Trylock, a case for temporal logic and eternity variables

Wim H. Hesselink

December 29, 2021

Abstract

An example is given of a software algorithm that implements its specification in linear time temporal logic (LTL), but not in branching time temporal logic (CTL). In LTL, a prophecy of future behaviour is needed to prove the simulation. Eternity variables are used for this purpose. The final phase of the proof is a refinement mapping in which two threads exchange roles.

The example is a software implementation of trylock in a variation of the fast mutual exclusion algorithm of Lamport (1987). It has been used fruitfully for the construction of software algorithms for high performance mutual exclusion.

Keywords: concurrency, locking, CTL, LTL, prophecy variables, simulation.

1 Introduction

When Dijkstra [4] proposed the mutual exclusion problem in 1965, he more or less apologized for its academic character. With the advent of multithreading and multicore computing in the last years, however, the importance of mutual exclusion or locking is no longer in doubt.

The problem of locking has two sides: granting exclusive access to one thread, and letting the other interested threads wait. This asks for a separation of concerns. One can separate the subproblem of granting exclusive access. This means that a thread asks for exclusive access and either gets it, or receives the answer no. This variation is known as trylock. It is described here as the system Try.

The system Try can easily be implemented in hardware. Here, a software solution TryL is proposed with shared variables and atomic read and write operations, based on Lamport's fast mutual exclusion algorithm [13]. In branching time temporal logic, CTL, however, TryL does not implement system Try. The implementation is valid only in linear time temporal logic, LTL. The fact that CTL rejects a suitable implementation of system Try, while LTL accepts it, provides an example where LTL is to be preferred over CTL in concurrency verification. This is in line with the arguments of Vardi and Nain [19, 17].

The LTL proof of validity of the implementation is based on simulation, in several phases. The first phase is a standard analysis of the invariants of the algorithm. The second phase consists of proofs of two progress properties of the algorithm. This is done with UNITY logic [2, 16]. Note that UNITY logic can be expressed in LTL. In some sense [11], it is even equivalent to LTL.

The third phase deals with the problem that the construction of a simulation requires a prophecy of future behavior. Abadi and Lamport [1] have introduced prophecy variables for this purpose. Prophecy variables are based on König's Lemma, have restricted applicability, and induce rather complicated proof obligations. We therefore prefer to use an eternity variable [6, 8, 9]. Eternity variables have been applied earlier, e.g., in [5, 7], but the present application is more illustrative.

The construction of a prophecy by means of an eternity variable is not enough, because it turns out that the crucial command is often executed by the wrong thread. The fourth phase consists of a refinement mapping with a subtle pas de deux of two threads to correct the mismatch.

The proof has been verified with the proof assistant PVS [18]. The proof script is available at [10]. For the sake of readability, there are minor differences between the presentation here and the proof script.

System TryL has been used previously in the construction of the Triangle Algorithm [12], a mutual exclusion algorithm that is very efficient both under high contention and under low contention.

This case study combines and illustrates several aspects of the treatment of shared-variable fine-grain concurrency: temporal logic, simulation, theorem proving, and UNITY.

Overview

Specification Try is presented in Section 2. In Section 3, the implementation TryL of Try is given. In Section 4, it is shown that TryL does not implement system Try in branching time temporal logic. This section also introduces the linear-time theory of specifications and simulations that is needed. Section 5 proves that TryL implements Try in LTL, in the four phases described above. Conclusions are drawn in Section 6.

Acknowledgments. The fruitful collaboration with Peter Buhr and Dave Dice on efficiently implementable mutual exclusion algorithms is gratefully acknowledged. The present paper is a theoretical offshoot of this work. The draft version of this paper has been greatly improved due to suggestions of the anonymous reviewers.

2 The system Try

In a shared-memory system with a finite number of threads, system Try offers functions *trylock* and *unlock*; a call of *trylock*(p) tries to obtain exclusive access for thread p, it does not wait, and returns a boolean to indicate its success. The second function *unlock* releases the lock. The typical application is that every thread is in the loop

```
loop of  $thread\ p$ :
    $NCS$ ;
    if  $trylock(p)$  then
        $CS$ ;
        $unlock(p)$
    endif
endloop .
```

Here *NCS* and *CS* are program fragments, *CS* stands for the critical section in which never more than one thread is allowed, *NCS* stands for the noncritical section. *CS* is supposed always to terminate, *NCS* need not terminate. *CS* and *NCS* do not refer to variables that are used in the implementation of *trylock* and *unlock*.

We take *thread* as the set of the thread identifiers, a finite subset of the integers. The system can be specified by means of a shared specification variable:

```
shared variable
    $\mathtt{mu} : thread_\perp = \perp$ ;
```

```
$trylock(p : thread)$ : boolean =
    $\langle$ **if** $\mathtt{mu} = \bot$ **then** $\mathtt{mu} := p$ ; **return** $true$ **else return** $false$ **endif** $\rangle$ .

$unlock(p : thread) = \langle\ \mathtt{mu} := \bot\ \rangle$ .
```

This functionality is the same as for *tryLock* in Java; similar functions exist for pthreads. The abstract variable $\mathtt{mu}$ is called the *mutex*. If $\mathtt{mu} \neq \bot$, its value is the identifier of the thread with exclusive access. The angular brackets indicate that the actions enclosed are performed atomically, i.e., without interference.

Thread p is only allowed to call *unlock* when it holds the mutex, i.e., when $\mathtt{mu} = p$. When it holds the mutex, it must not call *trylock*, but it must call *unlock* eventually.

The system Try can be implemented practically with a compare-and-swap instruction. This implementation is called TryH, with the letter H for hardware. Below, we give a software implementation TryL of system Try, based on the use of variables with atomic read and write instructions.

3 Implementing system Try, following Lamport

The system TryL, proposed now, implements system Try of Section 2. It is derived from Lamport's fast algorithm [13].

```
shared variables
    $\mathtt{x} : thread$ ;
    $\mathtt{y} : thread_\bot = \bot$ ;
    $\mathtt{bb}[thread] : \text{bool} = (false, \ldots)$ .

$trylockL(p : thread)$ : boolean =
    if $\mathtt{y} \neq \bot$ then return $false$ endif ;
    $\mathtt{bb}[p] := true$ ;
    $\mathtt{x} := p$ ;
    if $\mathtt{y} \neq \bot$ then
        $\mathtt{bb}[p] := false$ ; return $false$ endif ;
    $\mathtt{y} := p$ ;
    if $\mathtt{x} = p$ then return $true$ endif ;
    $\mathtt{bb}[p] := false$ ;
    for each $kk \in thread$ while $\mathtt{y} = p$ do
        await $\mathtt{y} \neq p \ \vee\ \neg\,\mathtt{bb}[kk]$ endfor ;
    return $(\mathtt{y} = p)$ .

$unlockL(p : thread)$ =
    $\mathtt{y} := \bot$ ;
    $\mathtt{bb}[p] := false$ .
```

A proof of correctness is given in Section 5.

The entry function can be explained as follows. If there is no interference, thread p sets $\mathtt{x} := p$, observes $\mathtt{x} = p$, and returns true. In case of contention, the last thread that sets $\mathtt{x} := p$, may observe $\mathtt{y} \neq \bot$, and fail. In that case, the last thread that has set $\mathtt{y} := p$ waits in its **for** loop until all competing threads have failed, and then returns true.

The initial test of $\mathtt{y} \neq \bot$ is necessary to preclude livelock. Indeed, if the initial test $\mathtt{y} \neq \bot$ is removed, there is the following livelock scenario. It starts with $\mathtt{y} = \bot$ and all threads idle. Two threads p_1 and p_2 call *trylock* and proceed to the assignment of `y`. At this point $\mathtt{x} = p_2$, say. Then p_2 does the assignment to `y`, and its call of *trylock* succeeds. It calls *unlock* and resets $\mathtt{y} := \bot$. Then p_1 sets $\mathtt{y} := p_1$. As $\mathtt{x} = p_2 \neq p_1$, thread p_1 enters the **for** loop with $\mathtt{y} = p_1$. Here the cycle starts. Thread p_2 calls *trylockL*. As the initial test is absent, the thread sets $\mathtt{bb}[p_2]$. Then thread p_1 tests $\mathtt{bb}[kk]$ with $kk = p_2$, and remains waiting. Then p_2 observes $\mathtt{y} \neq \bot$, resets $\mathtt{bb}[p_2]$, and becomes idle. This cycle is repeated indefinitely. Thread p_2 modifies $\mathtt{bb}[p_2]$ infinitely often, and thread p_2 cannot proceed because it always finds $\mathtt{bb}[p_2] = \mathit{false}$. As $\mathtt{y} = p_1 \neq \bot$ remains true, every subsequent call of *trylockL* by any thread fails. This is livelock.

Lamport's algorithm [13] is a mutual exclusion algorithm with essentially

```
$\mathit{acquire}(p) \approx$
    while $\neg\, \mathit{trylock}(p)$ do $\mathit{pause}()$ endwhile .
```

In the code of [13], the initial test of $\mathtt{y} \neq \bot$ is absent, and livelock is precluded by placing the test $\mathtt{y} \neq \bot$ at other points.

A second, less important deviation from Lamport's code is the disjunct $\mathtt{y} \neq p$ in the **await** condition. If this disjunct is omitted, a thread can be kept indefinitely in the waiting loop, giving starvation, see [12, Section 3.2]. The condition **while** $\mathtt{y} = p$ is added only for performance. These two modifications are useless in the context of Lamport's algorithm.

If one wants to implement the algorithm on current hardware, one has to reckon with the fact that compilers and hardware optimize programs as if they are sequential. Code for concurrent algorithms therefore needs additional directives to prevent incorrect optimizations. How this is done for the present algorithm is explained in [12, Section 6].

4 Some theory

The question whether TryL implements Try depends on the type of temporal logic one uses. The answer is no in branching time temporal logic, but yes in linear time temporal logic. Branching time is treated in Section 4.1. Section 4.2 introduces linear time. The specification formalism is given in Section 4.3. This section also treats invariants, and the way to find and prove them. Section 4.4 introduces simulation relations between specifications. Finally, eternity variables are introduced in Section 4.5.

4.1 Branching time

In branching time temporal logic, more specifically in CTL [3], the system TryL does not implement system Try. In CTL, computation tree logic, the computation of the algorithm is represented by a tree, rooted in the start state, in which every nondeterministic choice gives rise to branching. In CTL, one might prove that system TryL implements system Try by showing that the systems are (weakly) bisimilar. To show that they are not (weakly) bisimilar, and even that TryL does not implement Try, it suffices to give a CTL formula that holds for TryL and is invalid for Try.

In CTL, $\mathbf{EF}\varphi$ means that there is a directed path in the computation tree from the present node to a node where φ holds. Let there be three threads p_1, p_2, p_3 that each call *trylock* once. The winner is the first thread that obtains the mutex. Let $w(p)$ express that thread p is the winner, and $t(p)$ that the call of thread p has terminated. It is clear that these propositions should be observable. Consider the CTL-formula

$$\Psi : \quad \mathbf{EF}(t(p_1) \ \wedge\ \mathbf{EF}w(p_2) \ \wedge\ \mathbf{EF}w(p_3))$$

Formula Ψ expresses that a path exists from the root to a node where thread p_1 has terminated, and where two paths continue, one towards a node where p_2 is the winner and one towards a node where p_3 is the winner.

$$\begin{tikzcd} & & \bullet\ w(p_3) \\ \underset{root}{\bullet} \arrow[r] & \underset{t(p_1)}{\bullet} \arrow[ur] \arrow[dr] & \\ & & \bullet\ w(p_2) \end{tikzcd}$$

In the systems Try and TryH, formula Ψ is false, because when thread p_1 has terminated, the mutex `mu` has some value $\neq \bot$, so that the winner is determined, although this may not yet be observable.

In system TryL, however, formula Ψ is true. This is shown in the following scenario, with one additional thread p_4. Initially $\mathtt{y} = \bot$. The threads enter. Thread p_1 proceeds to the assignment $\mathtt{x} := p$; the threads p_2, p_3, p_4 proceed to the first assignment $\mathtt{y} := p$. Then p_4 executes $\mathtt{y} := p$. Subsequently p_1 does $\mathtt{x} := p$, establishing $\mathtt{x} = p_1$. It then observes $\mathtt{y} = p_4$, fails and terminates, establishing $t(p_1)$. Now the winner will be p_2 or p_3, dependent on which of the two does the last assignment to `y`. Therefore Ψ holds.

As Ψ is a CTL-formula about the observable tree of the system and is valid for TryL but not for Try, system TryL does not implement Try for branching time temporal logic.

4.2 LTL, linear time temporal logic

In LTL, computations are infinite sequences of consecutive states. Some notation is needed to reason about these sequences. If X is a set (a state space), let X^ω be the set of the infinite seqences xs of elements of X, regarded as functions $\mathbb{N} \to X$. If $xs \in X^\omega$ and $k \in \mathbb{N}$, the kth suffix of xs is defined as the sequence $(xs|k)$ with $(xs|k)(n) = xs(k+n)$. Let $\mathbb{P}(X)$ denote the set of the subsets of X, identified with the predicates on X. For $P \in \mathbb{P}(X)$ and $R \in \mathbb{P}(X^2)$ and $U \in \mathbb{P}(X^\omega)$, one defines

$$\begin{aligned} &[P] = \{xs \in X^\omega \mid xs(0) \in P\}\ , \\ &[R] = \{xs \in X^\omega \mid (xs(0), xs(1)) \in R\}\ , \\ &\Box U = \{xs \in X^\omega \mid \forall\, k : (xs|k) \in U\}\ , \\ &\Diamond U = \{xs \in X^\omega \mid \exists\, k : (xs|k) \in U\}\ . \end{aligned}$$

A state sequence xs satisfies a relation R *under weak fairness* iff it always holds that eventually R is taken by xs or is disabled. This is formally expressed in

$$\mathit{weakFair}(R) = \Box\Diamond([R] \vee [\mathit{dis}(R)])\ ,$$

where $\mathit{dis}(R) = \{x \in X \mid \forall\, y \in X : (x,y) \notin R\}$. The *leads-to* relation of UNITY [2, 16] is defined by

$$(P \mapsto Q) = \Box(\neg[P]\ \vee\ \Diamond[Q])\ .$$

4.3 Computational definitions

The following definitions are more liberal than those of [8] to avoid unnecessary proof obligations. Indeed, the additional requirements of [8] can be regarded as healthiness conditions that are usually satisfied automatically in the applications.

A *specification* K is a tuple (X, X_0, N, P) where X is a set, X_0 is a subset of X, N is a binary relation on X, and P is a subset of X^ω, the set of the infinite sequences over X. The set X is called the *state space*, the elements of X_0 are called *initial states*, relation N is called the *next-state* relation, and P is called the *supplementary property*.

The pairs in relation N are called *steps* of the specification. An *initial execution* of K is an infinite sequence xs of elements of X with $\mathit{xs}(0) \in X_0$, such that every pair of consecutive elements is a step. A *behaviour* of K is an initial execution that also belongs to P. Therefore, the set of behaviours of K is $\mathit{Beh}(K) = [X_0] \cap \Box[N] \cap P$. A predicate $Q \in \mathbb{P}(X)$ is called an *invariant* of K if it contains all states of all behaviours of K.

A subrelation of the next-state relation N is called a *command*. For a command S and predicates P and Q, the *Hoare triple* $\{P\}S\{Q\}$ is the proposition that

$$\begin{array}{l} \forall\, x, y : (x, y) \in S \;\wedge\; x \in P \;\Rightarrow\; y \in Q \text{ , or equivalently} \\ [P \;\Rightarrow\; \mathbf{wp}(S, Q)] \text{ ,} \end{array}$$

where **wp** stands for Dijkstra's weakest precondition.

A predicate P is said to be *preserved* by command S iff $\{P\}S\{P\}$. Predicate P is called *stable* if it is preserved by N. A predicate is called *inductive* iff it is stable and holds initially. Every inductive predicate is an invariant. A predicate implied by an inductive predicate is an invariant; it is called a *forward invariant*.

A predicate P is said to be *threatened* by a command S iff it is not preserved by S. If predicate P is threatened by command S, a predicate Q is called a *remedy* for P and S iff $\{P \wedge Q\}S\{P\}$.

Let $\mathcal{C}$ be a set of commands such that $N = \bigcup \mathcal{C}$. If one has a family of predicates such that any member of the family that is threatened by any command in $\mathcal{C}$, has some remedy consisting of members of the family, then the conjunction of the family is stable. If moreover all members hold initially, the conjunction is inductive, and each member of the family is a forward invariant. This is the method used below to obtain and prove forward invariants.

When one invokes the proof assistant to prove that command S preserves predicate P, it essentially calculates the weakest remedy. In terms of Dijkstra's weakest precondition **wp**, this is $\neg P \vee \mathbf{wp}(S, P)$, because, for every Q,

$$[P \wedge Q \;\Rightarrow\; \mathbf{wp}(S, P)] \quad \equiv \quad [Q \;\Rightarrow\; \neg P \vee \mathbf{wp}(S, P)] \text{ .}$$

Human intelligence, however, is needed to replace the weakest remedy by a remedy that is both manageable and provable.

4.4 Relating specifications

The principal way to relate specifications is by means of (strict) simulation relations, which formalize that the first specification implements the second one.

Let $K = (X, X_0, N, P)$ and $L = (X', X'_0, N', P')$ be specifications. A relation F between X and X' is called a *strict simulation* [8] from K to L if, for every $\mathit{xs} \in \mathit{Beh}(K)$, there exists $\mathit{ys} \in \mathit{Beh}(L)$ with $(\mathit{xs}(n), \mathit{ys}(n)) \in F$ for all n. In this paper, we can avoid the non-strict simulations of [8].

The composition of strict simulations is a strict simulation. Therefore simulations can be constructed in phases. For the phases, two constructions are available, refinement functions and extensions.

Let a function $f : X \to X'$ be called a *refinement function* from K to L iff, for every behavior xs of K, the composition $f \circ \mathit{xs}$ is a behavior of L. If f is a refinement function, its graph $\{(x, f(x)) \mid x \in X\}$ is easily seen to be a strict simulation.

Abadi and Lamport [1] defined $f : X \to X'$ to be a *refinement mapping* from K to L iff it satisfies $f(x) \in X'_0$ for every $x \in X_0$, and $(f(x), f(y)) \in N'$ for every pair $(x, y) \in N$ with $x \in J$ and $y \in J$ for some invariant J of K, and has $f \circ \mathit{xs} \in P'$ for every behaviour xs of K. It is easy to prove that every refinement mapping is a refinement function; the converse implication does not hold.

Let g be a function $X' \to X$. Specification L is called an *extension* of K by g iff
(a) g is a refinement function from L to K,
(b) for every $xs \in Beh(K)$, there is $us \in Beh(L)$ with $xs = g \circ us$.

If L is an extension of K by g, condition (b) implies that the cograph G given by $G = \{(g(u), u) \mid u \in X'\}$ is a strict simulation from K to L. Condition (a) ensures that no information is lost by going from K to L.

4.5 Eternity extensions

In order to prove that TryL implements Try, variables are needed that somehow prophesy future behaviour. Eternity extensions serve to make this possible and sound. They were introduced and treated in [5, 8]. An eternity extension just adds an eternity variable to a specification. This variable gets its constant value initially, by a nondeterministic choice that is guided by the behaviour restriction. The formal definition is as follows.

Let $K = (X, X_0, N, P)$ be a specification, and let M be an arbitrary set. A relation $R \subseteq X \times M$ is called a *behaviour restriction* of K at M iff, for every behaviour xs of K, there exists a value $m \in M$ such that

$$\forall\, n \in \mathbb{N} : (xs(n), m) \in R\ .$$

Let $fst : R \to X$ be the projection function to the first component. Let R be a behaviour restriction of K at M. The corresponding *eternity extension* W is defined as the specification $W = (R, R_0, N', P')$ with $R_0 = \{(x, m) \in R \mid x \in X_0\}$ and

$$((x, m), (x', m')) \in N' \equiv (x, x') \in N \ \wedge\ m = m'\ .$$

and $P' = \{ws \in R^\omega \mid fst \circ ws \in P\}$. It is easy to verify that W is an extension of K by fst, see [8, Thm. 4.1]. Indeed, the definitions imply that every behaviour of W is mapped by fst to a behaviour of K. Conversely, as R is a behaviour restriction, for every behaviour xs of K, there is a value $m \in M$, such that ws given by $ws(n) = (xs(n), m)$ for all n, is a behaviour of W, mapped by fst to xs.

Example. Let K be the specification with two shared integer variables `i` and `n`, both initially 0. There are two threads that under weak fairness concurrently do

```
$p_0$ :   **loop** $\langle$ **if** $\mathtt{i} = 0$ **then** $\mathtt{i} := 1$ **endif** $\rangle$ **endloop** ,
$p_1$ :   **loop** $\langle$ **if** $\mathtt{i} = 0$ **then** $\mathtt{n} := \mathtt{n} + 1$ **endif** $\rangle$ **endloop** .
```

In this system, eventually, p_0 does a step. After this step, both threads are disabled. Therefore, eventually, `n` has a constant value. An eternity extension can be used to prophesy this value. Indeed, take $M = \mathbb{N}$. Let relation R be given by

$$((\mathtt{i}, \mathtt{n}), m) \in R \quad \equiv \quad \mathtt{i} = 0 \ \vee\ \mathtt{n} = m\ .$$

It is easy to verify that R is a behaviour restriction: for any behaviour of K, one can choose for m the final value of `n`.

5 The simulation of Try by TryL

In this section, we prove that system TryL implements system Try. The specifications of Try and TryL are given in Sections 5.1 and 5.2. They are fairly similar. In both cases, all threads p have the same program, given by a list of atomic statements with consecutive line numbers. The current line number of thread p is given by the program counter $pc.p$. At each command of p, the value of $pc.p$ is incremented with 1, unless it is modified because of an **if**, **while**, or **goto** statement. Every step of the algorithm leaves the state unchanged or is the state change induced by a single step of a single thread.

For both specifications, the boolean result of the function *trylock* is encoded in the location. This is possible, because when thread p has obtained a result *true*, the thread must eventually call *unlock*. Between the function calls the thread is either idle, i.e., at the noncritical section *NCS*, or, when it has obtained the result *true*, at the critical section *CS*.

The observable state of each thread thus has one of four possible values: 0 when the thread is at *NCS*, 1 when the thread is trying to enter, 2 when the thread is at *CS*, 3 when the thread is exiting from *CS*. When *trylock* returns *false*, the thread goes from observable state 1 to observable state 0. The observation space is therefore the set of functions from *thread* to $\{0, 1, 2, 3\}$.

The state space X of the specification consists of all possible value assignments to the shared and private variables, as declared, including *pc*. Let $step_k(p)$ be the set of pairs (x, y), such that state x is transformed into state y when thread p executes line k of the program. At the lines k_0 of *NCS* and k_1 of *CS*, nothing more is done than incrementation of $pc.p$. Put $step(p) = \bigcup_k step_k(p)$. Initially, $pc.p = k_0$ for all threads p. The next-state relation N is given by

$$(x, y) \in N \quad \equiv \quad x = y \;\vee\; \exists\, p : (x, y) \in step(p) \ .$$

Therefore, $step(p)$ and $step_k(p)$ are commands in the sense of Section 4.3.

The supplementary property expresses weak fairness: every thread always eventually does a step unless it is at *NCS* or is disabled. Indeed, a thread at *NCS* is allowed to do a step and go to the next line, but it is not *expected* to do so. Formally, the supplementary property is defined by

(0) $$prop = \forall\, p : weakFair(fwd(p)) \ ,$$

where $fwd(p) = \bigcup_{k \neq k_0} step_k(p)$.

5.1 The abstract specification Try

Specification Try of Section 2 is formalized as follows.

The state space consists of the tuples $(\texttt{mu}, pc, c)$, where c is a private variable to hold the return value of *trylock*, and *pc* is the program counter. Initially $\texttt{mu} = \bot$ and $pc.p = 10$ and $c.p = false$ for all threads p. We use $k_0 = 10$ and $k_1 = 13$. The loop of Section 2 together with the abstract specification of the functions *trylock* and *unlock* gives rise to the following transition system for thread p:

```
10      $NCS$ ;
11      **if** $\texttt{mu} = \bot$ **then** $\texttt{mu} := p$ ; $c := true$ **endif** ;
12      **if** $\neg c$ **then goto** 10 **endif** ;
13      $CS$ ;
14      $\texttt{mu} := \bot$ ; $c := false$ ;
15      **goto** 10 .
```

Here, and henceforth, each numbered line is an atomic command. The dummy locations 12 and 15 are introduced, because after commands 11 and 14 the result need not be immediately observable.

Specification Try has the obvious invariant $c.p = (\texttt{mu} = p)$. One can therefore restrict the state space to the set where this invariant holds, replace line 12 by

```
12      **if** $\texttt{mu} \neq p$ **then goto** 10 **endif** ,
```

and remove the private variable c. This simplifies the specification.

5.2 The specification of TryL

Specification TryL of Section 3 is formalized as follows. The shared variables are

$$\begin{aligned}&\mathtt{x} : \mathit{thread}\ ;\\ &\mathtt{y} : \mathit{thread}_\perp = \perp\ ;\\ &\mathtt{bb} : \textbf{array}\ [\mathit{thread}]\ \textbf{of}\ \text{bool} = (\mathit{false}, \ldots)\ .\end{aligned}$$

We use $k_0 = 20$ and $k_1 = 33$. The code is translated into the transition system for thread p :

```
20      $NCS$ ;
21      **if** $\mathtt{y} \neq \perp$ **then**
22          **goto** 20 **endif** ;
23      $\mathtt{bb}[p] := true$ ;
24      $\mathtt{x} := p$ ;
25      **if** $\mathtt{y} \neq \perp$ **then**
26          $\mathtt{bb}[p] := false$ ; **goto** 20 **endif** ;
27      $\mathtt{y} := p$ ;
28      **if** $\mathtt{x} \neq p$ **then**
29          $\mathtt{bb}[p] := false$ ; $lis := allthreads$ ;
30          **while** $lis \neq \emptyset\ \wedge\ \mathtt{y} = p$ **do**
                **choose some** $kk \in lis$ ;
31              **await** $\mathtt{y} \neq p\ \vee\ \neg\mathtt{bb}[kk]$ ;
                **remove** $kk$ **from** $lis$ **endwhile** ;
32          **if** $\mathtt{y} \neq p$ **then goto** 20 **endif endif** ;
33      $CS$ ;
34      $\mathtt{y} := \perp$ ;
35      $\mathtt{bb}[p] := false$ ; **goto** 20 .
```

In comparison with the version given in Section 3, line numbers have been added, and a set-valued private variable *lis*, which holds the threads *kk* for which the loop body 31 has not yet been executed. In line 29, *allthreads* stands for the set of all threads. We introduce the notations

$$\begin{aligned}q\ \textbf{at}\ \ell &\equiv pc.q = \ell\ ,\\ q\ \textbf{in}\ L &\equiv pc.q \in L\ ,\end{aligned}$$

if ℓ is a line number and L is a set of line numbers.

Unfortunately, the proof that TryL refines Try needs (in Section 5.3) one critical invariant of the system TryL. This invariant depends on mutual exclusion. We therefore first prove mutual exclusion for TryL. Indeed, mutual exclusion holds. In fact, something stronger is valid: there are never two different threads both in the lines 33 and 34:

Pq0: $\quad q\ \textbf{in}\ \{33\ldots34\}\ \wedge\ r\ \textbf{in}\ \{33\ldots34\}\ \Rightarrow\ q = r\ .$

This is proved by means of a family of nine predicates. The proof is roughly the same as the proof given in a more complicated setting in [12, Section 4.2].

Predicate *Pq0* is threatened only by the commands of the lines 28 and 32. Remedies for these commands (see Section 4.3) are, respectively, the predicates

Pq1: $\quad q\ \textbf{in}\ \{27\ldots28\}\ \wedge\ r\ \textbf{in}\ \{33\ldots34\}\ \Rightarrow\ \mathtt{x} \neq q\ ,$
Pq2: $\quad q\ \textbf{at}\ 32\ \wedge\ \mathtt{y} = q\ \wedge\ r\ \textbf{in}\ \{27\ldots34\}\ \Rightarrow\ r\ \textbf{in}\ \{29\ldots32\}\ .$

Predicate *Pq1* is threatened only by the commands 25 and 32. *Pq2* is a remedy for command 32. A remedy for command 25 is

Pq3: $\quad q\ \textbf{in}\ \{33\ldots34\}\ \Rightarrow\ \mathtt{y} \neq \perp\ .$

Predicate *Pq2* is threatened only by command 30. A remedy is the predicate

Pq4 : q **in** $\{30 \ldots 31\} \ \wedge\ \mathtt{y} = q \ \wedge\ r$ **in** $\{27 \ldots 34\}$
$\Rightarrow\ \ r$ **in** $\{29 \ldots 32\} \ \vee\ r \in \mathit{lis}.q$.

Predicate *Pq3* is threatened only by the commands 28 and 34. *Pq0* is a remedy for command 34. A remedy for command 28 is

Pq5: q **at** $28 \ \wedge\ \mathtt{x} = q \ \Rightarrow\ \mathtt{y} \neq \bot$.

Predicate *Pq4* is threatened only by command 31. A remedy is formed by

Pq6 : q **in** $\{24 \ldots 29\} \ \Rightarrow\ \mathit{bb}[q]$,
Pq7 : q **in** $\{33 \ldots 34\} \ \Rightarrow\ \mathit{bb}[q] \ \vee\ \mathtt{y} = q$.

Predicate *Pq5* is threatened only by command 34. A remedy is *Pq1*. Predicate *Pq6* is inductive. Predicate *Pq7* is threatened only by the commands 27, 28, and 34. Remedies for commands 28 and 34 are *Pq6* and *Pq0*, respectively. A remedy for command 27 is the new predicate

Pq8 : q **at** $27 \ \wedge\ r$ **in** $\{33 \ldots 34\} \ \Rightarrow\ \mathit{bb}[r]$.

Predicate *Pq8* is threatened only by the commands 25, 28, and 32. Remedies are *Pq3*, *Pq6*, and *Pq2*, respectively. It follows that the conjunction *Pq** of the nine predicates is stable. As they all hold initially, the conjunction *Pq** is inductive. Therefore all these predicates are forward invariants. This concludes the proof of mutual exclusion: *Pq0*.

The refinement proof also needs progress of system TryL. To prove this, we first note the inductive invariants

Qq0 : $\mathtt{y} = q \ \Rightarrow\ q$ **in** $\{28 \ldots 34\}$,
Qq1 : $\mathtt{bb}[q] \ \Rightarrow\ q$ **in** $\{24 \ldots 29\} \cup \{33 \ldots 35\}$.

States where the invariants do not hold are irrelevant for the algorithm. They are therefore removed from the state space. This is formalized by introducing a specification TryL1 that only differs from TryL in its smaller state space. It has the same behaviors as TryL. Formally speeking, the identity function is an extension from TryL to TryL1.

The next thing is to prove that TryL1 has the progress property

(1) $\Box\Diamond[\mathtt{y} = \bot]$.

This formula means that in every behaviour every state with $\mathtt{y} \neq \bot$ is followed eventually by a state where $\mathtt{y} = \bot$. It is proved as follows. Consider a state with $\mathtt{y} \neq \bot$. In this and all following states, let S hold the set of the threads in $\{23 \ldots 35\}$. If $\mathtt{y} \neq \bot$, then S is nonempty because of *Qq0*. As long as $\mathtt{y} \neq \bot$ holds, the set S cannot grow because of the test in line 21; the set S can only shrink. Now consider the function

$$\mathit{vf}(p) = 37 - \mathit{pc}.p + 2 \cdot (\mathit{pc}.p < 30?\ \#\mathit{allthreads} : \#\mathit{lis}.p)\ .$$

If $p \in S$ then $\mathit{vf}(p) > 0$. Whenever thread $p \in S$ does a step and remains in S, it decreases $\mathit{vf}(p)$ because the forward steps increase $\mathit{pc}.p$ and the backward step from line 31 decreases $\#\mathit{lis}.p$. Therefore, the sum $\mathit{vf} = \sum_{p \in S} \mathit{vf}(p)$ decreases in every step of elements of S. Let S' be the subset of S of the threads that are enabled, i.e.,

$$S' = \{q \in S \mid q \ \textbf{at}\ 31 \ \Rightarrow\ \mathtt{y} \neq q \ \vee\ \neg\mathtt{bb}[\mathit{kk}.q]\}\ .$$

If the set S is nonempty, then S' is nonempty because of *Qq1*. If a step of some thread p modifies S', then $p \in S'$ and the step decreases vf. It follows that every thread $q \in S'$ is continuously enabled until some thread $p \in S'$ decreases vf.

In UNITY [2, 16], an assertion P **ensures** Q means that, if P holds, this remains true unless Q holds, and that the system has a command that is continuously enabled while $P \wedge \neg Q$ holds, and that establishes Q.

In the present case, the formal argument is

$$q \in S' \ \wedge \ vf \leq k+1 \ \wedge \ \mathtt{y} \neq \bot \ \textbf{ensures} \ vf \leq k \ \vee \ \mathtt{y} = \bot \ .$$

Indeed, thread $q \in S'$ remains enabled until it decreases vf or $\mathtt{y} = \bot$. Using the fact that **ensures** implies $\mapsto$ (leads-to), and the union rule for $\mapsto$, one obtains

$$vf \leq k+1 \mapsto vf \leq k \ \vee \ \mathtt{y} = \bot \ .$$

By transitivity of $\mapsto$ and another application of the union rule, one obtains $true \mapsto \mathtt{y} = \bot$, or equivalently $\Box\Diamond[\mathtt{y} = \bot]$.

Finally, every thread p is always eventually idle, that is: $true \mapsto p$ **at** 20, or equivalently

(2) $$\Box\Diamond[p \ \textbf{at} \ 20] \ .$$

Indeed, if thread p is not at line 20, by weak fairness it proceeds until it is disabled at line 20 or line 31. If it waits at line 31, eventually $\mathtt{y}$ becomes $\bot$, and hence $\mathtt{y} \neq p$. The latter predicate remains true. Therefore, from then onward, thread p is continuously enabled and proceeds to line 20.

The progress properties (1) and (2) are valid only because of the modifications of Lamport's code [13] mentioned at the end of Section 3. An alternative would be to postulate strong fairness.

5.3 How to construct a prophecy?

Specification TryL uses the variable `y` as a gate, which is closed if and only if $\mathtt{y} \neq \bot$. When the gate is open, $\mathtt{y} = \bot$, the first thread that executes line 27 closes the gate. In specification Try, the variable `mu` has the same role. TryL is therefore extended with a ghost variable `mu` that gets its value when the first thread executes line 27. In system Try, the value `mu` gets, is the thread that succeeds. In TryL, this thread has not yet been decided. Therefore, a prophecy of the succeeding thread needs to be constructed.

For this purpose, first, shared history variables `time` and `log` are introduced, such that `log` holds the sequence of consecutive threads that have succeeded, and `time` holds their number, as a discrete notion of time. So, initially $\mathtt{time} = 0$ and `log` is the empty sequence, and these variables are modified (only) in the extended line

34 $\quad \mathtt{y} := \bot \ ; \ \mathtt{log(time)} := p \ ; \ \mathtt{time{+}{+}} \ ;$

Specification TryL1 is thus extended to specification TryL2 with additional variables `log` and `time`. The natural relation between the state spaces of TryL and TryL2 is the one where the common variables have the same values, while `log` and `time` are ignored. This relation is an extension from TryL1 to TryL2, a so-called history extension.

The next phase is to introduce an eternity extension. Let M be the set of the infinite sequences of threads. Let BR be the relation between the state space of TryL2 and M given by

$$(x, \mathtt{m}) \in BR \quad \equiv \quad \forall\, n : n < x.\mathtt{time} \ \Rightarrow \ x.\mathtt{log}(n) = \mathtt{m}(n) \ .$$

As `time` never decreases and the values `log`(n) with $n <$ `time` are never modified, every behaviour of TryL2 determines a unique finite or infinite limit sequence `log`. If the sequence is infinite, it determines `m` by *BR*; if it is finite, it can be extended to an infinite sequence `m` that satisfies *BR*. This proves that predicate *BR* is a behaviour restriction. The state space can therefore be extended with M while assuming validity of *BR*. This results in an eternity extension [8], a strict simulation from TryL2 to TryL3. In specification TryL3, we have the ghost variables `log`, `time`, and `m`, and they are connected by predicate *BR*.

The next phase is to introduce a ghost variable `mu`, initially equal to $\perp$ which is modified in the lines 27 and 34.

27 $\quad \mathtt{y} := p\ ;\ \textbf{if}\ \mathtt{mu} = \perp\ \textbf{then}\ \mathtt{mu} := \mathtt{m}(\mathtt{time})\ \textbf{endif}\ ;$
34 $\quad \mathtt{y} := \perp\ ;\ \mathtt{log}(\mathtt{time}) := p\ ;\ \mathtt{time}{+}{+}\ ;\ \mathtt{mu} := \perp\ ;$

Here, `m(time)` can be regarded as the prophecy of `log(time)`.

The ghost variable `mu` satisfies the obvious inductive invariants

Mq0: $\quad \mathtt{mu} \neq \perp \quad \equiv \quad \mathtt{y} \neq \perp\ ,$
Mq1: $\quad \mathtt{mu} \neq \perp\ \Rightarrow\ \mathtt{mu} = \mathtt{m}(\mathtt{time})\ .$

Less obvious is the invariant that expresses that the value `mu` received in line 27, is a correct prophecy of the succeeding thread:

Bq0: $\quad q\ \textbf{in}\ \{33 \ldots 34\} \quad \Rightarrow \quad \mathtt{mu} = q\ .$

This predicate is proved as follows. *Bq0* holds trivially if q is at line 20. By property (2), every state is followed eventually by a state where *Bq0* holds. By Section 4.3, it therefore suffices to prove that

(3) $\quad (u, v) \in \mathit{step}(p)\ \wedge\ \mathit{Bq0}(q, v)\ \Rightarrow\ \mathit{Bq0}(q, u)\ .$

Here, we make it explicit that the predicate is about q and about either the post-state v or the pre-state u, and that p is the thread that performs the step. Moreover, in the proof of this implication, we may use that u and v satisfy the invariants that have been established earlier. Note that state variables u and v are used to avoid confusion with the program variables `x` and `y`.

Implication (3) holds trivially for all steps of thread p, except for the steps at line 34 and 27. The step at line 34 uses that the post-state v satisfies *BR*, in particular $\mathtt{m}(t) = \mathtt{log}(t) = p$ for the value t of `time` in the pre-state. Moreover, the pre-state u satisfies $\mathtt{mu} = \mathtt{m}(\mathtt{time})$ because of *Pq3*, *Mq0*, and *Mq1*. The proof for line 27 is easier; it only uses *Pq3* and *Mq0*. Note, that the invariant *Pq3* introduced in Section 5.2 is used here, twice.

Predicate *Bq0* is called a *backward* invariant because of the proof by means of Formula (3).

There is one other backward invariant:

Bq1: $\quad \mathtt{mu} = q\ \Rightarrow\ q\ \textbf{in}\ \{27 \ldots 34\}$
$\qquad \wedge\ (q\ \textbf{in}\ \{29 \ldots 32\}\ \Rightarrow\ \mathtt{y} = q)$
$\qquad \wedge\ (q\ \textbf{at}\ 28\ \Rightarrow\ \mathtt{x} = q\ \vee\ \mathtt{y} = q)\ .$

The proof is similar to the proof of *Bq0*. The predicate holds always eventually because of Formula (1) and *Mq0*. The backward step relation analogous to Formula (3) holds because of *Mq0* and *Bq0*.

5.4 The construction of the refinement mapping

In line 27, as refined in Section 5.3, the variable `mu` gets the value of the thread that gets access to the critical section because of *Bq0*. The assignment to `mu`, however, is done by the thread that is the first to modify `y`. This is not necessarily the thread that eventually gets the mutex, as it is the case in specification Try. If these two threads differ, we must somehow swap the roles of them; a modest application of Lamport's dictum "Processes are in the Eye of the Beholder" [14].

In order to prepare the swapping, we introduce a ghost variable `nu`, which has to be swapped with `mu`. This variable is initially $\perp$. It holds the number of the first thread that has executed line 27, as long as this thread is at line 28 and thread `mu` has not yet executed line 27. Therefore, the lines 27 and 28 become

```
27      y := $p$ ;
        if mu = $\perp$ $\wedge$ m(time) $\neq p$ then nu := $p$
        elsif mu = $p$ then nu := $\perp$ endif ;
        if mu = $\perp$ then mu := m(time) endif ;
28      if nu = $p$ then nu := $\perp$ endif ;
        if x = $p$ then goto 33 endif ;
```

Note that, complicated as they are, the lines 27 and 28 are still executed atomically. This is possible because `mu` and `nu` are ghost variables (used in the proof, but omitted in implementation). In this way one obtains the specification TryL4 with the obvious extension from TryL3 to TryL4. The composition of extensions is an extension. Therefore specification TryL4 is an extension of TryL by the projection function that forgets the auxiliary variables `log`, `time`, `m`, `mu`, and `nu`. This extension induces a strict simulation from TryL to TryL4.

The variable `nu` in TryL4 satisfies the invariants

Nq0: $\quad \texttt{nu} \neq \perp \;\Rightarrow\; \texttt{nu}\ \textbf{at}\ 28$,
Nq1: $\quad \texttt{nu} \neq \perp \;\Rightarrow\; \texttt{mu} \neq \perp \;\wedge\; \texttt{mu}\ \textbf{at}\ 27$.

Indeed, *Nq0* is inductive. Predicate *Nq1* is a forward invariant which is threatened only by the commands 27 and 34. It is preserved by command 34 because of *Bq0*. It is preserved by command 27 because of *Pq3*, *Pq4*, and *Mq0*, and because *Bq1* holds in the post-state.

Now everything is prepared to construct the refinement mapping from TryL4 to specification Try of Section 5.1, with private variable c eliminated. This is the function f that maps a state x of TryL4 to the state u of Try given by

$$
\begin{array}{l}
u.\texttt{mu} = x.\texttt{mu}\ , \\
u.pc.q = mpc(x.pc.q, x.\texttt{mu} = q, x.\texttt{nu} = q) \ \text{ where} \\
\quad mpc(k, b_1, b_2) = \\
\qquad (\, k = 20\ ?\ 10 \\
\qquad :\, 33 \leq k\ ?\ k - 20 \\
\qquad :\, k = 27 \ \wedge\ b_1\ ?\ 12 \\
\qquad :\, k = 28 \ \wedge\ b_2\ ?\ 11 \\
\qquad :\, k = 22 \ \vee\ k = 26 \ \vee\ 28 \leq k\ ?\ 12 \\
\qquad :\, 11\ )\ .
\end{array}
$$

Here the booleans b_1 and b_2 serve in swapping the actions of `mu` and `nu`.

The correspondence between the line numbers of TryL4 and Try is suggested by lines between the two transition systems in the diagram. To avoid cluttering of arrows the letter Z represents *NCS*. Circles indicate the critical sections.

$$\begin{tikzcd}
 & & & & & & & & 31 \arrow[d, shift left] & Z & & & \\
 & & 22 \arrow[dl] & & Z & 26 \arrow[l] & & 29 \arrow[r] & 30 \arrow[u, shift left] \arrow[r] & 32 \arrow[u] \arrow[d] & & & \\
\text{TryL4:} & Z \arrow[r] \arrow[d, dash] & 21 \arrow[r] \arrow[u] & 23 \arrow[r] & 24 \arrow[r] & 25 \arrow[r] \arrow[u] & 27 \arrow[r] \arrow[d, dash] \arrow[dr, dash] & 28 \arrow[rr] \arrow[u] \arrow[d, dash] \arrow[dl, dash] & & \textcircled{33} \arrow[r] \arrow[d, dash] & 34 \arrow[r] \arrow[d, dash] & 35 \arrow[r] \arrow[d, dash] & Z \arrow[d, dash] \\
\text{Try:} & Z \arrow[rrrrr] & & & & & 11 \arrow[r] & 12 \arrow[rr] \arrow[llllll, bend left] & & \textcircled{13} \arrow[r] & 14 \arrow[r] & 15 \arrow[r] & Z
\end{tikzcd}$$

It is clear that function f maps start states of TryL4 to start states of Try. The hard work is to show that it maps steps of TryL4 to steps of Try.

For the step of thread p at line 27 there are four different cases:
1. Under the precondition $\texttt{mu} = \perp$, it is mapped to the step of Try from line 11 by the thread $\texttt{m}(\texttt{time})$. This assertion uses the invariants *Pq3*, *Pq5*, *Mq0*, *Bq1*, *Nq1*.
2. Under the precondition $\texttt{mu} = p \ \wedge \ \texttt{nu} \neq \perp$, it is mapped to step 11 of thread $\texttt{nu}$ because of *Nq0*.
3. Under the precondition $\texttt{mu} \neq p \ \wedge \ \texttt{mu} \neq \perp$, it is mapped to the step 11 of p because of *Nq0*.
4. In the case of the remaining precondition $\texttt{mu} = p \ \wedge \ \texttt{nu} = \perp$, it is mapped to *skip* (no state change).

All other steps of thread p in TryL4 are mapped to *skip* or steps of p in Try. In particular, the steps of thread p from line 28 are mapped to steps of p from lines 11 or 12, when $\texttt{nu} = p$ or $\texttt{x} = p$, respectively, because of *Bq0* and *Nq1*.

The steps of p from lines 21 and 25 when $\texttt{y} \neq \perp$ are mapped to the step from 11 because of *Mq0*. These steps are mapped to *skip* when $\texttt{y} = \perp$ (in the second case because of *Bq1*).

The steps from lines 22 and 26 are mapped to steps from line 12 because of *Bq1*. The step from line 32 is mapped to the step from line 12 because of *Bq0* and *Bq1*. The step from line 34 is mapped to the step from line 14 because of *Bq0*.

The remaining steps need no invariants. All steps of p to line 20 are mapped to steps of p from lines 12 or 15 to line 10. The step from line 33 is mapped to the step from lines 13. All other steps are mapped to *skip*.

It remains to show that every behaviour *xs* of TryL4 is mapped to a state sequence of Try that satisfies the supplementary property of Try. Let sequence *us* be the image of the sequence *xs*. Let p be a thread. By Formula (0), it suffices to prove that *us* is weakly fair for $\mathit{fwd}(p)$. Behaviour *xs* satisfies $\Box\Diamond[p \ \textbf{at} \ 20]$ by Formula (2). Therefore, the sequence *us* satisfies $\Box\Diamond[p \ \textbf{at} \ 10]$. As $\mathit{fwd}(p)$ is disabled at line 10, it follows that $\mathit{fwd}(p)$ is always eventually disabled. This proves that *us* is weakly fair for $\mathit{fwd}(p)$.

This concludes the proof that the function $x \mapsto u$ is a refinement mapping from TryL4 to Try. Composing it with the strict simulation from TryL to TryL4, one thus obtains:

Theorem. There is a strict simulation from system TryL to system Try.

6 Conclusions

Branching time temporal logic, CTL, rejects the useful implementation relation between TryL and Try, which is accepted by LTL, linear temporal logic. This supports the opinion of Vardi and Nain [19, 17], that LTL is to be preferred over CTL in concurrency verification. The present case study further indicates that, in cases where a simulation relation for CTL does not exist, the construction of an LTL simulation may need prophecy or eternity variables.

Abadi and Lamport [1] used König's lemma to justify prophecy variables. This lemma is also used for the backward simulations of Lynch and Vaandrager [15]. Therefore, the soundness of prophecy variables or backward simulations imposes finiteness conditions that limit the applicability and imply heavy proof obligations, compare [6]. We therefore prefer eternity variables. They are easier to prove sound and have wider applicability.

The mechanical proof, done with the proof assistant PVS [18], deviates marginally from the proof rendered here, for the sake of readability of the paper and simplicity of the proof script. The proof script is available at [10], but can only be advised for readers with a working version of PVS. The proof assistant was in particular indispensable for the treatment of the invariants and the final refinement mapping.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.

[2] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison–Wesley, 1988.

[3] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71, 1981.

[4] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569, 1965.

[5] W. H. Hesselink. Using eternity variables to specify and prove a serializable database interface. *Sci. Comput. Program.*, 51:47–85, 2004.

[6] W. H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. On Comp. Logic*, 6:175–201, 2005.

[7] W. H. Hesselink. A criterion for atomicity revisited. *Acta Inf.*, 44:123–151, 2007.

[8] W. H. Hesselink. Universal extensions to simulate specifications. *Information and Computation*, 206:108–128, 2008.

[9] W. H. Hesselink. Simulation refinement for concurrency verification. *Sci. Comput. Program.*, 76:739–755, 2011.

[10] W. H. Hesselink. PVS proof scripts for trylock, the rectangle algorithm, and access selection. `http://wimhesselink.nl/mechver/trylock`, 2020.

[11] W. H. Hesselink. UNITY and Büchi automata. *Formal Aspects of Computing*, 33:185–205, 2021. https://doi.org/10.1007/s00165-020-00528-x.

[12] W. H. Hesselink, P. A. Buhr, and D. Dice. Fast mutual exclusion by the Triangle algorithm. *Concurrency Computat.: Pract. Exper.*, 30(4), February 2018. https://doi.org/10.1002/cpe.4183.

[13] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5:1–11, 1987.

[14] L. Lamport. Processes are in the eye of the beholder. *Theor. Comput. Sci.*, 179:333–351, 1997.

[15] N. Lynch and F. Vaandrager. Forward and backward simulations, part I: untimed systems. *Inf. Comput.*, 121:214–233, 1995.

[16] J. Misra. *A discipline of multiprogramming: programming theory for distributed applications.* Spinger V., New York, 2001.

[17] S. Nain and M. Y. Vardi. Branching vs. linear time: semantical perspective. In K. S. Namjoshi et. al., editor, *Automated Technology for Verification and analysis, 5th International Symposium, ATVA 2007*, volume 4762 of *LNCS*, pages 19–34, 2007.

[18] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 7.1, System Guide, Prover Guide, PVS Language Reference*, 2020. `http://pvs.csl.sri.com`, accessed 1 Dec. 2021.

[19] M. Y. Vardi. Branching versus linear time: final showdown. In *7th International Conference on tools and algorithms for the construction and analysis of systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer V., 2001.