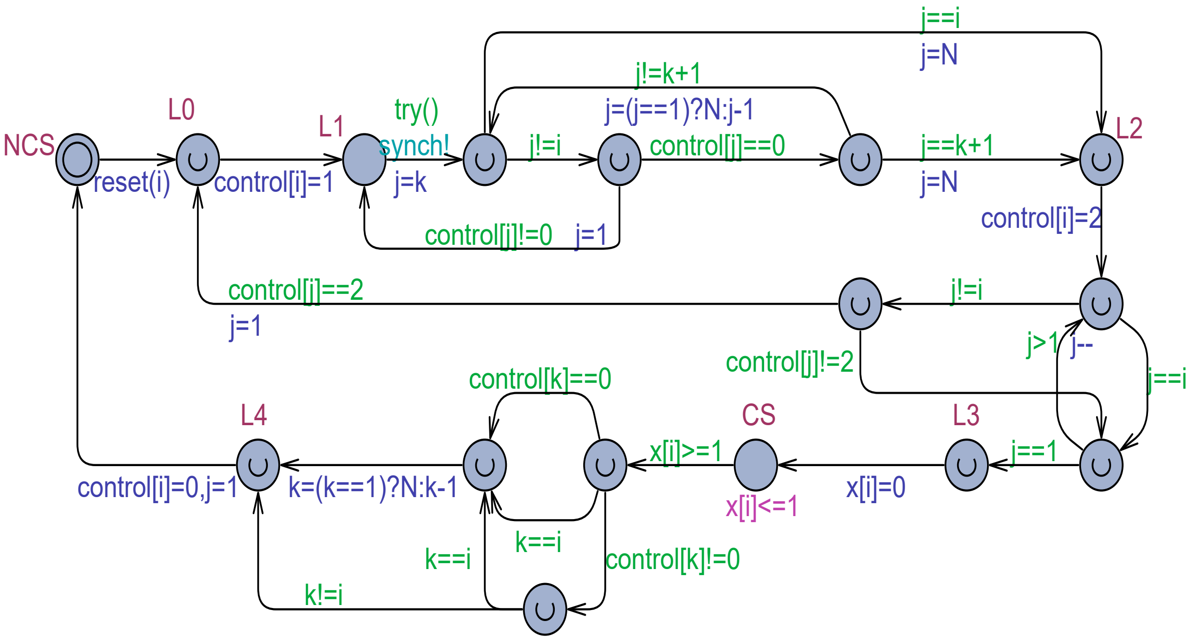

Figure 7. Uppaal model of de Bruijn's algorithm.

Table 4. Observed overtakings vs. N for de Bruijn's algorithm.

N	ov
2	1
3	3
4	5
5	7

By increasing N, it seems the overtakings tend to be (slightly) smaller than $\frac{1}{2}N(N-1)$. On the adapted model for U-SMC, the query

```
Pr[<=10000](<>Process(tp).L1 && x[tp]>(N*(N−1)/2))
```

for varying values of N responds with an event that is almost impossible.

4.2.4. The Eisenberg and McGuire Algorithm

A further improvement of Knuth's algorithm was proposed by Eisenberg and McGuire in [35]. It uses the same shared variables and the same initializations as in de Bruijn's algorithm (see Section 4.2.3). The new algorithm (see Algorithm 8) is committed to providing a linear overtaking factor of $N-1$.

The Uppaal model of Eisenberg and McGuire's algorithm is depicted in Figure 8. The try() function was adjusted in a straightforward way according to steps L1 to L3 of Algorithm 8. All the basic properties as in the Knuth and de Bruijn algorithms are satisfied. In addition, the liveness query (absence of starvation):

```
Process(1).L1 --> Process(1).CS
```

is now satisfied too. Table 5 reports the overtakings vs. N for the new algorithm, which confirms the expected trend of $N-1$.

Algorithm 8. Eisenberg and McGuire's Algorithm for N $\geq$ 2 Processes

```
Process(i):
begin int j;
   L0: control[i]=1;
   L1: for j=k step 1 until N, 1 step 1 until k do
          begin
             if j==i then go to L2;
             if control[j]!=0 then go to L1;
          end;
   L2: control[i]=2;
          for j=1 step 1 until N do
             if j!=i && (control[j]==2) then go to L0;
   L3: if control[k]!=0 && k!=i then go to L0;
   L4: k=i;
          critical_section;
   L5: for j=k step 1 until N, 1 step 1 until k do
             if j!=k and control[j]!=0 then
                begin
                   k=j;
                   go to L6;
                end;
   L6: control[i]=0;
          remainder of the cycle;
          go to L0;
end
```

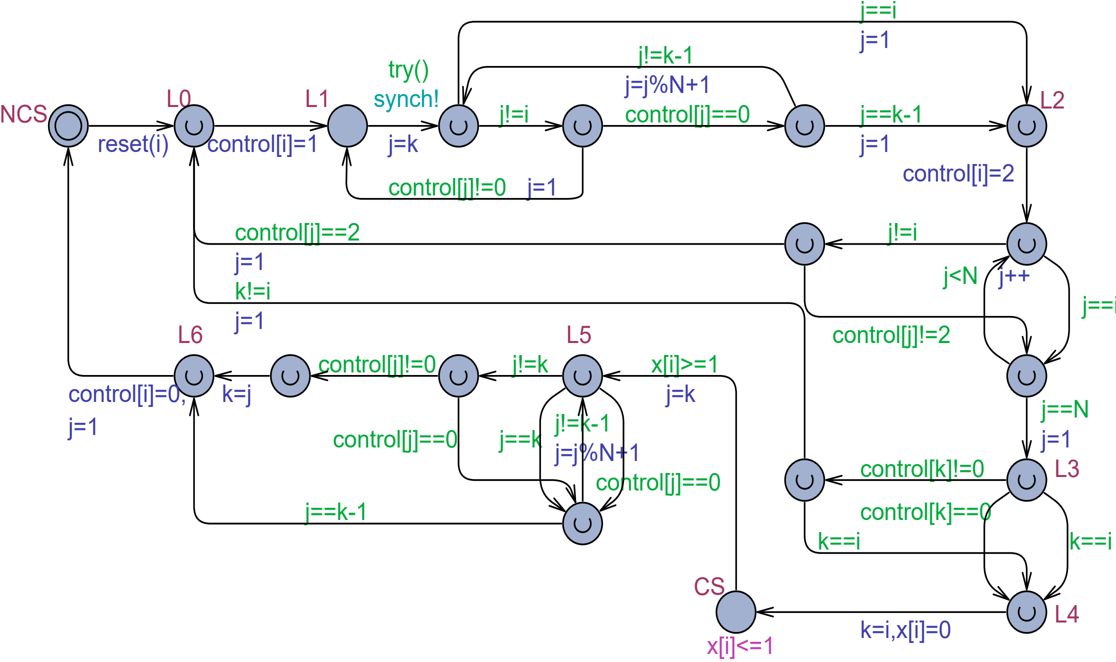

Figure 8. Uppaal model of de Eisenberg and McGuire's algorithm.

In the case N = 5, query 5 of Table 1 terminated in 147 sec with a RAM peak of about 5 GB.

The analysis using the Uppaal Statistical Model Checker of the model of Figure 8, adapted for simulation, confirmed, for N = 10, an estimation of the maximum value of the overtaking factor to be 9.

In light of the model-checking work, Eisenberg and McGuire's algorithm confirmed its superior character with respect to both the de Bruijn and Knuth algorithms, because it fulfills all the mutual exclusion properties and exhibits an overtaking factor, as expected,

of $N-1$. The algorithm is often indicated (see, e.g., [11]) as a reference for a correct and preferable mutual exclusion solution.

Table 5. Observed overtakings vs. N for Eisenberg and McGuire's algorithm.

N	ov
2	1
3	2
4	3
5	4

4.2.5. The Generalized Peterson's Algorithm

Peterson's algorithm (see Section 4.1.2) for two processes was generalized by his author to $N \geq 2$ processes as follows. Processes have unique numbers from 1 to N. There is a ladder of $N-1$ levels, numbered from 1 to $L = N-1$. Each process who wants to enter the critical section has to climb the ladder. The first process who reaches the last level grants the right to use the resource. A process moves to the next level, provided all the other processes occupy previous levels (competing processes or processes not interested in the use of the resource) (that is, (∀ k!=i, q[k]<j) or when another process enters the same level (that is, turn[j]!=i). The algorithm is reported in Algorithm 9. It is based on the global shared variables: int [0,N] q [1,N]; int [1,N] turn [1,N−1]

Algorithm 9. Peterson's solution for $N \geq 2$ processes

```
Process(i): local int j;
while( true ){
    NCS;
    for j=1 step 1 until N−1 do
        begin
            q[i]=j; turn[j]=i;
            wait-until(∀ k!=i, q[k]<j) or (turn[j]!=i) );
        end;
    CS;
    q[i]=0;
}
```

The corresponding Uppaal model is reported in Figure 9. The try() function is simply the following:

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

Queries 1 to 4 of Table 1 are satisfied. In particular, query 1, which checks for the absence of deadlocks, in the case $N = 5$, ends in 102 s with a memory peak of 2.3 GB.

Table 6 collects the overtaking factor for N ranging from 2 to 5. It emerged an overtaking bound of $\frac{1}{2}N(N-1)$. This result agrees with that reported in [3] but not with the papers [12,36] which claimed the bound to be $N-1$.

The correct and expected behavior of the algorithm for N > 5 was confirmed by using the queries of Table 2 on the model of Figure 9 preliminarily adapted for the Statistical Model Checker.

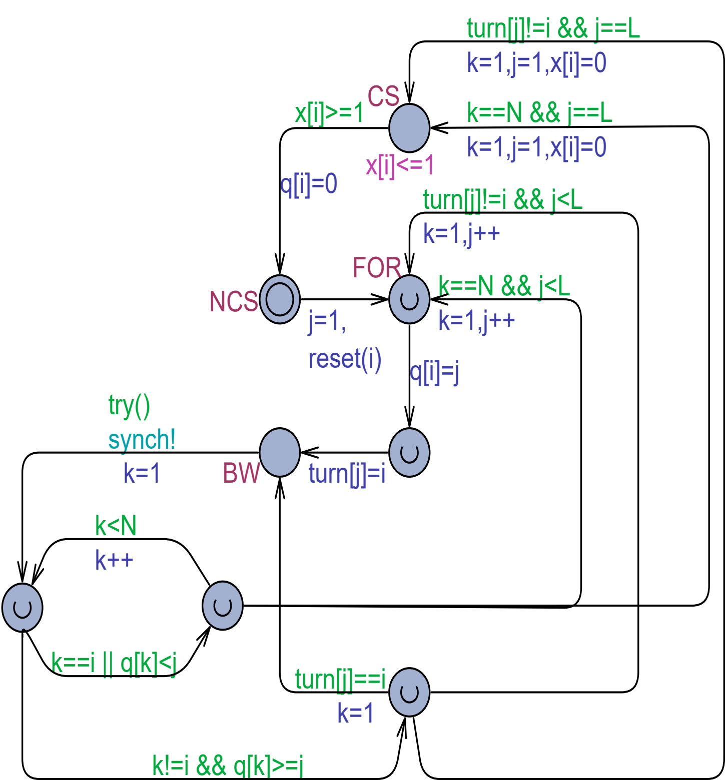

Figure 9. Uppaal model for Peterson's algorithm for N $\geq$ 2 processes.

Table 6. Observed overtakings vs. N for the generalized Peterson's algorithm.

N	ov
2	1
3	3
4	6
5	10

4.2.6. The Block and Woo Algorithm

This algorithm was proposed in [37] with the aim of improving the behavior of the generalized Peterson's algorithm. Using the same variable names as in Peterson's algorithm, the new shared variables are as follows:

bool q [1,N]; int [1,N] turn [1,N]

The array q[] instead of holding process ids now contains bool values (0 or 1). In addition, the ladder has N levels here, not N − 1. Differently from Peterson's solution, where a process has to reach the last level of the ladder for entering its CS, in the Block and Woo algorithm (see Algorithm 10), a process can obtain permission when it obtains a level "high enough" with respect to the number of competing processes. Should this number be m, the process goes into its CS when it reaches the level m. The Uppaal model corresponding to Algorithm 10 is depicted in Figure 10.

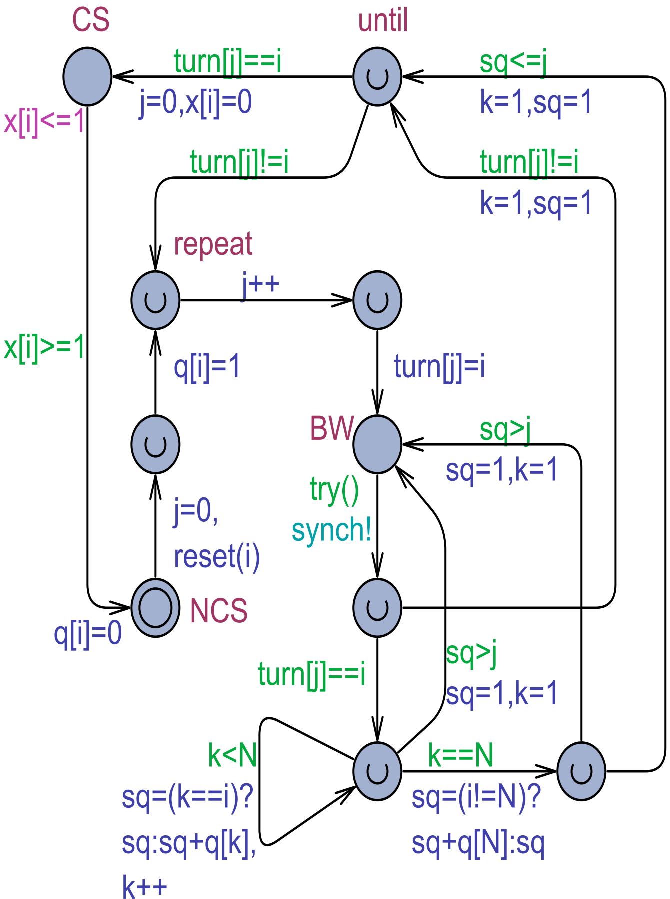

Figure 10. Uppaal model for the Block and Woo algorithm for $N \geq 2$ processes.

Algorithm 10. The Block and Woo Algorithm for $N \geq 2$ processes

```
Process(i): int j;
while( true ){
    NCS;
    j = 0;
    q[i] =1;
    repeat
        j = j+1;
        turn[j] =i;
        wait-until (turn[j]!=i) or (∑_{k=1}^{N} q[k] ≤ j);
    until (turn[j]==i);
    CS;
    q[i] =0;
}
```

The adopted try() function of Figure 10 is the following:

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

Model checking the Block and Woo algorithm for $N \leq 4$ and using the queries of Table 1 confirmed that it satisfies all the basic properties of a mutual exclusion algorithm. In addition, the solution does not have Zeno-cycles, and its overtaking factor was found to be $\frac{1}{2}N(N-1)$, in agreement with the evaluation provided by the authors in [37]. This number coincides with that which we have found for the generalized Peterson's algorithm (see Section 4.2.5); so, the new algorithm does not seem to be a real improvement. Using

the adapted model for statistical model checking, all the queries in Table 2 confirmed the qualitative and quantitative behavior of the algorithm for $N > 4$ also.

4.2.7. The Aravind and Hesselink Algorithm

This algorithm was proposed in [13] as a more efficient extension of the generalized Peterson's algorithm (see Section 4.2.5). The algorithm is very interesting, although it is heavier to analyze under model checking than either Peterson's algorithm or Block and Woo's algorithm. Its properties were assessed in [13] using the PVS theorem prover.

The algorithm's behavior can be summarized by using the "children party metaphor" with N children. The party room hosts N corners identified by the numbers from 0 to $N - 1$. At his arrival, a child reaches the corner $N - 1$. When a child at corner level finds that there are no more than k children (not counting itself) in the room, he/she can move to the corner k, provided $k <$ level. Reaching the corner 0 represents the entering of the critical section. At the corner 0, there is a table where a piece of cake can be picked to eat in the garden. A child can come back from the garden to the room if he/she feels like eating more cake. A chair is available at each corner level > 0, which a child can occupy. However, any child present at the same corner can climb the chair, thus possibly pushing a previous occupant. A pushed child then reaches the corner level $- 1$.

What is expensive in the algorithm is the children re-counting phase. The algorithm is based on the following shared data:

```
bool act [1,N]; pid turn [0,N−1];
```

which receive their default values at the initialization time. An abstract formulation of the algorithm is shown in Algorithm 11. An Uppaal model corresponding to Algorithm 11 is portrayed in Figure 11.

Algorithm 11. The Aravind and Hesselink Algorithm for Mutual Exclusion

```
Process(i):
local variables: int [0,N−1] level; bool bb; pid q; int [0,N−1] card;
while( true ){
    NCS;
    act[i]=true; level=N−1;
    while( level>0 ){
        card=N−1-|{q,1<=q<=N : !act[q]}|; //number of not CS interested processes
        if( card<level ){ level=card; bb=false; }
        else if( !bb ){ turn[level]=i; bb=true; card=|{q,1<=q<=N, q!=i && act[q]}|; }
        else if( turn[level]!=i ){ level--; bb=false; }
    }
    CS;
    act[i]=false;
}
```

The try() function for the model of Figure 11 is

```
bool try(){
        return level==0 || N−1-(sum(q:pid)!act[q])<level || !bb || turn[level]!=i;
}//try
```

Due to the use of many global and local variables in the model of Figure 11, there is a state explosion even for $N = 4$. For N = 2 and N = 3, all the TCTL queries 1 to 4 of Table 1 were found satisfied. Query 5 suggested a $N - 1$ overtaking factor, which exactly confirms the authors prediction in [13].

For this model, the MITL queries in Table 2 were useful for investigating the algorithm's behavior on the adapted model for U-SMC for N >= 4. In particular, all the responses to Table 2 queries confirmed expectations, with a bound on the overtakings of $N - 1$. As an example, for $N = 10$, Figure 12 reports the result generated by U-SMC for

query 2 of Table 2. To give an idea of the wall-clock time required by some U-SMC queries, query 4 of Table 2 furnishes an indication of an impossible event after 29 runs and 62 s.

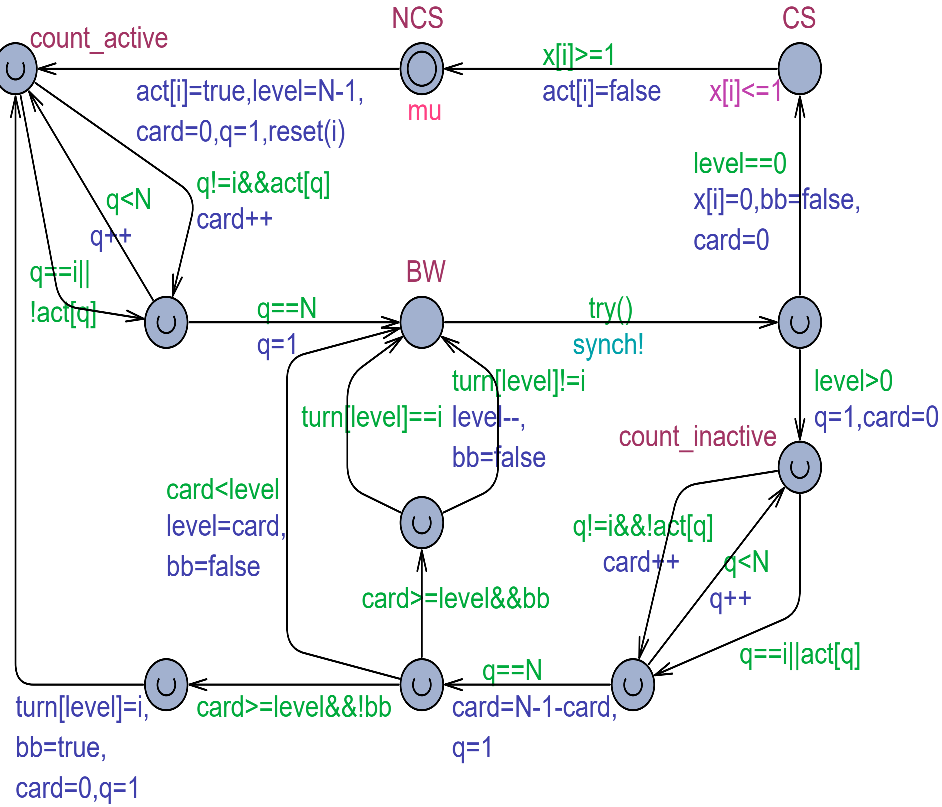

Figure 11. Uppaal model for the Aravind and Hesselink algorithm for $N \geq 2$ processes.

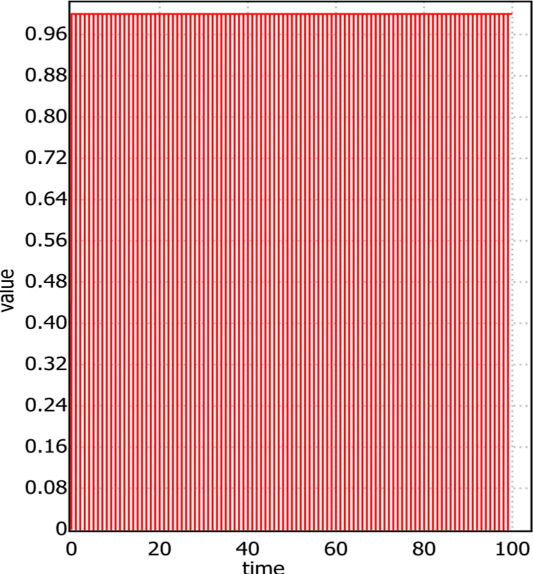

Figure 12. Number of processes vs. time, simultaneously found in their CS for $N = 10$.

4.2.8. Solutions under the Weak Memory Model

It has been pointed out, e.g., in [9,13,26], that the hypothesis of a RAM memory with atomic read/write operations is today almost ideal, especially when one considers the use of personal devices (e.g., cell phones) with multi-port memories [38]. In these cases, multiple reads can occur during a write operation on a given memory cell. The effect of

multiple readers on the same memory cell holding a shared variable *var*, to which the value *v* has to be assigned, can be modeled by returning a non-deterministic value (*flickering*) belonging to the type of *var*.

The presence of flickering in the model of a mutual exclusion algorithm increases the non-determinism and partial order in the state graph, which complicates the exploration of model checking. As a consequence, not only does the algorithm become more challenging to analyze, but it could lose its properties.

All the algorithms for N $\geq$ 2 processes previously studied in this paper were also checked under flickering. Only the generalized Peterson's algorithm, Block and Woo's algorithm and Aravind and Hesselink's algorithm were found to be robust and safe under the weak memory model. For brevity, the following only considers the Araving and Hesseling's algorithm/model modified with flickering (see Figure 13). As always, in the Uppaal model of Figure 13, the guards of edge commands are shown in green, the non-deterministic selection of a value belonging to its type is depicted in yellow and the update actions appear in blue.

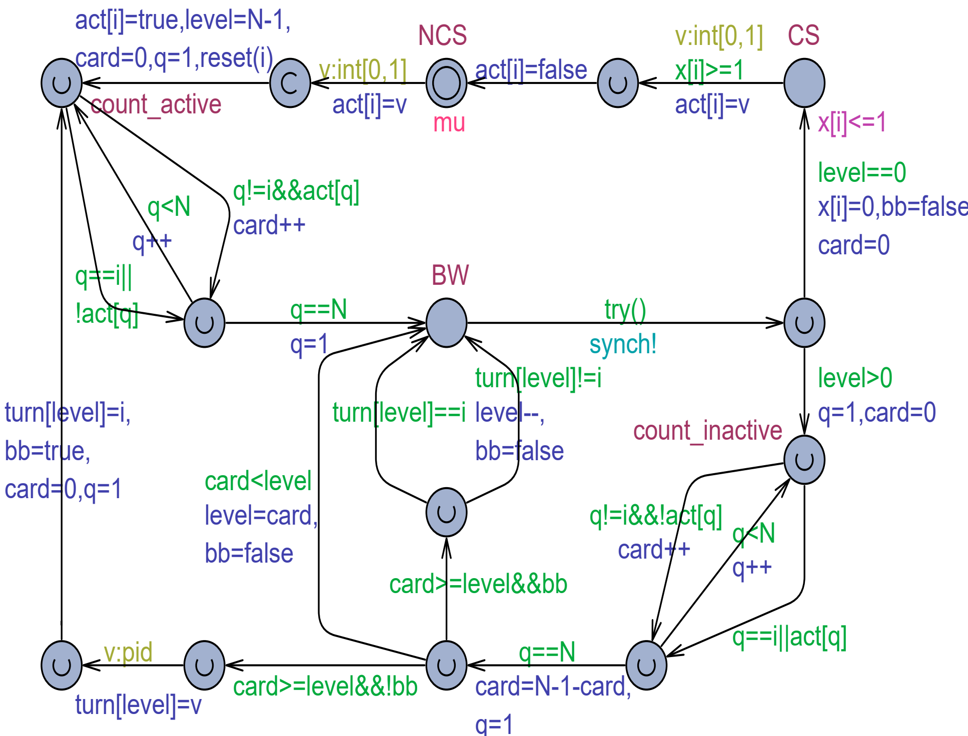

Figure 13. The Uppaal model of the Aravind and Hesselink algorithm under the weak memory model and flickering.

As an example of flickering, when the Process(i) automaton leaves its NCS in Figure 13, it has, among the other actions, to set (write) its act[i] shared variable to the true value. Due to flickering, though, a non-deterministic assignment of a value belonging to int [0,1] (the integers equivalent of false and true) is first anticipated, which is then followed by the correct assignment of act[i]=true. This way, it is perfectly possible, for non-determinism, that a reader can achieve the value defined by the flickering and not the right value.

The model of Figure 13 was analyzed by exhaustive model checking for N in [2,3] and by statistical model checking for N in [4,10]. It was found to be correct from all the TCTL and MITL queries, respectively, of Tables 1 and 2. What is interesting is that the bounded overtaking factor was confirmed to be N $-$ 1 as in the normal case without flickering. For the above properties, Aravind and Hesselink's algorithm emerged as one of the most safe and efficient mutual exclusion algorithms.

5. Conclusions

Reasoning on a concurrent/parallel software system [3,8,9] can be very complex. The origin of this difficulty stands in the non-determinism and action interleaving which dominate the evolution of the system. All of this can jeopardize the human intuition with which informal reasoning tries to predict the properties of a concurrent system.

This paper proposes an original approach based on formal modeling for the automated analysis of mutual exclusion algorithms, which are at the heart of concurrent/parallel, real-time operating systems. The approach is based on the Timed Automata [19] as implemented in the popular and continually evolving Uppaal toolbox [20]. The major limitation of the proposed method is the well-known state explosion problem, which arises in complex algorithms which make use of or generate too much data. As an example, solutions like Lamport's bakery algorithm [39], which grants access to the critical section on the basis of an ever-increasing service number, are difficult to be handled by exhaustive model checking; it could be studied by the Statistical Model Checker instead. The state explosion restriction forbids, even for addressable algorithms, the analysis when the number N of the involved processes is greater than 5. Another reason for having chosen Uppaal for the experiments rests on the fact that the tool also provides a Statistical Model Checker [24,25] which does not build the model state graph but instead relies on simulations and stochastic behavior. This way, the tool permits an exploration of the properties of a complex algorithm by estimating event probabilities which can be of great practical value by providing further arguments about the correctness of a given algorithm for non-trivial values of N.

This paper applies the proposed modeling and verification approach to several challenging algorithms. For all these algorithms, all the expected properties were found to be satisfied, except for the overtaking factor. This way, the known results defined in the literature were retrieved and confirmed. However, in some cases, unexpected results emerged regarding the bounded degree of the overtaking which a competing and waiting process can suffer before obtaining the permission to enter its critical section.

This paper investigates mutual exclusion algorithms from the point of view of a classic memory model with read/write atomic operations. However, modeling and verification aspects under the adoption of a weak memory model—that is, when multiple readers can simultaneous access an under writing memory cell with the uncertainty (*flickering*) [9,13] which accompanies the reading process—are also investigated.

Besides the reported algorithms, the proposed approach was successfully applied and its properties were retrieved for the following algorithms: Burns and Lamport [2,40], Peterson and Fisher [41], the state-of-the-art tournament tree (TT)-based solutions studied in [17] as well as the TT versions of the algorithms of Kessels [31], Hafidi et al. [18] and others.

The prosecution of the research will address the following points: first, to optimize the implementation of the approach and deepen the differences from alternative methods like the theorem provers which normally are unable to deal with timing aspects; second, to extend the approach for experimenting with more general contexts of memory consistency models (e.g., [42,43]); and third, to exploit parallel computing [44] for studying large models on a multi-core machine.

Author Contributions: Methodology, L.N.; Validation, L.N. and F.C.; Formal analysis, L.N.; Investigation, F.C.; Data curation, L.N.; Writing – original draft, L.N.; Writing – review & editing, F.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All the presented models can be achieved by writing to authors.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lamport, L. The mutual exclusion problem: Part I—A theory of interprocess communication. In *Concurrency: The Works of Leslie Lamport 2019*; ACM: New York, NY, USA, 2019; pp. 227–245.
2. Lamport, L. The mutual exclusion problem: Part II—Statement and solutions. In *Concurrency: The Works of Leslie Lamport 2019*; ACM: New York, NY, USA, 2019; pp. 247–276.
3. Raynal, M. *Algorithms for mutual exclusion 1986*; MIT Press: Cambridge, MA, USA, 1986.
4. Raynal, M.; Taubenfeld, G. A visit to mutual exclusion in seven dates. *Theor. Comput. Sci.* **2022**, *919*, 47–65. [CrossRef]
5. Buhr, P.A.; Dice, D.; Hesselink, W.H. High-performance N-thread software solutions for mutual exclusion. *Concurr.Comput. Pract. Exp.* **2015**, *27*, 651–701. [CrossRef]
6. Dijkstra, E.W. Co-operating sequential processes. In *Programming Languages: NATO Advanced Study Institute: Lectures Given at a Three Weeks Summer School Held in Villard-le-Lans*; Genuys, F., Ed.; Academic Press Inc.: Cambridge, MA, USA, 1966; pp. 43–112.
7. Raynal, M. *Concurrent Programming: Algorithms, Principles, and Foundations*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012.
8. Misra, J. A foundation of parallel programming. In *Constructive Methods in Computing Science: International Summer School Directed*; Bauer, M.F.L., Dijkstra, B.E.W., Hoare, C.A.R., Eds.; Springer: Berlin/Heidelberg, Germany, 1989; pp. 397–445.
9. Lamport, L. On interprocess communication. *Distrib. Comput.* **1986**, *1*, 77–101. [CrossRef]
10. Nigro, L.; Cicirelli, F. Formal modeling and verification of embedded real-time systems: An approach and practical tool based on Constraint Time Petri Nets. *Mathematics* **2024**, *12*, 812. [CrossRef]
11. Silbershatz, A.; Galvin, P.B.; Gagne, G. *Operating System Concepts*, 10th ed.; Wiley: Hoboken, NJ, USA, 2018.
12. Hofri, M. Proof of a mutual exclusion algorithm—A 'Class'ic example. *ACM SIGOPS Oper. Syst. Rev.* **1990**, *24*, 18–22. [CrossRef]
13. Aravind, A.A.; Hesselink, W.H. A queue based mutual exclusion algorithm. *Acta Inform.* **2009**, *46*, 73–86. [CrossRef]
14. Hesselink, W.H. Verifying a simplification of mutual exclusion by Lycklama–Hadzilacos. *Acta Inform.* **2013**, *50*, 199–228. [CrossRef]
15. Hesselink, W.H. Mutual exclusion by four shared bits with not more than quadratic complexity. *Sci. Comput. Program.* **2015**, *102*, 57–75. [CrossRef]
16. Hesselink, W.H. Correctness and concurrent complexity of the Black-White Bakery Algorithm. *Form. Asp. Comput.* **2016**, *28*, 325–341. [CrossRef]
17. Hesselink, W.H. Tournaments for mutual exclusion: Verification and concurrent complexity. *Form. Asp. Comput.* **2017**, *29*, 833–852. [CrossRef]
18. Hafidi, Y.; Keiren, J.J.; Groote, J.F. Fair Mutual Exclusion for N Processes. In *International Conference on Software Testing, Machine Learning and Complex Process Analysis*; Springer Nature: Cham, Switzerland, 2021; pp. 149–160.
19. Alur, R.; Dill, D.L. A theory of timed automata. *Theor. Comput.Sci.* **1994**, *126*, 183–235. [CrossRef]
20. Behrmann, G.; David, A.; Larsen, K.G. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*; Bernardo, M., Corradini, F., Eds.; LNCS 3185; Springer: Berlin/Heidelberg, Germany, 2004; pp. 200–236.
21. Clarke, E.M. Model checking. In Proceedings of the Foundations of Software Technology and Theoretical Computer Science: 17th Conference, Kharagpur, India, 18–20 December 1997; Proceedings 17. Springer: Berlin/Heidelberg, Germany, 1997; pp. 54–56.
22. Cicirelli, F.; Nigro, L. Modelling and verification of mutual exclusion algorithms. In Proceedings of the IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), London, UK, 21–23 September 2016; pp. 136–144.
23. Cicirelli, F.; Nigro, L.; Sciammarella, P.F. Model checking mutual exclusion algorithms using Uppaal. In Proceedings of the 5th Computer Science On-Line Conference, Zlin, Czech Republic, 27–30 April 2016; Springer: Berlin/Heidelberg, Germany, 2016.
24. Agha, G.; Palmskog, K. A Survey of Statistical Model Checking. *ACM Trans. Model. Comput. Simul.* **2018**, *28*, 39.
25. David, A.; Larsen, K.G.; Legay, A.; Mikucionis, M.; Poulsen, D.B. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.* **2015**, *17*, 397–415. [CrossRef]
26. Buhr, P.A.; Dice, D.; Hesselink, W.H. Dekker's mutual exclusion algorithm made RW-safe. *Concurr.Comput. Pract. Experience* **2016**, *28*, 144–165. [CrossRef]
27. Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* **1994**, *16*, 872–923. [CrossRef]
28. Bowman, H.; Gomez, R.; Su, L. A tool for the syntactic detection of zeno-timelocks in Timed Automata. *Electron. Notes Theor. Comput. Sci.* **2005**, *139*, 25–47. [CrossRef]
29. Lee, E.A. Modeling in engineering and science. *Commun. ACM* **2018**, *62*, 35–36. [CrossRef]
30. Peterson, G.L. Myths about the mutual exclusion problem. *Inf. Process. Lett.* **1981**, *12*, 115–116. [CrossRef]
31. Kessels, J.L.W. Arbitration without common modifiable variables. *Acta Inform.* **1982**, *17*, 135–141. [CrossRef]
32. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Commun. ACM* **1965**, *8*, 569. [CrossRef]
33. Knuth, D.E. Additional comments on a problem in concurrent programming control. *Commun. ACM* **1966**, *9*, 321–322. [CrossRef]
34. de Bruijn, N.G. Additional comments on a problem in concurrent programming control. *Commun. ACM* **1967**, *10*, 137–138. [CrossRef]
35. Eisenberg, M.A.; McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. *Commun. ACM* **1972**, *15*, 999. [CrossRef]
36. Kowaltowski, T.; Palma, A. Another solution of the mutual exclusion problem. *Inf. Process. Lett.* **1984**, *19*, 145–146. [CrossRef]

37. Block, K.; Woo, T.K. A more efficient generalization of Peterson's mutual exclusion algorithm. *Inf. Process. Lett.* **1990**, *35*, 219–222. [CrossRef]
38. Frenzel, L.E. Dual-port SRAM accelerates smart-phone development. In *Electronic Design*; Endeavor Business Media: Nashville, TN, USA, 2004.
39. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* **1974**, *17*, 453–455. [CrossRef]
40. Burns, J.E.; Lynch, N.A. Mutual exclusion using indivisible reads and writes. In Proceedings of the 18th Allerton Conference on Communication, Control and Computing, Monticello, IL, USA, 8–10 October 1980; pp. 833–834.
41. Peterson, G.L.; Fischer, M.J. Economical solutions for the critical section problem in a distributed system. In Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 4 May 1977; pp. 91–97.
42. Sorensen, T.; Gopalakrishnan, G.; Grover, V. Towards shared memory consistency models for GPUs. In Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, New York, NY, USA, 10–14 June 2013; pp. 489–490.
43. Mascarell, J.B. Formal Definitions of Memory Consistency Models. *arXiv* **2021**, arXiv:2101.09527.
44. Nigro, L. Parallel Theatre: An actor framework in Java for high performance computing. *Simul. Model. Pract. Theory* **2021**, *106*, 102189. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.