mathematics

MDPI

Article

Formal Modeling and Verification of Lycklama and Hadzilacos's Mutual Exclusion Algorithm

Libero Nigro

DIMES Department, University of Calabria, 87036 Rende, Italy; libero.nigro@unical.it

Abstract: This study describes our thorough experience of formal modeling and exhaustive verification of concurrent systems, particularly mutual exclusion algorithms. The experience focuses on Lycklama and Hadzilacos's (LH) mutual exclusion algorithm. LH rests on the reduced size of the shared state, contains a mechanism that tries to enforce an FCFS order to processes entering their critical section, and embodies Burns and Lamport's (BL) mutual exclusion algorithm. The modeling methodology is based on timed automata and the model checker of the popular Uppaal toolbox. The effectiveness of the modeling and analysis approach is first demonstrated by studying the BL's solution and retrieving all its properties, including, in general, its unbounded overtaking, which is the non-limited number of by-passes a process can suffer before accessing its critical section. Then, the LH algorithm is investigated in depth by showing it fulfills all the mutual exclusion properties when it operates with atomic memory. However, as this study demonstrates, LH is not free of deadlocks when used with non-atomic memory. Finally, a state-of-the-art mutual exclusion solution is proposed, which relies on a stripped-down LH version for processes, which is used as the arbitration unit in a tournament tree (TT) organization. This study documents that LH's TT-based algorithm satisfies all the mutual exclusion properties, with a linear overtaking, both using atomic and non-atomic memory.

Keywords: concurrency/parallelism; mutual exclusion algorithms; properties; atomic and non-atomic memory models; formal modelling; exhaustive model checking; Uppaal

MSC: 68Q60; 68Q25; 68Q45; 68M20; 68W10; 68W15

Citation: Nigro, L. Formal Modeling and Verification of Lycklama and Hadzilacos's Mutual Exclusion Algorithm. *Mathematics* **2024**, *12*, 2443. https://doi.org/10.3390/math12162443

Academic Editor: Janez Žerovnik

Received: 10 July 2024
Revised: 1 August 2024
Accepted: 2 August 2024
Published: 6 August 2024

1. Introduction

Mutual exclusion [1,2] is a major problem in concurrent/parallel and distributed systems [3,4]. Its fundamental form can be stated as follows. There are $N \geq 2$ processes, e.g., running on distinct cores of a multi/many-core machine, and a shared resource R. During their execution, processes can require access to R. For predictability, only one process at a time can be given the permission to access and possibly modify R. The code segment of a process containing the operations on R is said to be its *critical section* (CS). Each critical section concerning R must be executed atomically, thus excluding multiple processes from simultaneously accessing R, which could imply that R becomes corrupted.

A mutual exclusion algorithm provides a protocol that regulates what to do when a process raises its intention to use R (competition or entry part of the protocol) and, after having achieved the permission to enter its CS, what operations to execute at the exit from its CS (exit part of the protocol). In this study, pure-software-based solutions for mutual exclusion are considered, which are not based on hardware instructions such as *test-and-set* and the like. It has been pointed out in the literature that software-based solutions can be designed, whose efficiency is comparable to hardware-assisted solutions.

The first mutual exclusion solution for $N = 2$ processes was invented by T.J. Dekker [5,6]. An algorithm for $N > 2$ processes was then developed by E.W. Dijkstra in [7]. A better solution, with the guarantee of bounded waiting for a competing process (a property that was missing in Dijkstra's solution), was defined by D. Knuth [8]. Knuth's solution was further improved

by deBruijn [9] and by Eisenberg and McGuire [10]. New solutions for $N = 2$ and $N > 2$ were proposed by G.L. Peterson [11] and Block and Woo [12]. All these mutual exclusion algorithms were studied informally and under atomic memory only, where read/write operations are assumed to be indivisible. As shown recently in [13], only the Peterson and the Block and Woo algorithms were found to be safe under non-atomic or weak memory [14], where, e.g., multiple read operations can occur simultaneously to a write operation on the same variable. In this circumstance, a read can return (by *flickering*) a non-deterministic value belonging to the type of variable. Non-atomic memory is today widespread in smartphones and similar devices equipped with multi-port memories [15].

Proving the properties of a mutual exclusion algorithm can be very difficult, because of the non-determinism (*partial order*), which characterizes the action execution order (interleaving) of the involved concurrent processes. This explains why testing and informal/intuitive reasoning can easily fail in assessing the true properties of a mutual exclusion solution. Two main formal methods prevail in the literature: (a) the *assertional method* [16], which relies on a transition system derived from a formal, mathematical characterization of the mutual exclusion algorithm, upon which assertions can be issued, which are analyzed with the help of a theorem prover as a proof assistant; (b) *model checking* [17–21], which depends on a formal model, e.g., based on the timed automata (TA) [18,22], from which the state graph of all the possible execution states of the model is automatically built. Efficient navigation algorithms can then be used, driven by queries expressed in a temporal logic language. Assertional methods can be difficult to apply to complex algorithms. Model checking, however, can suffer from the state explosion problem, which occurs when the model has too many data to handle, e.g., tied to a non-small number N of processes. However, model checking based on timed automata [18,20] can rely on an intuitive graphical form, which can help model reasoning and understanding. In addition, the possibility of exploring the effects of time on a model (see later in this study) can be of great help to disclose subtle aspects of a mutual exclusion algorithm. Theorem provers, on the other hand, normally are not able to deal with timing aspects.

The work described in this study develops a modeling and verification approach that is based on the Uppaal timed automata and model checker [18]. The approach has been successfully applied to the property checking of different mutual exclusion algorithms [13,23].

The contributions of this study concern an original and thorough investigation of the Lycklama and Hadzilacos (LH) solution [24], which is characterized by its reduced shared space (only $5N$ bits), and its structure, which is composed of two parts: an outer part whose code tries to ensure the First-Come-First-Served (FCFS) order among processes, which compete for entering their critical section, and an inner part, which is based on Burns and Lamport's (BL) mutual exclusion algorithm [1,25,26].

The flexibility and power of the adopted modeling and verification approach are first demonstrated by a thorough analysis of BL's solution, which was independently discovered by J.E. Burns [25,26] and by L. Lamport [1] (part II, page 17). BL requires a minimal shared data space (N bits) and works correctly under both atomic and non-atomic memory. All the properties of BL, for $N \geq 2$ processes, are retrieved and are in total agreement with the predictions reported, e.g., in [1] (part II). In particular, this study confirms the general unbounded overtaking (or unbounded waiting) of competing processes that is concerned with the number of by-passes a process can experiment with before (hopefully) entering its critical section. The same properties are retrieved under both atomic and non-atomic memory.

This study continues by modeling and analyzing Lycklama and Hadzilacos's (LH) [16,24,27] algorithm. As this study demonstrates, LH fulfills all the properties of a mutual exclusion algorithm, including a bounded overtaking (also called an "absence of individual starvation" for processes). In a sense, the FCFS provision in LH seems to correct the unbounded overtaking suffered by the underlying BL algorithm. However, the correct behavior of LH refers only to its use with atomic memory. In fact, as this study shows, when LH is adapted for working with non-atomic memory, it loses the fundamental

property of being deadlock-free. It should be noted that both the studies in [16,27] analyze LH by informal mathematical reasoning in [26] and by the assertional method assisted by the PVS theorem prover in [16], with the aim of minimizing the size of the shared data space. However, this study argues that the proposed model checking approach is unique in revealing a weakness of LH when it operates with non-atomic memory.

As a further original contribution, this study proposes a standard tournament tree (TT)-based solution [23,28,29] that uses the LH version for $N = 2$ processes (LH2) as the arbitration unit in the TT. This new algorithm is studied for $N \geq 2$ processes. This study proves that the achieved realization based on TT and LH2, which uses $4(N + 2^{\lceil \log_2 N \rceil} - 1)$ bits of shared data space, is a fully correct mutual exclusion solution under both atomic and non-atomic memory.

The rest of this study is organized as follows. Section 2 discusses the basic concepts and properties of a mutual exclusion algorithm. Section 3 illustrates the modelling approach based on Uppaal. As a concrete example, Burns and Lamport's (BL) solution is formally modeled and thoroughly analyzed under both the atomic and non-atomic memory models. Section 4 contains an in-depth modeling and verification work of Lycklama and Hadzilacos's (LH) algorithm. LH weakness when used with non-atomic memory is demonstrated. Section 5 verifies the correctness of an original solution based on a tournament tree, which rests on LH for two processes as the arbitration unit. Finally, Section 6 concludes this study with an indication of ongoing and future work.

2. Properties of a Mutual Exclusion Algorithm

2.1. Process Structure

The design of a mutual exclusion algorithm (see Algorithm 1), often very difficult to grasp intuitively, depends on a (hopefully very small) number of *shared communication variables* and a *protocol*, which states the operations to execute to enter/compete for the achievement of the permission to use the shared resource, and when exiting from the critical section. In the entry part, and sometimes also in the exit part, a process can become *busy-waiting*; that is, it can, by wasting cpu/core cycles, continually check the shared variables until a condition is satisfied for abandoning the busy-waiting. In a preferable case [29], the shared communication variables are associated one to one with the processes. Each process can change its variable, but all the processes can check all the communication variables.

Algorithm 1. The abstract structure of a process engaged with mutual exclusion.

```
shared communication variables
Process(i):
local variables of process i
repeat
   NCS;
   protocol-entry-part;
   CS;
   protocol-exit-part;
forever
```

The abstract structure of a process involved in a mutual exclusion situation is shown in Algorithm 1. Unique process IDs are assumed to be from 1 to N, with $N \geq 2$. The non-critical section (NCS) denotes the code segment where the process has no interest in accessing the shared resource. The NCS duration can be any value in $[0, \infty]$. An infinite duration expresses the process, which stops executing within the NCS. This case is important to explicitly consider because it must be proved that a terminated process does not impede other processes from using the resource.

2.2. Safety and Liveness Properties

A safety property wants to exclude that a bad/hazardous state is never reached in the system evolution. A liveness property aims to ensure that a good state is eventually

reached. When the good state is reached within a given time frame, the property is said to be of a bounded liveness type. A correct mutual exclusion algorithm needs to satisfy all the following properties.

1. (*Safety*) one process, at most, at any time, can be executing its critical section (CS);
2. (*Safety*) the protocol must ensure all the execution states are *free of deadlock*, that is, the fatal circular situation in which each process waits for an action to be executed by another process, which never arrives;
3. (*Liveness*) all the processes eventually enter their CS;
4. (*Bounded liveness*) a competing and waiting process eventually enters its CS in a bounded time (*absence of individual starvation*);
5. (*Liveness*) a process executing within in its NCS must not impede another process from entering its CS;
6. (*Liveness*) no assumption is made on the process relative speeds.

The main goal of the formal modeling and verification approach described in this study is to enable the *automated assessment* through model checking of the properties of a mutual exclusion solution, thus going beyond testing or informal reasoning based on traditional mathematics. Model checking, indeed, is relatively efficient when applied in practice. It is far more doable than formally proving correctness. Moreover, it is very good at finding mistakes in protocols, even if only applied to smaller instances. This is not the case when using simulation, which may or may not be effective. Therefore, massive use of model checking can allow us to become far more productive in designing correct distributed and parallel protocols.

3. Modelling Issues

The proposed method for transforming a mutual exclusion algorithm into a formal model is based on the high-level timed automata (TA) [22] language supported by Uppaal [18]. Integer and boolean data variables and arrays of these primitive types can be introduced globally or locally to a process, together with C-like functions, which can contribute to compact and more readable models. Each process is modeled as an independent timed automaton instance that interacts with peers solely through the global shared communication variables. The behavior of a process is specified by a state machine composed of *locations* (local states) and *edges* (transitions among locations). An edge is tagged by a *guarded command*, which has three major (and optional) attributes: a *guard*, a *synchronization* and an *update*. The *guard* (true if absent) is a boolean expression that indicates the eligibility (enabledness) of the transition to be taken. The *synchronization* is a channel input (!)/output (?) operation. For this study, only broadcast channels are considered with asynchronous communications. The *update* consists of an ordered list of variable assignments and clock resets.

A command constitutes the fundamental *atomic action* of the Uppaal concurrent language. The timing behavior of a model is controlled by clocks. A clock can be reset and then it advances automatically, thus measuring the relative time elapsed from its last reset. Uppaal models rely on global time. All the clocks grow with the same advancement rate. A fundamental concept is the time spent by an automaton in a location. A normal location is one where the automaton can remain for an arbitrary time, from 0 to ∞. To improve the progress of a model, an *invariant* (e.g., a clock constraint) can be attached to locations. An automaton can stay in a location equipped with an invariant, as long as the invariant remains true. In the instant the invariant is up to be falsified, the location has to be abandoned immediately; otherwise, a deadlock occurs. A particular location is the urgent one (flagged with a U). It is equivalent to a normal location with a clock invariant as follows: $x \leq 0$. Clock x is supposed to be reset at the entrance to the location. When in a model where multiple timed automata exist, which are in urgent locations at the same time, the exit order from these locations is non-deterministic. Another progress measure can be achieved by sending asynchronous communication over an urgent broadcast channel, even without receivers. The exit order from urgent locations and from a normal location where

an edge exists with a true guard and an output synchronization on an urgent broadcast channel remains totally non-deterministic.

Property assessment is carried out by the model checker, by raising TCTL [18] queries, which capture the fulfillment of distinct (existential or to be always satisfied) properties. Uppaal builds the state graph of a model, whose nodes capture all the possible execution states of the model. A TCTL query implies navigation of the state graph aiming at proving or disproving a property. As a preliminary, intuitive phase, a Uppaal model can also be animated in the symbolic simulator, where particular sequences of events and their effects on the model data variables can be inspected. All of this can help the modeler understand the model's behavior.

3.1. Modelling a Mutual Exclusion Algorithm

As a concrete example of how a mutual exclusion algorithm can be translated into a Uppaal model for property verification, Algorithm 2 shows Burns and Lamport's solution (BL) for $N \geq 2$ processes. The processes, uniquely identified by the IDs from 1 to N, share the following communication variables:

bool X [1..N], initialized to all false (default)

Each variable $X[i]$ can be written only by process i but can be checked by the remaining processes. Assigning *true* to $X[i]$ means the process's i is interested in entering its critical section. Basically, to achieve permission to enter its CS, a process i must go over two busy-waiting cases. The first one is repeated as there exists a process j, $1 \leq j < i$, which has $X[j] == false$. In this case, first $X[i]$ is lowered to $false$; then, the condition $X[j] == false$ is awaited. After that, the algorithm is repeated from the L point. The second case starts a busy waiting as any process j, from $i < j \leq N$ in that order, is found with $X[j] == false$. It is worth noting that the instruction

await-until (cond) is equivalent to: **while** (!cond);

At the end of the critical section, each process resets its $X[i]$.

The correctness of the BL algorithm in Algorithm 2 was discussed in [1] (part II) through informal mathematics reasoning. As is normally the case, diving into the logic of a mutual exclusion solution, and predicting its properties, can be a difficult task. In this study, to help our understanding and to favor a systematic assessment of the properties, an algorithm like Algorithm 2 is first translated into an equivalent Uppaal model (see Figure 1), according to a few rules as follows.

Algorithm 2. Burns and Lamport's solution for $N \geq 2$ processes.

```
Process (i)
local variable: int [1..N] j;
repeat
    NCS;
L: X[i] = true;
    For (j =1 to i − 1){
      If (X[j]){
        X[i] = false;
        await-until (!X[j]);
        goto L;
      }
    }
    For (j = i + 1 to N) await-until (!X[j]);
    CS;
    X[i] = false;
forever
```

1. All the actions referring to shared communication variables must be associated with guarded commands attached to the edges exiting from Uppaal locations. Whatever the memory model, only one shared variable can be read/written in the same command, to testify single memory accesses. This rule can be relaxed for the local variables of a process, where one can think the variables are held in distinct registers and can be freely accessed by the process.
2. To enforce non-determinism and action interleaving in the execution of the concurrent processes, the source locations of the actions can be purposely realized as urgent locations, thus witnessing a negligible action duration. This rule is not adopted in the modeling of the non-critical section (NCS), the critical section (CS), and a busy-waiting situation.
3. The NCS is represented by a normal location whose dwell time can be any time in [0,∞]. The exit from NCS can, thus, be a spontaneous edge.
4. The CS is represented by a normal location with permanence of exactly 1-time unit. For this purpose, each process is associated with a clock variable z, which is reset just before entering the CS. The invariant $z[i] \leq 1$ is attached to CS, and the exit from CS is guarded by the clock constraint $z[i] \geq 1$. This construction was adopted to simplify the prediction of the overtaking bound (ov). In particular, ov is measured regarding a target process (tp). Any process ID can be chosen as tp. The clock of tp is reset as soon as tp exits from the NCS (see the function $reset(i)$ in Figure 1) and starts competing. Of course, as other competing processes by-pass (overtake) the competing tp and enter their CS, then the ov is incremented by 1. The maximal value of $z[tp]$ observed when tp is in the position to enter its CS (see the location E in Figure 1) furnishes the (hopefully finite) overtaking bound.
5. To reduce the amount of non-determinism in the model, a busy-waiting case is modeled by a normal location (see BW1 and BW2 in Figure 1) from which an exit is forced as soon as the awaited condition holds. This provision avoids, in the Uppaal model, the active and very often unproductive spin cycles, which would waste the model checking work (navigation of the state graph). To this purpose, an urgent and broadcast channel is used (see the channel *synch* in Figure 1), upon which an output send operation (!) is commanded, which is received by no other process (asynchronous communications). In this way, exiting from a busy-waiting location becomes an urgent action not distinguishable from the actions tied to urgent locations. All these actions will occur in a non-deterministic way.

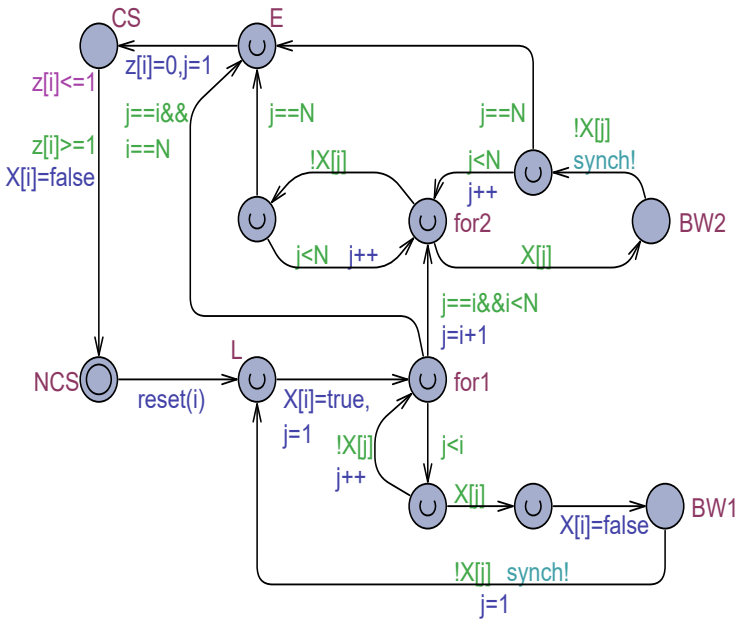

Figure 1. A Uppaal model for the BL's solution in Algorithm 1.

A particular case of rule 5 happens when a busy-waiting condition is a complex one, involving the evaluation of multiple shared variables linked by the usual logical operators: and (&&), or (||), not (!). Since it is not possible to evaluate in a single guard one, such a condition, a $try()$ function (see Figure 5), can be used, which optimistically checks the overall awaited condition in one step. However, when $try()$ returns $true$, a detailed evaluation has to be carried out, by assembling the various components of the condition piece by piece according to the logical operators. Due to non-determinism, at any time of the detailed evaluation, the condition can be proved to be $false$. In this case, the busy-waiting location is immediately re-entered.

3.1.1. Semantic Aspects

Some semantic issues concerning the above-mentioned transformation rules deserve some further comments.

(a) If N are the processes, N clocks are used in the model. However, if the algorithm is a correct mutual exclusion solution, at any moment, only two clocks can be active and affecting the model checker: that of the competing target process $z[tp]$ and that of another process entering its CS.

(b) The use of time in the CS location in no case alters the natural evolution of the mutual exclusion algorithm. If process p is in its CS, any other competing process $q! = p$, due to non-determinism and urgent actions certainly reaches a busy-waiting location from which it can exit when p, at the end of its CS, finally changes some shared communication variables. Since exiting from a busy-waiting location normally modifies shared variables, a cascade effect can occur on other processes in busy waiting and so forth.

(c) Also, the timing in the NCS location complies with the correct behavior of the algorithm. If Δ_{ov} is the bounded overtaking (time duration) of the algorithm, two cases can be considered. The process in NCS exits at a time instant lower than Δ_{ov}. In this case, the process actively participates in the current competition and contributes to the definition of Δ_{ov}. If the process leaves NCS after Δ_{ov}, it will then attend the next competition.

(d) The above points (b) and (c) shed some light on the importance of using a time-sensitive framework in the proposed Uppaal-based modeling approach. For example, after having assessed (as expected) that the termination of a process in the NCS does not alter the correct behavior of the mutual exclusion model, a next analysis phase can be conducted by making NCS urgent, thus forcing a process that exits its CS to immediately re-enter the system and start competing. Making NCS urgent, on the other hand, can improve the model checker work by simplifying the execution paths that depend on the time spent in NCS, which, in turn, can make the model more scalable in the number N of the admitted processes.

3.1.2. Uppaal Model Details

The following reports the global declarations required by the BL model in Figure 1:

```
const int N = 4; //example
typedef int[1,N] pid; //type of the process IDs
urgent broadcast chan synch;
//shared communication variables
bool X[pid]; //all false initially by default
//process clocks
clock z[pid];

pid tp = 2; //target process example
void reset( const pid i ){
if(i == tp) z[tp] = 0;
```

}//reset

The *Process* automaton admits one single parameter, *const pid i*, and declares the local variable *j* (of type *pid*), which is used for controlling the two loops in Algorithm 2. Due to the adopted parameter, the following system declaration

$$system\ Process;$$

automatically creates, at the bootstrap time, N instances of *Process*, one per each process ID. The notation $Process(1)$, $Process(tp)$, and so forth, permits one, during model checking, to refer to a particular *Process* instance.

As a final remark, it is worth noting that in the graphical form of a model, like Figure 1, a guard is green-colored, a synchronization is drawn in azure, and an update is shown in blue. When a non-deterministic selection of the value of a variable is also specified in the guarded command, it is depicted in yellow (see also Figure 4).

3.2. Verification of Burns and Lamport's Algorithm

The BL algorithm was first analyzed using the symbolic simulator, by observing, step by step, particular event sequences where, at each step, it is the modeler, which explicitly selects the process, which will execute the next action. During this monitoring, the global shared variables were watched too, and the overall behavior was traced. In addition, the random evolution, where it is the simulator that randomly chooses the next event to occur, intuitively confirmed that all the processes can reach their CS, no deadlock happens, and in no case can two or more processes enter their CS simultaneously. Figure 2 shows a snapshot for the case $N = 2$, where process 1 is entering its CS and the other is in the BW1 busy-waiting location.

Accurate property assessment rests on the exhaustive verification ensured by model checking. Table 1 collects the TCTL queries [18] used for checking the properties of BL (see also Section 2.2), their meaning, and the achieved result. It is important to note that exactly the same results emerged when using NCS as a normal location (default) and when NCS is changed to urgent. The state predicate $Process(i).CS$ evaluates to *true* (numerically 1) if process i is found in its CS, or $false$ otherwise (numerically 0).

Table 1. TCTL queries for property assessment of BL.

#	Query	Meaning	Result
1	A[] (sum(i:pid)Process(i).CS) ≤ 1	Is it always true, i.e., in all the states of the state graph, that the number of processes simultaneously found in CS is less than or equal to 1?	satisfied
2	A[] !deadlock	Are all the states of the state graph deadlock free?	satisfied
3	For any process j, 1 $\leq$ j $\leq$ N: E<> Process(j).CS	Does the process j eventually enters its CS?	satisfied
4	sup{ Process(tp).E}: z[tp]	What is the maximal value of z[tp] when process tp is found in the location E of Figure 1?	N $-$ 1 if tp == 1; unbounded if tp! = 1
5	E<> Process(tp).NCS && exists(j:pid)Process(j).CS	Is there any state where the tp process in its NCS and other processes can be in the CS?	satisfied

Table 1 confirms that the BL algorithm satisfies all the mutual exclusion properties when the target process is 1. The fourth sup (suprema) query, in particular, checks the overtaking bound, that is, the maximum time the competing target process tp has to wait before it gets permission to enter its CS. For $N = 4$, it results in $ov = N - 1 = 3$. By changing the target process to $tp = 2$ (or higher), all the mutual exclusion properties continue to be

satisfied, as shown in Figure 3, except that the overtaking factor now becomes unbounded. The overtaking factor was also investigated by the query

A[] Process(tp).E imply z[tp] <= *finite_bound*

that for $tp! = 1$, which always terminates as unsatisfied, whatever is the value of *finite_bound*, e.g., 1000 or greater, to mirror the fact that $z[tp]$ is unbounded.

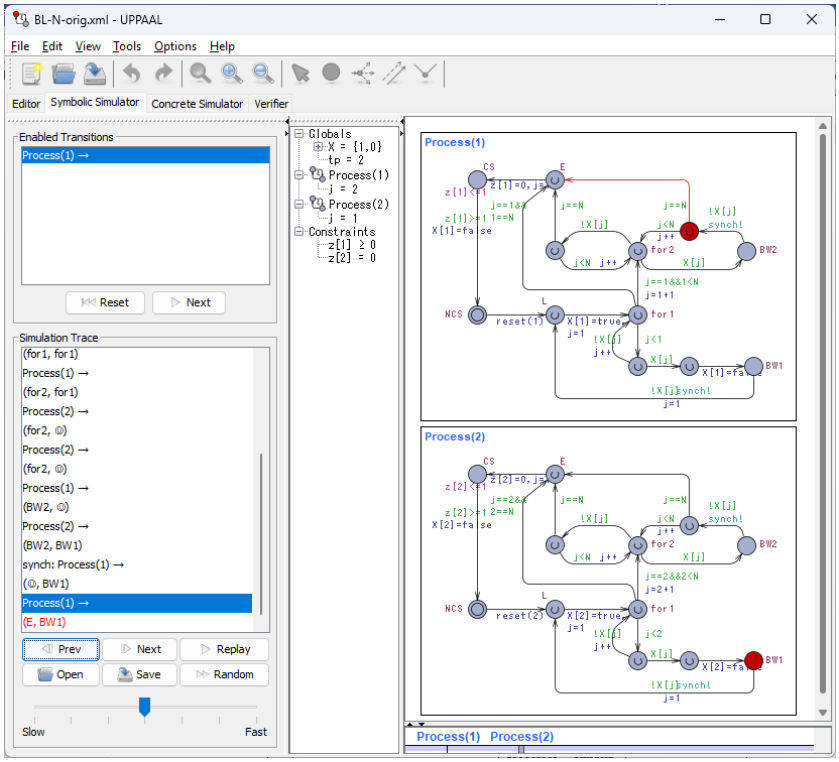

Figure 2. A snapshot of the BL model of Figure 1 for N = 2 processes.

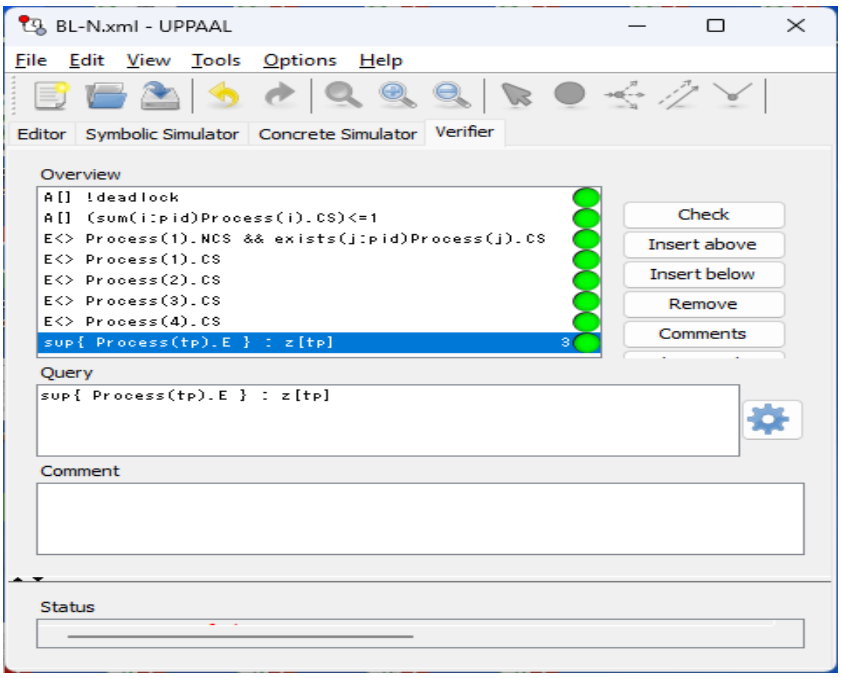

Figure 3. Property checking of BL when $N = 4$ and $tp = 1$.

3.3. The BL Model under Non-Atomic Memory

The BL model in Figure 1 assumes, by default, the atomic memory model, where read/write operations on the same memory cell are indivisible. Figure 4 shows the same model adapted for working with the non-atomic memory model. Since the use of only *exterior* variables [29], that is, the fact that any shared variable $X[j]$, is only written by the process j but consulted by all the other processes, as in [16], only the *flickering* phenomena has to be considered. Flickering arises when multiple read operations occur during a write operation on the same variable. Due to flickering, a read concurrent to a write is supposed to return a non-deterministic value belonging to the type of variable. Flickering can be modeled in Uppaal by a *non-deterministic selection* of a value that is temporarily assigned to the variable. Flickering is then followed by the effective value assigned to the variable. For non-determinism, a reader process can gain the flickered value instead of the true value. Of course, flickering augments the non-determinism degree in the model behavior, which in turn makes the model checking activities more expensive.

The extensive verification of the BL model in Figure 4 confirmed that it is RW-safe [16] and possesses the same properties of the BL model shown in Figure 1. This includes the unbounded overtaking factor for any process ID different from 1.

The documented BL model checking work complies with the same properties and results predicted in [1] (part II). The carried work was intended as a preliminary step to motivate and illustrate the practical use of the Uppaal-based approach. In addition, the experience has furnished a clear argument about its semantic correctness.

The next step will be devoted to formal modeling and correctness verification of the more complex solution by Lycklama and Hadzilacos [24] that embodies the BL algorithm as one of its components.

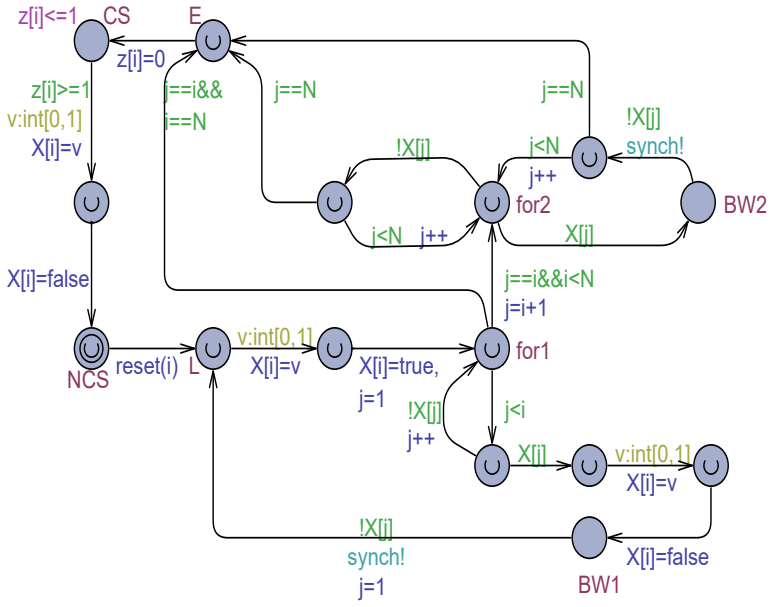

Figure 4. The BL model adapted to work with non-atomic memory.

4. Modelling and Verification of Lycklama and Hadzilacos's Solution

Lycklama and Hadzilacos's (LH) mutual exclusion algorithm was proposed in [24] as a solution that tries to ensure a First-Come-First-Served (FCFS) order to compering processes for entering their critical section. The algorithm is characterized by its small number ($5N$) of bits in the shared data space.

The LH algorithm, adapted from [27], is reported in Algorithm 3 and consists of an outer part and an inner part. The inner part coincides with the BL algorithm. The outer part is devoted to FCFS behavior. In terms of Uppaal modeling, the shared variables of LH can be represented as follows, with the process IDs which range from 1 to N:

```
const int N =. . .;
typedef int[1,N] pid;
typedef int[0,k] value; //k = 3 in the original proposal in [24]
typedef int[0,1] bit;
//shared communication variables
bit X[pid], V[pid], D[pid]; //all initialized to 0
value T[pid] //initialized to 0
```

Each process introduces the local variables $S[]$ of the same type as the global $T[]$, and j, which is used as the control variable of the for-loops.

Instructions from 1 to 5 define the so-called *doorway* [1,27], which is a sequence of actions executed by a competing process before entering busy waiting. The for-loop at line 6 serves to resolve the FCFS order of the arrival processes. Instructions from 7 to 9 are easily recognized as the BL algorithm studied in the previous section of this study. $D[i]$ is used to signal the start point ($D[i] = 1$) and the endpoint ($D[i] = 0$) of the doorway. $V[i] = 1$ mirrors that the process i has started engaging in the FCFS strategy toward the entering of its CS. After the CS, $V[i]$ is reset to 0. $X[i]$ is used according to the BL algorithm. The role played by the array $T[]$ is subtle and fundamental.

When a process starts its doorway, it first copies the global $T[]$ onto the local $S[]$. Then, $T[i]$ is advanced to its next value. In [24], it was conjectured that four values (two bits) for any $T[i]$ are sufficient for the proper management of FCFS. A possible sequence for the values of $T[i]$ is then (as in [24]) $00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00$, that is $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$. In [27], it was observed that it is not the exact sequence of the values of $T[i]$ that matters but rather the number of possible values available for $T[i]$.

Algorithm 3. The Lycklama and Hadzilacos's FCFS mutual exclusion algorithm.

```
Process (i):
local variables: value S[pid]; pid j;
repeat
        NCS;
1.      D[i] = 1; //doorway entry
2.      for (j from 1 to N) S[j] = T[j]; //copy of T onto the local S
3.      T[i] = next(T[i]); //next value available for T[i]
4.      V[i] = 1;
5.      D[i]= 0; //doorway exit
6.      for (j from 1 to N) await-until (D[j] == 0 and (V[j] == 0 or S[j]! = T[j]));
7.      X[i] = 1; //start of BL algorithm
8.      for (j from 1 to i − 1){
                if (X[j] == 1){
                        X[i] = 0; await-until (X[j] == 0); goto 7;
                }
        }
9.      for (j from i+1 to N) await-until( X[j] == 0 ); //end of BL algorithm
        CS;
10.     X[i] = 0;
11.     V[i] = 0;
forever
```

After having updated $T[i]$, the process i declares its FCFS engagement by assigning 1 to $V[i]$ and by announcing it exited its doorway ($D[i] = 0$). Then, the for-loop at line 6 of Algorithm 3 "resolves" the FCFS order. Process i will wait until there are processes j that have exited their doorway ($D[j] = 0$) *and* either they are out of the CS or not interested in the CS ($V[j] = 0$), or the value of the local $S[j]$ differs from the corresponding global $T[j]$. A process j that enters its CS before i, at its exit from the CS, can immediately re-enter its doorway (by a 0 stay time in the NCS). Then, such a process j should now not be ahead again of i, which would destroy the FCFS property. Toward this, j copies the global $T[]$ into its $S[]$ and advances its $T[j]$ value. Now, if process i is still in the for-loop of line 6, surely

the new competition of $T[j]$ will be forced to wait. In fact, for the process j, the values of $S[i]$ and $T[i]$ do not differ. In addition, for process i to prosecute before the new arrival of process j, it is mandatory that the new value of $T[j]$ still differs from the value that process i has in its $S[j]$. The key factor is then the length of the distinct values of $T[j]$. A short length can imply a deadlock situation, because neither process i nor the new activation of j could observe the case $S[j]! = T[j]$.

A possibility offered by the Uppaal modeling approach is the easy check of the minimal number of values for a $T[j]$, that is, the minimal shared data for the processes that would guarantee the correct behavior for the LH algorithm (see later in this study).

Figure 5 reports the Uppaal model of the LH algorithm in Algorithm 3. As one can see from Figure 5, since it is not possible to read and write the $T[i]$ variable in the same action, the existing value of $T[i]$ is first saved in the (at the moment free) j variable. To improve the readability of the model, locations were mostly named according to the instruction line numbers of Algorithm 3. Functions $next(j)$ and $try()$ are detailed in Algorithm 4.

As a preliminary investigation, the four values of the $T[]$ variables of [24] were assumed, and the behavior of the model for $N = 3$ was observed both in the symbolic simulator and then through the Uppaal model checker. Effectively, the model was found to be correct (see also the snapshot in Figure 6) according to all the mutual exclusion properties. In addition, the overtaking bound was found to be $N - 1$, whatever the chosen target process. The results in Figure 6 emerged when using NCS either as a normal location or as an urgent location.

Algorithm 4. The next(j) and try() functions used in the model of Figure 5.

```
bool try(){
    return D[j] == 0 && (V[j] == 0 | |S[j]! = T[j]);
}//try

value next(const value j){
    return j == 0?1:j == 1?3:j == 3?2:0;
}//next
```

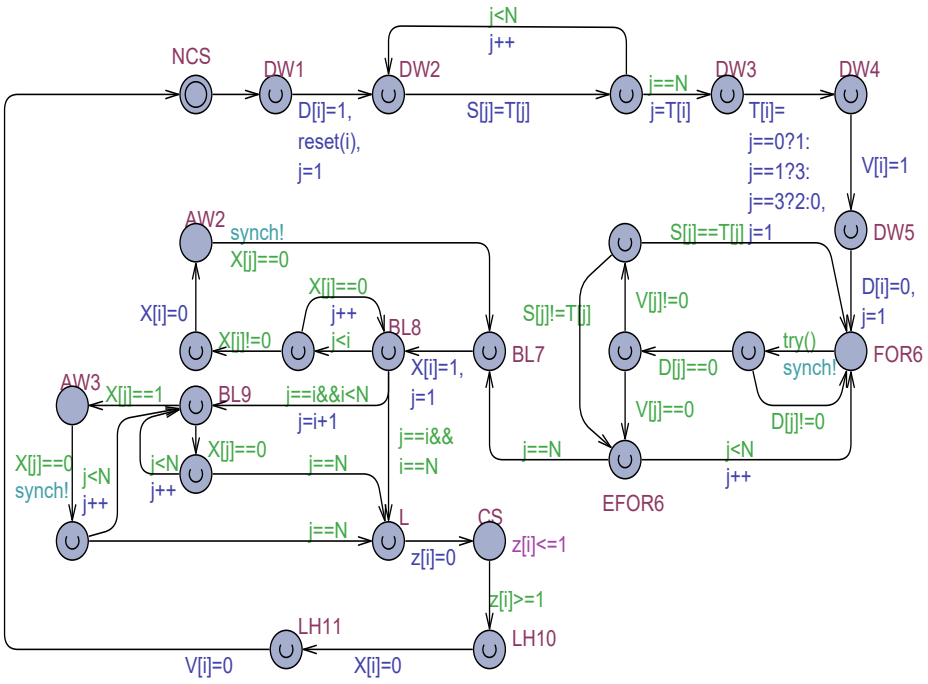

Figure 5. A Uppaal model for the LH solution in Algorithm 2.

A further liveness property, which logically precedes the prediction of the overtaking bound, was assessed by the following query, which uses the *leads-to* operator [18] (see Figure 6):

Process(1).DW1 --> Process(1).CS

The query asks if (invariantly) starting from a state where process 1 is at the beginning of its doorway (the location DW1), it always (inevitably) happens that a state can be reached where process 1 enters its CS. The property is satisfied and confirms the absence of starvation, which is a precondition for a bounded overtaking. The *sup* query, finally, was used to infer the exact value of the overtaking bound.

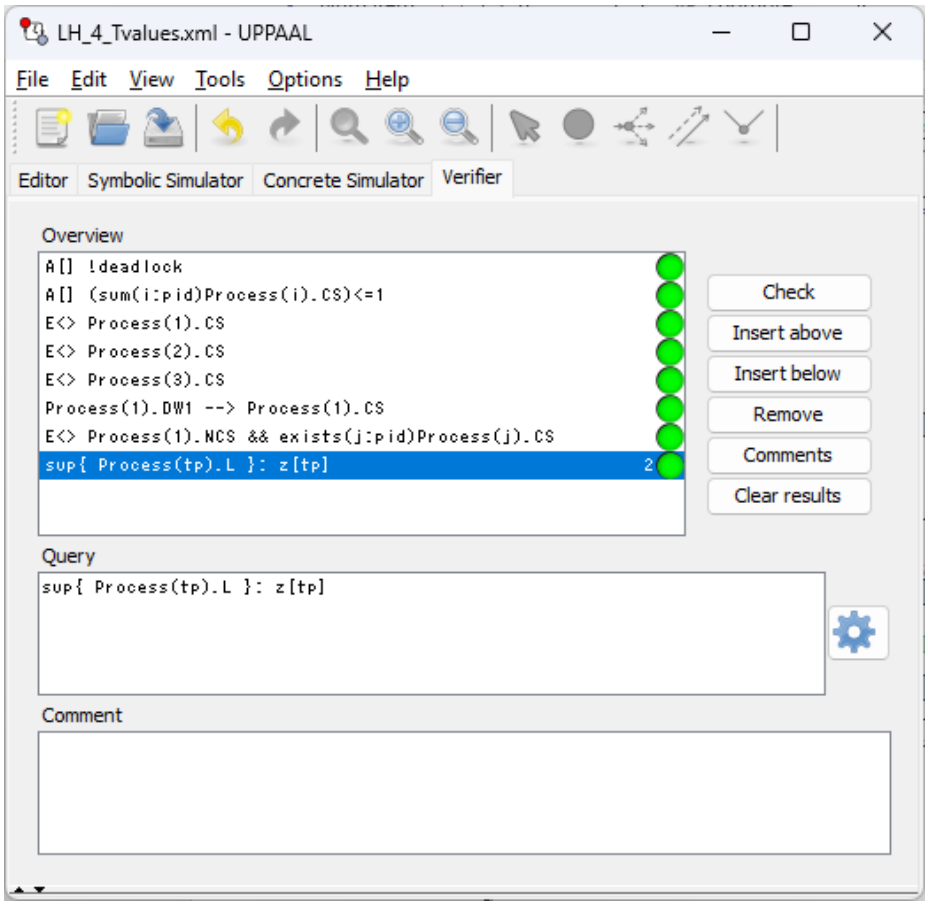

Figure 6. A model checking snapshot of the LH model of Figure 5 when N = 3 processes and 4 values for $T[i]$ are used.

The analysis of the LH model was extended by checking the minimal number of values for $T[i]$, which can guarantee the algorithm continues to be correct in all of its properties. Three values were then assumed: from 0 to 2, with a straightforward sequence $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ enforced by a simple adaptation of the $next()$ function in Algorithm 4. Despite these three values, the model remains fully correct and satisfies all its properties. Table 2 collects the overtaking factor, for N, which ranges from 2 to 4 (with $N = 5$ the state graph explodes), which is confirmed to be $N - 1$. For completeness, the elapsed time (ET), in s, required by the *sup* query to terminate, and the corresponding memory usage (MU), in MB, are reported too.

Table 2. Overtaking bound (ov) vs. N for LH with 3 values for $T[]$ variables.

N	ov	ET (s)	MU (MB)
2	1	0	18
3	2	0.6	38
4	3	179.4	3300

Fewer than three values for the $T[]$ variables proved to be insufficient. With two values (1 bit) and $N >= 3$, the LH model incurs into deadlocks. Two values are instead ok for the simple case of $N = 2$ processes.

The following gives some further arguments about the FCFS order in LH. As observed by Aravind in [27], in reality, the FCFS property is the effective order of entering the CS only when the competing processes execute their doorways sequentially. In the more common case of a concurrent execution of the doorways, the processes do not necessarily enter their CS in the FCFS order. This property was observed on the LH model animated in the symbolic simulator. Despite this behavior, LH has the merit of having improved the underlying BL algorithm by ensuring that the overall model now has a linear, bounded overtaking factor.

All the execution experiments were performed on a Win11 Pro desktop platform, Dell (Round Rock, TX, USA) XPS 8940, Intel i7-10700 (8 physical cores), CPU@2.90 GHz, and 32 GB RAM, using version 5 of Uppaal 64 bit.

Adapting LH's Algorithm to Non-Atomic Memory

LH's solution was also analyzed in the presence of flickering, that is, when multiple read operations can occur simultaneously to a write operation on a shared variable. The new Uppaal model, with three values admitted for the $T[]$ variables, is shown in Figure 7.

Unfortunately, the model in Figure 7 was found to be *not* deadlock-free. The reason is because when a process copies the global values of $T[]$, it can copy flickered values, which can block it in the FOR6 location. This same result was found when using the four values for $T[]$ assumed in [24]. In the light of the carried-out model checking work, LH's solution emerged to be a fully correct mutual exclusion algorithm, but only under atomic memory.

A final solution was then devised and investigated by using LH's algorithm for two processes and with two values for the T[] variables, as the arbitration algorithm is in a state-of-the-art tournament binary tree organization [23,28]. Algorithm 5 shows a stripped-down version of LH for two processes (LH2), which relies on the following Uppaal global declarations:

```
const int N = 2;
typedef int[1,N] pid;
typedef int[0,1] bit;
//shared communication variables
bit X[pid], V[pid], D[pid], T[pid]; //all initialized to 0 (default)
```

with the $next()$ function, which reduces to:

```
bit next(const bit j){return j == 0?1:0;}//next
```

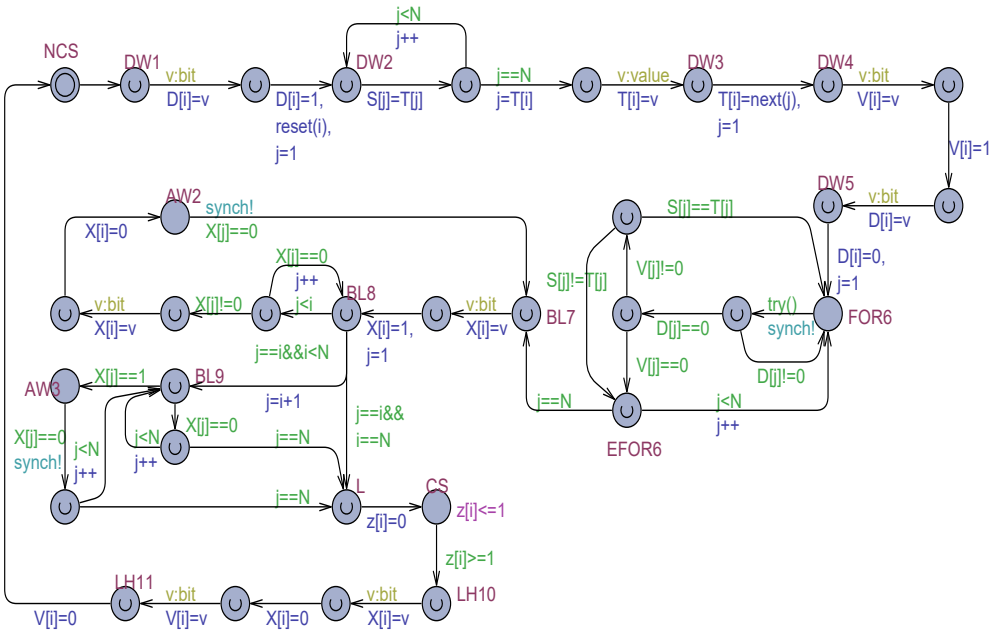

Figure 7. The LH model adapted for working with non-atomic memory.

In Algorithm 5, for-loops are flattened to the only existing iteration. Constant *j* denotes the partner process.

Algorithm 5. LH2: the LH's algorithm tailored to 2 processes and 2 values for $T[]$.

```
Process(i):
local variables: bit S[pid]; const int j = 3 − i;
repeat
        NCS;
  1.    S[i] = T[i];
  2.    S[j] = T[j];
  3     T[i] = next (T[i]);
  4     V[i] = 1;
  5     D[i] = 0;
  6     await-until (D[j] == 0 and (V[j] == 0 or S[j]! = T[j]));
  7     X[i] = 1;
  8     if (j < i){
                    if (X[j]! = 0){X[i] = 0; await-until (X[j] == 0); X[i] = 1;}
        }
        else if (X[j] == 1) await-until (X[j] == 0);
        CS;
  9     X[i] = 0;
  10    V[i] = 0;
forever
```

5. Embedding LH in a Tournament Tree Standard Solution

The tournament binary tree (TT) advocated, e.g., in [28], and implemented in [23] was tailored to work with LH for two processes (see Algorithm 5). LH2 is used to arbitrate pairs of processes at the intermediate nodes of the TT, including the root. The overall solution can handle $N \geq 2$ processes.

Processes enter the TT at the leaf nodes of the last, possibly incomplete, level of the tree. After arbitration, the winner process moves to its ancestor node where it will engage the next arbitration and so forth. The process that reaches the root node is the one that has permission to enter its critical section. The concrete implementation of the TT maps

the tree, level by level, onto a linear array, as usually happens when supporting the heap sorting algorithm. Slot 1 of the array denotes the root. The j variable in Figure 8 starts with the index of the leaf node, which becomes occupied by the arriving process i. Then, j is halved at each iteration where process i is the winner of an arbitration until it becomes 1.

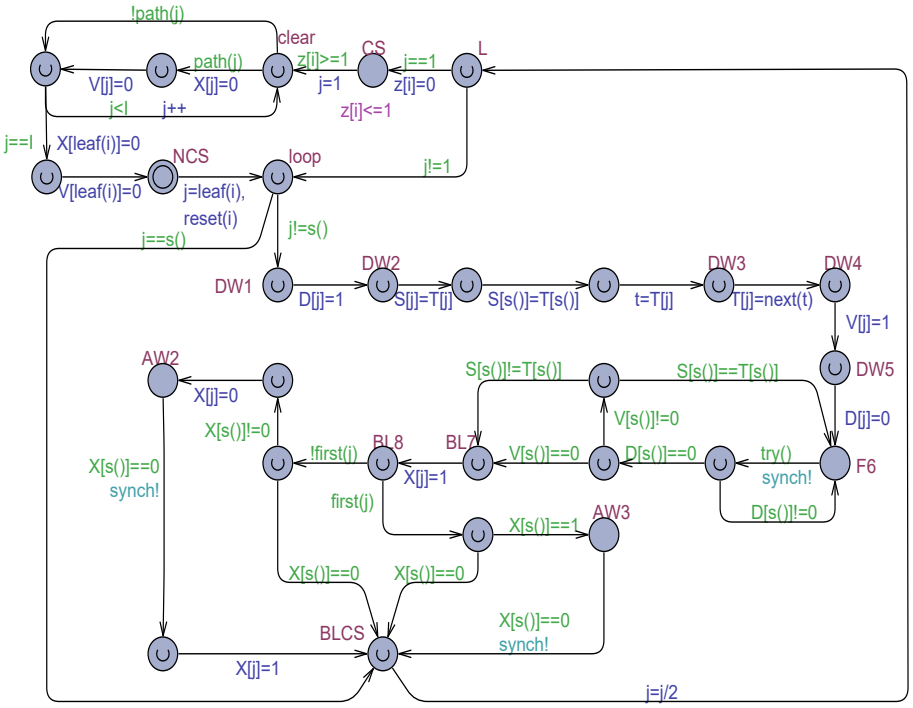

Figure 8. TT-LH2: A Uppaal model for the TT's solution based on LH2.

During its upward movement along a path toward the root, process i becomes represented by the index j of the slot occupied in the array. The two processes of arbitration are siblings of the same ancestor node. The sibling, if there is one, of the process in j is returned by the $s()$ function. Therefore, the typical arbitration pair of processes is composed of the processes $<j, s()>$. The function $first(j)$ returns *true* if $j < s()$. To better characterize the TT organization, it is convenient to illustrate some Uppaal global declarations.

```
const int N = 5; //example
const int lev = fint(ceil(log2(N))); //last level of TT
const int pow2toLev = fint(pow(2,lev)); //2^lev
const int L = pow2toLev + N − 1; //number of nodes of TT
const int I = pow2toLev − 1; //number of intermediate nodes
typedef int[1,N] pid; //process indexes
typedef int[1,L] findex; //TT node indexes
typedef int[0,1] bit;
//shared communication variables
bit X[findex], V[findex], D[findex], T[findex]; //all 0 initially
```

Leaf nodes are deterministically associated to the N processes through the $leaf()$ function:

```
findex leaf(const pid i){ //returns the leaf node index for the process i
return pow2toLev + i − 1;
}//leaf
```

The winner of a local arbitration based on LH2 moves to BL's critical section (BLCS), and then it is ready to repeat the main loop by first halving the j variable ($j = j/2$). BLCS must not be confused with the overall critical section (CS) of the TT solution. In the implementation, a process in BLCS immediately moves to its next arbitration. The unitary

duration is only spent in the CS location of Figure 8. When the sibling process does not exist, due to a partially filled last level of the tree, process j moves directly to the BLCS. Otherwise, the BL algorithm is played to determine the winner process.

In the developed TT, each of the shared variables is associated with every distinct node of the tree, which can be occupied by a process. Practically, the shared variables *represent* the tree structure. The resultant size of the shared space is $4(N + 2^{\lceil \log_2 N \rceil} - 1)$ bits, which becomes $4(2N - 1)$ when N is a power of two.

Purposely, the reset of $X[j]$ and $V[j]$ is delayed until the process j reaches the root node and enters its CS. Upon exiting from the CS, all the path positions previously occupied during the upward movement are cleared, exactly in the opposite direction. It is at this time that the $X[j]$ and $V[j]$ shared variables are reset to 0. These operations can awaken blocked processes at arbitration nodes, which can resume their advancement.

The algorithm/model in Figure 8 satisfies all the mutual exclusion properties. The overtaking bound ov vs. N, by using NCS urgent or not, emerged, as shown in Table 3. The table also reports the elapsed time (ET) (in s) and the memory usage (MU) in MB of the model checker. For $N = 6$, the state graph of the TT-LH2 model explodes.

Table 3. The overtaking bound (ov) for the TT-LH2 under atomic memory.

N	ov	ET (s)	MU (MB)
2	1	0	18
3	3	0.2	26
4	3	5.5	153
5	7	671	8329

Table 3 confirms ov is bounded and varies according to $ov = 2^{\lceil \log_2 N \rceil} - 1$, that is, the numerosity of the intermediate nodes in the TT. When the last level is completely occupied (N is a power of 2), the relation becomes $ov = N - 1$.

The final challenging point was to check TT-LH2's algorithm under flickering and non-atomic memory.

Adapting TT-LH2 to Non-Atomic Memory Model

An important point of using LH is that all the shared variables can be affected by flickering only due to multiple read operations, which can occur during a write operation. In other terms, no *scrambling* can occur (which should be fenced by lower-level mechanisms [28]) on the value of a shared variable that could be affected by simultaneous write operations. The TT-LH2 model of Figure 8, modified by introducing flickering, is shown in Figure 9.

Model checking the model in Figure 9 confirmed all the mutual exclusion properties are satisfied exactly as for the model with atomic memory in Figure 8, although the new model is more difficult to analyze due to the increased degree of non-determinism caused by the flickering operations. Also, the overtaking bound was confirmed to be identical to that of the model in Figure 8 (see Table 4). Now, for $N = 5$, the state graph explodes.

Table 4. The overtaking bound (ov) of the TT-LH2 with flickering.

N	ov	ET (s)	MU (MB)
2	1	0	18
3	3	0.8	26
4	3	34	611

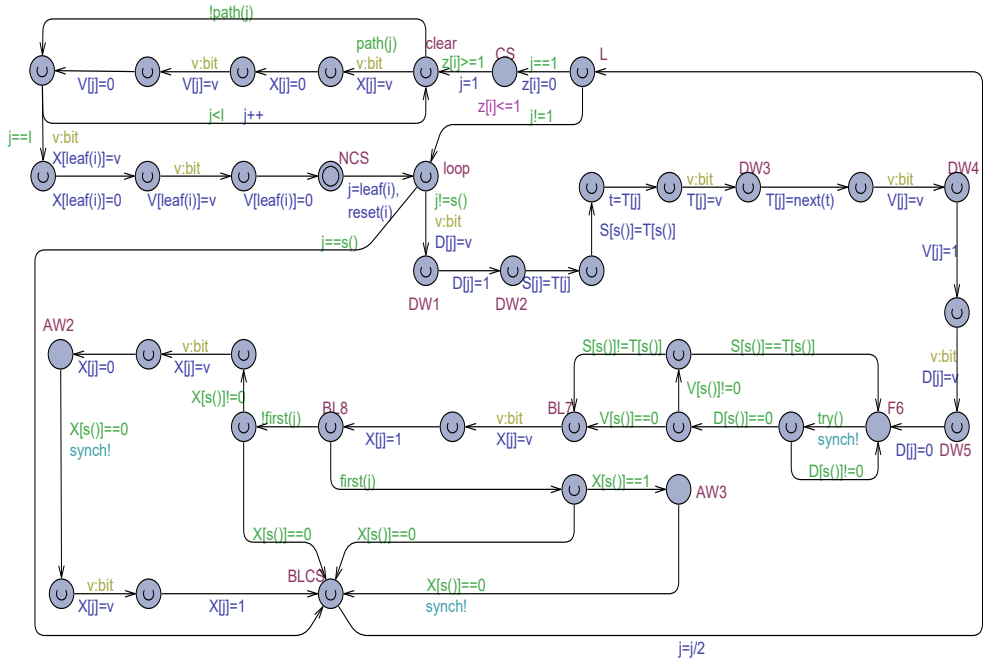

Figure 9. The TT-LH2 model adapted for working with non-atomic memory.

6. Conclusions

This study describes a modeling and verification approach and applies it for an in-depth analysis of Lycklama and Hadzilacos's (LH) [16,24,27] algorithm. The methodology confirmed how LH, together with its support for an FCFS order of the processes entering their critical section, improves Burns and Lamport's (BL) solution, embodied in LH, by eliminating the unbounded overtaking, of which BL, in general, suffers. However, as this study demonstrates, LH itself manifests deadlock problems when used in the context of non-atomic memory. Therefore, a new solution based on LH streamlined to two processes (LH2) is proposed, where LH2 is used as the arbitration unit in a tournament tree (TT) state-of-the-art general and standard solution [28]. TT-LH2 emerged to be a fully correct mutual exclusion algorithm for $N >= 2$ processes, with a guaranteed bounded, linear overtaking factor.

The power and flexibility of the adopted methodology for reasoning on concurrency derive from the use of formal modelling and exhaustive model checking ensured by the timed automata language and the tools of Uppaal [18].

Prosecution of the research work will address the following points. First, apply the developed approach to check the correctness of all the variants of LH's algorithm proposed in [16,27]. Second, port the approach on a cluster of many cores and high memory size to permit the verification of complex models. Third, experiment with a simulation framework in Java based on the Theatre actor system [30] to study mutual exclusion algorithms scalable in the number N of the involved processes.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Lamport, L. The mutual exclusion problem: Part I—Theory of interprocess communication, and part II—Statement and solutions. *J. ACM* **1986**, *33*, 313–348. [CrossRef]
2. Raynal, M. *Algorithms for Mutual Exclusion Problem*; The MIT Press: Cambridge, MA, USA, 1986.
3. Raynal, M. *Concurrent Programming: Algorithms, Principles, and Foundations*; Springer-Verlag: Berlin/Heidelberg, Germany, 2013.

4. Herlihy, M.; Shavit, N. *The Art of Multiprocessor Programming*; Elsevier: Amsterdam, The Netherlands; Morgan Kaufmann: Cambridge, MA, USA, 2012.
5. 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.
6. Dekker, T.J. History of Dekker's Algorithm for Mutual Exclusion. In *Tales of Electrologica: Computers, Software and People*; Alberts, G., Groote, J.F., Eds.; Springer Nature: Berlin/Heidelberg, Germany, 2022; Chapter 6; pp. 111–120.
7. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Commun. ACM* **1965**, *8*, 569. [CrossRef]
8. Knuth, D.E. Additional comments on a problem in concurrent programming control. *Commun. ACM* **1966**, *9*, 321–322. [CrossRef]
9. de Bruijn, N.G. Additional comments on a problem in concurrent programming control. *Commun. ACM* **1967**, *10*, 137–138. [CrossRef]
10. Eisenberg, M.A.; McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. *Commun. ACM* **1972**, *15*, 999. [CrossRef]
11. Peterson, G.L. Myths about the mutual exclusion problem. *Inf. Process. Lett.* **1981**, *12*, 115–116. [CrossRef]
12. Block, K.; Woo, T.K. A more efficient generalization of Peterson's mutual exclusion algorithm. *Inf. Process.Lett.* **1990**, *35*, 219–222. [CrossRef]
13. Nigro, L.; Cicirelli, F. Correctness Verification of Mutual Exclusion Algorithms by Model Checking. *Modelling* **2024**, *5*, 694–719. [CrossRef]
14. Buhr, P.A.; Dice, D.; Hesselink, W.H. Dekker's mutual exclusion algorithm made RW-safe. *Concurr. Comput. Pract. Exp.* **2016**, *28*, 144–165. [CrossRef]
15. Wang, Z.; Zuo, Q.; Li, J. An intelligent multi-port memory. In Proceedings of the 2008 International Symposium on Intelligent Information Technology Application Workshops, Shanghai, China, 21–22 December 2008; IEEE Computer Society: Los Alamitos, CA, USA, 2008; pp. 251–254.
16. Hesselink, W.H. Verifying a simplification of mutual exclusion by Lycklama–Hadzilacos. *Acta Inform.* **2013**, *50*, 199–228. [CrossRef]
17. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
18. 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.
19. Cicirelli, F.; Furfaro, A.; Nigro, L. Model checking time-dependent system specifications using time stream Petri nets and Uppaal. *Appl. Math. Comput.* **2012**, *218*, 8160–8186. [CrossRef]
20. 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]
21. Atif, M. Formal Modeling and Verification of Distributed Failure Detectors. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2011. ISBN 978-90-386-2620-8.
22. Alur, R.; Dill, D.L. A theory of timed automata. *Theor. Comput. Sci.* **1994**, *126*, 183–235. [CrossRef]
23. Nigro, L.; Cicirelli, F.; Pupo, F. Modeling and Analysis of Dekker-Based Mutual Exclusion Algorithms. *Computers* **2024**, *13*, 133. [CrossRef]
24. Lycklama, E.A.; Hadzilacos, V. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Trans. Program. Lang. Syst.* **1991**, *13*, 558–576. [CrossRef]
25. Burns, J.E. Complexity of Communication among Asynchronous Parallel Processes. Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, 1981.
26. Burns, J.E.; Lynch, N.A. Bounds on shared memory for mutual exclusion. *Inf. Comput.* **1993**, *107*, 171–184. [CrossRef]
27. Aravind, A.A. Simple, space-efficient, and fairness improved FCFS mutual exclusion algorithms. *J. Parallel Distrib. Comput.* **2013**, *73*, 1029–1038. [CrossRef]
28. Hesselink, W.H. Tournaments for mutual exclusion: Verification and concurrent complexity. *Form. Asp. Comput.* **2017**, *29*, 833–852. [CrossRef]
29. Kessels, J.L.W. Arbitration without common modifiable variables. *Acta Inform.* **1982**, *17*, 135–141. [CrossRef]
30. Nigro, L. Parallel Theatre: An actor framework in Java for high performance computing. *Simul. Model. Pract. Theory* **2021**, *106*, 102189. [CrossRef]