Real Time is Really Simple

Leslie Lamport
Microsoft Research

Technical Report MSR-TR-2005-30

4 March 2005
Revised 16 August 2005

Abstract

It is easy to write and verify real-time specifications with existing languages and methods; one just represents time as an ordinary variable. The resulting specifications can be verified with ordinary model checkers. This basic idea and some less obvious details are explained with simple examples.

Contents

1 Introduction **1**

2 Specifying a Simple Algorithm **4**
2.1 The Algorithm 4
2.2 The Untimed Version in TLA$^+$ 5
2.3 Adding Timing to an Untimed Specification 8
2.4 Specification *FSpec*1: Progress Through Liveness 10
2.5 Specification *FSpec*2: Progress Through Timing Bounds . . . 17

3 Proving Correctness of Fischer's Algorithm **19**
3.1 Proof of Mutual Exclusion 21
3.2 Proof of Eventual Progress 21
3.3 Proof of Real-Time Progress 24

4 A Simple Distributed Algorithm **29**

5 Avoiding Zeno Specifications **35**

6 Model Checking **37**
6.1 General Observations 37
6.2 Model Checking with Symmetry 38
6.2.1 Specifications and Temporal Properties 38
6.2.2 Symmetry 38
6.2.3 Model Checking 39
6.2.4 Expressing Symmetry 41
6.2.5 Symmetry Under Time Translation 43
6.2.6 Periodicity and Zeno Behaviors 45
6.2.7 Checking Inductive Invariance 49
6.3 Model Checking Our Specifications 50
6.3.1 Modifying the Specifications 50
6.3.2 Measurements 51
6.3.3 Specification *FSpec*1 52
6.3.4 Specification *FSpec*2 56
6.3.5 Specification *HFSpec*2 56
6.3.6 Specification *LSpec* 57
6.4 Comparison With Uppaal 59
6.4.1 The Leader Algorithm 61
6.4.2 Fischer's Algorithm 64

7 Conclusion **66**
7.1 Objections . . . 66
7.2 Hybrid-System Specifications . . . 67
7.3 Concluding Remarks . . . 67

Acknowledgements **68**

References **68**

1 Introduction

There is a simple and obvious way to describe a real-time algorithm, protocol, or system design:

- Introduce a variable *now*, whose value represents the current time, and model the passage of time with an action that increments *now*.
- Express timing bounds as follows:
  - Describe a lower bound by allowing certain actions to occur only if *now* is greater than some value.
  - Describe an upper bound by not allowing the action that increments *now* to occur if that would cause a violation of the bound.

This produces what I will call an *explicit-time* description.

No special language or tools are required to handle explicit-time descriptions. They can be written and debugged exactly like untimed ones; they can be verified with conventional assertional reasoning; and they can be checked with conventional model checkers.

Another approach is to write *implicit-time* descriptions. One creates a new language with special constructs for expressing timing properties, where the current time is implicit rather than being represented by a variable. This is often done by modifying an existing untimed language. Examples include timed CSP [38], timed Petri nets [42], and timed I/O automata [30].

A new language requires a new semantics, new tools, and new proof rules. Implicit-time methods have therefore been the subject of hundreds of papers and theses. In contrast, nothing new is required to handle explicit-time descriptions. They are apparently so simple and obvious that very little has been written about them. Except for a handful of papers using TLA [2, 26], the only published work I know of in which both the algorithm or system description and the properties to be checked are expressed in an ordinary untimed language is that of Dutertre and Sorea [14] and Brinksma, Mader, and Fehnker [9]. The very fact that such specifications are easily model checked seems never to have been stated explicitly in print, though it has been known for many years and is implicit in several published results [16]. Explicit-time specifications of the correctness properties to be proved have been used [33], and Ostroff has described a method for translating an implicit-time description into an explicit-time one for verification [34]. However, other than the aforementioned TLA-based work and the

work of Dutertre and Sorea and of Brinksma et al., I know of no previously published, purely explicit-time approach.

An implicit-time approach may be necessary if we want to reason about an actual implementation—for example, one written in a programming language like Java or in a hardware description language. Modifying a real Java program by adding extra variables to express the timing constraints would be difficult and error-prone, and it would alter the program's timing properties. The earliest published real-time verification method that I know of, by Bernstein and Harter [7], was an implicit-time approach based on a toy programming language. They presumably hoped that their method could be extended to verify actual programs. However, the difficulties encountered in trying to extend ordinary program verification to actual programs makes such a hope now seem naive. Most recent work on real-time verification has been directed toward higher-level descriptions of algorithms, protocols, or system designs. There is no *a priori* reason to prefer an implicit-time approach for such higher-level descriptions.

The preponderance of publications advocating implicit-time approaches seems to have created the impression that explicit-time ones are not as good—or perhaps even do not exist. In fact, implicit-time approaches offer no practical advantage over explicit-time ones—except perhaps for use in model checking, discussed below. Their proponents may argue that the more complex implicit-time approaches are more elegant, abstract, hierarchical, or compositional, or have some other wonderful attributes. I will not attempt to challenge such claims. I assert only that implicit-time methods are no better than the simple explicit-time approach for the practical problem of describing and verifying the correctness of a high-level algorithm, protocol, or system design.

A number of algorithms and programs have been developed especially for model-checking real-time specifications [3, 18, 27, 43]. They have used implicit-time languages, usually based on timed automata. Most of these languages seem to have been developed for modeling finite-state controllers, and they are not sufficiently expressive for describing more complicated systems such as network protocols. Such languages cannot model the simple distributed algorithm of Section 4. The only real-time model checker capable of handling this example that I know of is Uppaal [27].

Special-purpose model checkers may be needed if one wants to verify a specification that models continuous time, since representing continuous time by an ordinary variable produces an inherently infinite-state specification. (Dutertre and Sorea use a more general approach that handles some other infinite-state specifications as well.) However, in practice, model

checking is used to debug a specification by checking small instances of it. Discrete-time specifications seem adequate for that purpose, and ordinary model checkers can be applied to them.

The previously published recipe for writing explicit-time TLA specifications [2], formulated more recently in TLA$^+$ [25], is a bit subtle because it conjoins timing properties to a complete untimed specification. This makes the method less obvious and a little more complicated than necessary. Moreover, the TLC model checker for TLA$^+$ cannot handle the resulting specifications. So, I describe here an even simpler and more obvious approach and show how to model check the resulting specifications.

The approach described here works with almost any language or formalism based on state machines [8] or shared-memory programs [4, 11, 40]. However, because they rely on the global variable *now*, explicit-time specifications are difficult to write in process-based languages and formalisms with no explicit global state, such as CCS [32], CSP [19], Petri nets [39], streams [10], and I/O automata [29]. Instead, those formalisms are usually extended to allow writing implicit-time specifications.

Section 2 explains with a simple example how to write an explicit-time specification in TLA$^+$. Section 3 discusses how to prove the correctness of such a specification. Section 4 is devoted to a more sophisticated example—a real-time message-passing algorithm. A potential pitfall of the approach and how to avoid it are discussed in Section 5. Section 6 explains how to model check explicit-time specifications and presents the results of checking the examples of Sections 2 and 4. It also describes how ordinary model checkers compare with the Uppaal real-time model checker on these examples. A concluding section briefly discusses objections to the method that I have heard and the extension of explicit-time specifications to hybrid systems.

The method of writing and checking real-time specifications described here appears not to have been published before—perhaps because it seems too simple to be worth publishing. Yet that simplicity is what makes the method so appealing. In the face of a succession of more complicated implicit-time methods, it is useful to state the obvious: a simple explicit-time approach works at least as well.

Although basically simple, some aspects of writing and verifying explicit-time specifications may not be obvious. These include the different choices of which actions update timers (Section 2.3), the relation between the proofs of liveness and of real-time progress (Section 3.3), how to check that a specification is nonZeno (Section 5), and several techniques for model checking the specifications (Section 6).

2 Specifying a Simple Algorithm

The explicit-time approach is described mainly in terms of a single example. Space constraints require that the example be simple. The question then arises of whether larger problems can be handled with explicit time, or whether implicit-time methods are needed. I can answer this only by observing that explicit-time specifications can be written in languages such as TLA$^+$ that have been applied to problems at least as large as any tackled by the formalisms on which implicit-time methods are based [6]. Adding a variable *now* clearly does not change how things scale—except perhaps for model checking, which is discussed in Section 6.

I have chosen as the example Fischer's mutual exclusion algorithm [41], mainly because it is the simplest interesting real-time algorithm I know. It is a shared-memory, multithreaded algorithm. There seems to be a common misconception that methods based on shared variables or global state are not good for specifying and reasoning about distributed systems, and one must instead use a process-based method. It has been known for years that shared-variable languages are fine for specifying distributed systems, and that one wants to reason about such systems in terms of global invariants [21]. Section 4 briefly presents an explicit-time specification of a real-time distributed algorithm. It should serve as yet one more illustration that process-based methods offer no practical advantage for specifying or reasoning about distributed systems.

2.1 The Algorithm

Fischer's algorithm uses a single shared variable x whose value is either a thread identifier or the special value *NotAThread*; its initial value is *NotAThread*. Figure 1 shows the program for thread t without the timing constraints needed to ensure mutual exclusion. Those constraints are as

$$
\begin{array}{rl}
ncs: & \text{noncritical section;} \\
a: & \textbf{wait until } x = NotAThread; \\
b: & x := t; \\
c: & \textbf{if } x \neq t \textbf{ then goto } a; \\
cs: & \text{critical section;} \\
d: & x := NotAThread; \;\; \textbf{goto } ncs;
\end{array}
$$

Figure 1: The program of thread t, with timing constraints omitted.

follows, where δ and ϵ are parameters:

- Step b must be executed at most δ seconds[1] after the preceding execution of step a.
- Step c cannot be executed until at least ϵ seconds after the preceding execution of step b.

Additional constraints are needed to ensure progress. Two kinds of constraints can be used:

1. Suitable fairness conditions on certain program statements, or
2. Upper bounds on the execution times of those statements.

I explain how to specify both kinds of constraints and prove their corresponding progress properties.

2.2 The Untimed Version in TLA$^+$

Before discussing the timing constraints, I describe how the untimed algorithm can be specified in TLA$^+$, a complete specification language based on the logic TLA [25]. TLA has a trace-based semantics in which a specification describes a set of behaviors, and a behavior is a sequence of states. I assume no knowledge of TLA$^+$. All TLA$^+$ syntax appearing in the specifications that differs from ordinary mathematical or programming notation is explained. I do assume that the reader is at least somewhat acquainted with the basic concepts of safety, liveness, and fairness [40].

The customary form of a TLA specification is

$$\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{Liveness}$$

where the state predicate *Init* describes the possible initial states, the next-state action *Next* describes possible state changes, *Liveness* is the conjunction of liveness or fairness conditions, and *vars* is the tuple of specification variables.

We first determine the specification variables. We obviously want a variable x that represents the program variable of that name. We also need a variable to represent the algorithm's control state. We introduce a

[1] For the sake of euphony, I assume that the unit of time is the second. There is no way for a specification, which is ultimately a mathematical formula, to state formally that one unit of time equals a certain multiple of the frequency of an emission line of cesium.

variable pc, where $pc[t]$ describes the control state of thread t. For example, $pc[t] =$ "a" means that control in thread t is at program statement a.

We won't specify any liveness requirement on the untimed algorithm, so the interesting part of its specification is the definition of the next-state action *Next* that describes possible steps—that is, possible pairs of successive states in an allowed behavior of the algorithm. An algorithm step is a step of some thread, so *Next* is defined by

$$Next \triangleq \exists\, t \in Thread : TNext(t)$$

where $TNext(t)$ is the next-state action of thread t. A step of thread t is one that is performed by executing one of t's program statements, so

$$TNext(t) \triangleq NCS(t) \lor StmtA(t) \lor StmtB(t) \lor StmtC(t) \lor CS(t) \lor StmtD(t)$$

where each of the actions $NCS(t)$, $StmtA(t)$, ... describes an execution of the corresponding program statement. We now define these actions.

In a conventional programming language, a step that changes the value of a variable v can be described by an assignment statement $v := exp$. In TLA$^+$, the corresponding change to v is expressed by the action expression[2] $v' = exp$. However, while the assignment statement leaves all variables other than v unchanged[3], the TLA$^+$ action expression says nothing about the new values of any other variables. The change to an array variable v described by an assignment statement $v[c] := exp$ is represented by the action expression $v' = aexp$, where $aexp$ is an expression whose value is the same as that of v, except $aexp[c] = exp$. This expression $aexp$ is written in TLA$^+$ as

$$[v \text{ EXCEPT } ![c] = exp]$$

Before defining the actions corresponding to a thread's program statements, we define some operators for describing the control state. The state predicate $At(t, loc)$ is true iff control in thread t is at program location loc, and $GoTo(t, loc)$ is the action expression that describes t's control state changing to loc:

$$At(t, loc) \triangleq pc[t] = loc$$
$$GoTo(t, loc) \triangleq pc' = [pc \text{ EXCEPT } ![t] = loc]$$

[2] I am using the term *action expression* to mean a formula containing primed and unprimed variables. I reserve the term *action* to mean an action expression that determines the new (primed) values of all the specification variables as functions of their old (unprimed) values.

[3] More precisely, in a programming language, what variables an assignment statement leaves unchanged must be determined from the context [28].

We define $GoFromTo(t, loc1, loc2)$ to assert that control in thread t goes from $loc1$ to $loc2$:

$$GoFromTo(t, loc1, loc2) \triangleq At(t, loc1) \land GoTo(t, loc2)$$

We assume that each of the statements a, b, c, and d is an atomic operation, meaning that its execution is described by a single step. Statement a of thread t can then be represented by the action $StmtA(t)$ defined as follows, where a list of assertions bulleted with $\land$ (or $\lor$) denotes their conjunction (or disjunction), and the expression UNCHANGED exp is defined to equal $exp' = exp$, asserting that expression exp is left unchanged:

$$\begin{aligned} StmtA(t) \triangleq\ & \land\ x = NotAThread \\ & \land\ GoFromTo(t, \text{"a"}, \text{"b"}) \\ & \land\ \text{UNCHANGED } x \end{aligned}$$

A conjunct of an action that, like $x = NotAThread$, contains no primed variables is an enabling condition for the action. Action $StmtC(t)$, describing program statement c of thread t, is:

$$\begin{aligned} StmtC(t) \triangleq\ & \land\ At(t, \text{"c"}) \\ & \land\ \text{IF } x \neq t \text{ THEN } GoTo(t, \text{"a"}) \\ & \phantom{\land\ \text{IF } x \neq t\ } \text{ELSE } GoTo(t, \text{"cs"}) \\ & \land\ \text{UNCHANGED } x \end{aligned}$$

The definitions of actions $StmtB(t)$ and $StmtD(t)$ should be obvious. We represent the noncritical section by the action

$$\begin{aligned} NCS(t) \triangleq\ & \land\ GoFromTo(t, \text{"ncs"}, \text{"a"}) \\ & \land\ \text{UNCHANGED } x \end{aligned}$$

Since TLA specifications allow "stuttering steps" that do not change any of the specification's variables, our specification allows the execution of t's noncritical section to consist of any number of steps that do not change pc or x, followed by an $NCS(t)$ step that changes $pc[t]$ from "ncs" to "a" and leaves x unchanged. The critical section is similarly represented by a single action $CS(t)$, so execution of the critical section can also consist of any sequence of steps that change neither pc nor x.

Except for the definitions of the initial predicate $Init$, actions $StmtB(t)$, and $StmtD(t)$, and the liveness condition $Liveness$, this completes the specification of the untimed algorithm. I will not bother writing the remaining definitions.

2.3 Adding Timing to an Untimed Specification

In an explicit-time specification of the actual real-time algorithm, a variable *now* represents the current time. For our purposes, it makes no difference whether time is continuous or advances in femtosecond steps. Moreover, instead of representing nature's time (measured in some inertial coordinate system), *now* could represent some particular digital clock. We could therefore pretend that time is discrete and let *now* assume only integral values. However, it's just as easy to represent continuous time by letting *now* assume real values.

The variable *now* is incremented by a special action, which I like to call *Tick*, that can increment *now* by any positive real value. Thus, even though the range of possible values of *now* is continuous, *now* increases in discrete ticks. We can view a behavior, which is a sequence of states, to be a sequence of snapshots of the system, each taken at some instant of time. These snapshots are taken often enough to capture every state reached by the *ordinary* variables—that is, the variables of the untimed systems, such as x and pc in the specification of Fischer's algorithm. Since the ordinary variables are assumed to change in discrete steps, and *now* can be incremented by arbitrary amounts, nothing is lost by representing continuous time in this way.

Timing constraints are expressed by adding special timer variables. A constraint that something must occur within or after τ seconds is expressed by having a variable *timer* set to time out in precisely τ seconds. There seem to be three basic ways to do this. The method I have used before [2], which is perhaps the simplest, is with *expiration* timers. An expiration timer *timer* is left unchanged by the *Tick* action. It can be set to $now + \tau$, in which case the timeout occurs when $now = timer$, or it can be set to *now*, in which case the timeout occurs when $now = timer + \tau$. A possibly more intuitive way is with a *countdown* timer that is decreased by the *Tick* action. If the timer is set to τ, then the timeout occurs when it becomes equal to 0. The third type of timer is a *count-up* timer that is increased by the *Tick* action. If the timer is set to 0, then timeout occurs when it becomes equal to τ. Timers may be reset when not in use; countdown and count-up timers that have been reset are left unchanged by the *Tick* action. Typically, the value of a reset timer equals 0, ∞, or $-\infty$. Although it makes little difference to TLC, some model checkers require the use of countdown or count-up timers, so I will use countdown timers here.

Dutertre and Sorea [14] and Brinksma et al. [9] use an alternative approach in which timer variables store the exact time at which future actions

will occur. Such a timer variable is set, perhaps nondeterministically, when it becomes possible to predict that the action will happen. The *Tick* action advances *now* directly to the time at which the next action is to occur. This approach eliminates intermediate *Tick* actions and could reduce the number of reachable states in some models, but it can be more awkward for describing some systems. I will not consider it further.

If an operation of an untimed program is represented by action A, then a timing constraint on when that operation may or must occur is expressed as a bound τ on the length of time between when some predicate P becomes true and when an A step may or must occur. (Typically, P is the predicate ENABLED A that is true iff A is enabled.) Such a constraint is expressed by having a timer variable that is set to τ when either the value of P changes from false to true or an A step occurs that leaves P true. The timer is reset when P becomes false. For most timing constraints that occur in practice, any A step makes P false.

There are two kinds of timing constraints. A lower-bound constraint requires that the operation not occur until P has been true for at least τ seconds. An upper-bound constraint requires that the operation must occur if P has been true for τ seconds. For each of these constraints, we have the choice of allowing the elapsed time before the operation occurs to equal exactly τ seconds, or of requiring a strict inequality to hold. With continuous time, the choice has no significance because an infinitely precise measurement would be required to determine whether exactly τ seconds had passed. When using count-down timers, it's a bit more convenient for a lower-bound constraint to allow the operation to occur when τ seconds or more have elapsed. This is done by representing the program operation with the action $(\mathit{timer} = 0) \wedge A$, taking 0 to be the special value indicating that the timer has been reset. For an upper-bound constraint, we arbitrarily choose to require that the operation occur strictly less than τ seconds after P becomes true. This is done by requiring that the *Tick* action increase *now* by less than the current value of *timer*. In other words, *Tick* must imply $\mathit{now}' < \mathit{now} + \mathit{timer}$. In this case, P must imply ENABLED A; otherwise, the requirement could assert that a step of a disabled action must occur, which is impossible.[4] It is most convenient to let ∞ be the special value indicating that the timer is reset.

We next must decide what specification actions set and reset variable *timer*. There are two choices: the *Tick* action, which increments *now*, or

[4]More precisely, this would result in a Zeno specification. Such specifications are discussed in Section 5.

the "ordinary" actions that modify the ordinary variables. Either is possible, but I find it easier to let the ordinary actions do it.[5] We therefore conjoin, to actions of the untimed specification, action expressions that specify the new values of timer variables. For a multithreaded algorithm like Fischer's, there are three natural ways of doing this. Action expressions describing a timer's new value can be conjoined to:

- each of the individual actions $NCT(t)$, $StmtA(t)$, etc.,
- the action $TNext(t)$, which describes the steps of thread t, or
- the entire next-state action $Next$.

Different choices could be made for different timers in the same specification.

We use the first method for our first specification of the Fischer algorithm, in which progress is achieved by liveness. We will use the second for our second specification, in which progress is achieved by timing bounds.

2.4 Specification $FSpec1$: Progress Through Liveness

The specification of the untimed algorithm was developed above in a top-down fashion, starting with the complete specification. However, TLA$^+$ requires that every identifier be defined or declared before it is used, which leads to a bottom-up description. (Splitting the specification into multiple modules would allow it to be read in a more top-down fashion.)

We put declarations and definitions shared by both specifications in module *FischerPreface*, which appears in Figure 2. The module begins by importing the standard *Reals* module that defines the set *Real* of real numbers and the usual arithmetic operations on them. It then defines $Max(a, b)$ to be the maximum of a and b, if they are real numbers. Next comes the declaration of the specification's constant parameters: the set *Thread* of thread identifiers and the timing bounds *Delta* and *Epsilon*. The ASSUME statement asserts assumptions about those bounds. The constant *NotAThread* is then defined to be an arbitrary value that is not a thread.

Module *FischerPreface* next declares the specification's variables and defines some state functions. Variables *ubTimer* and *lbTimer* are arrays of timers used to express upper and lower time bounds, respectively. For later use, *vars* is defined to be the tuple of all specification variables.

The state predicate *Init* specifies the initial values of the variables. It uses the TLA$^+$ notation that $[x \in S \mapsto exp]$ is the function f with domain

[5] There are lower-bound constraints for which the corresponding timer *must* be set by the ordinary actions, but such constraints do not seem to occur in practice.

```
---------------------------- MODULE FischerPreface ----------------------------
EXTENDS Reals

Max(a, b) ≜ IF a ≥ b THEN a ELSE b

CONSTANTS Thread, Delta, Epsilon

ASSUME ∧ (Delta ∈ Real) ∧ (Epsilon ∈ Real)
       ∧ 0 < Delta
       ∧ Delta ≤ Epsilon

NotAThread ≜ CHOOSE t : t ∉ Thread
-------------------------------------------------------------------------------
VARIABLES x, pc, ubTimer, lbTimer, now
vars ≜ ⟨x, pc, ubTimer, lbTimer, now⟩

Init ≜ ∧ pc      = [t ∈ Thread ↦ "ncs"]
       ∧ x       = NotAThread
       ∧ now     = 0
       ∧ ubTimer = [t ∈ Thread ↦ Infinity]
       ∧ lbTimer = [t ∈ Thread ↦ 0]

MutualExclusion ≜
  ∀ t1, t2 ∈ Thread : (t1 ≠ t2) ⇒ (pc[t1] ≠ "cs") ∨ (pc[t2] ≠ "cs")
-------------------------------------------------------------------------------
At(t, loc)     ≜ pc[t] = loc
GoTo(t, loc)   ≜ pc' = [pc EXCEPT ![t] = loc]
GoFromTo(t, loc1, loc2) ≜ At(t, loc1) ∧ GoTo(t, loc2)
-------------------------------------------------------------------------------
TimedOut(t, timer) ≜  timer[t] = 0
===============================================================================
```

Figure 2: Module *FischerPreface*, containing declarations and definitions common to our two specifications of Fischer's algorithm.

S such that $f[x] = exp$ for all x in S. (An array indexed by a set S is just a function whose domain is S.) Since absolute time values have no significance to the algorithm, I have chosen to initialize *now* with the arbitrary value 0. Another natural choice would be to let *now* initially equal any real number—a choice expressed by the assertion $now \in Real$.

The state predicate *MutualExclusion* asserts that two different processes are not both in their critical section. (The symbol $\Rightarrow$ denotes implication.) Mutual exclusion means that this predicate is always true. So, to prove that Fischer's algorithm implements mutual exclusion, we have to prove that *MutualExclusion* is an invariant of the algorithm's specification.

The module next defines the operators *At*, *GoTo*, and *GoFromTo* introduced in Section 2.2 above. It concludes by defining $TimedOut(t, timer)$ to be the state predicate asserting that timer variable $timer[t]$ has timed out. For count-down timers, this predicate is simply $timer[t] = 0$.

Our first specification continues in Module *Fischer*1 of Figure 3, which begins by importing module *FischerPreface*. It then defines two action expressions for setting timer variables: $SetTimer(t, timer, tau)$ sets timer $timer[t]$ to time out in *tau* seconds and $ResetUBTimer(t, timer)$ resets it to *Infinity*, which is defined in the *Reals* module to be a value greater than any real number.

The structure of the timed specification is similar to that of the untimed one, with the algorithm's next-state action *Next*, the next-state action $TNext(t)$ of thread t, and actions $NCS(t)$, $StmtA(t)$, ... , corresponding to the program statements. However, there is also the *Tick* action, so *Next* is defined by

$$Next \triangleq Tick \lor (\exists t \in Thread : TNext(t))$$

Moreover, the actions corresponding to program statements specify the new values of *lbTimer* and *ubTimer*, and they assert that *now* is left unchanged. An additional enabling condition is also needed in action $StmtC(t)$ to express the lower-bound timing constraint on the execution of statement c.

The Fischer algorithm has one upper-bound timing constraint—that a thread must execute statement b within δ seconds after it executes statement a. In our specification, this means that action $StmtB(t)$ must be executed within *Delta* seconds of when control reaches b. We express this constraint with the timer $ubTimer[t]$. Since control reaches b only by a $StmtA(t)$ step, we let $StmtA(t)$ set the timer. As we'll see, $StmtA(t)$ should leave $lbTimer[t]$

MODULE *Fischer1*

EXTENDS *FischerPreface*

$SetTimer(t, timer, tau) \triangleq timer' = [timer \text{ EXCEPT } ![t] = tau]$

$ResetUBTimer(t, timer) \triangleq SetTimer(t, timer, Infinity)$

$NCS(t) \triangleq \land GoFromTo(t, \text{"ncs"}, \text{"a"})$
$\qquad\qquad \land \text{UNCHANGED } \langle x, now, lbTimer, ubTimer \rangle$

$StmtA(t) \triangleq$ Defined on page 14.

$StmtB(t) \triangleq$ Defined on page 14.

$StmtC(t) \triangleq$ Defined on page 14.

$CS(t) \triangleq \land GoFromTo(t, \text{"cs"}, \text{"d"})$
$\qquad\qquad \land \text{UNCHANGED } \langle x, now, lbTimer, ubTimer \rangle$

$StmtD(t) \triangleq \land GoFromTo(t, \text{"d"}, \text{"ncs"})$
$\qquad\qquad \land x' = NotAThread$
$\qquad\qquad \land \text{UNCHANGED } \langle now, lbTimer, ubTimer \rangle$

$Tick \triangleq$ Defined on page 15.

$TNext(t) \triangleq NCS(t) \lor StmtA(t) \lor StmtB(t) \lor StmtC(t) \lor CS(t) \lor StmtD(t)$

$Next \triangleq Tick \lor (\exists t \in Thread : TNext(t))$

$SafetySpec \triangleq Init \land \Box[Next]_{vars}$

THEOREM $SafetySpec \Rightarrow \Box MutualExclusion$

$Liveness \triangleq$
$\quad \land \forall t \in Thread : \mathrm{WF}_{vars}(StmtA(t) \lor StmtB(t) \lor StmtC(t) \lor StmtD(t))$
$\quad \land \forall r \in Real : \Diamond(now > r)$

$FSpec1 \triangleq SafetySpec \land Liveness$

$Progress \triangleq$
$\quad (\exists t \in Thread : pc[t] \in \{\text{"a"}, \text{"b"}, \text{"c"}\}) \leadsto (\exists t \in Thread : pc[t] = \text{"cs"})$

THEOREM $FSpec1 \Rightarrow Progress$

Figure 3: Module *Fischer1*, containing the specification of our first version of Fischer's algorithm.

unchanged. Therefore, it is defined by

$$
\begin{array}{lll}
StmtA(t) & \triangleq & \land\ x = NotAThread \\
 & & \land\ GoFromTo(t, \text{“a”}, \text{“b”}) \\
 & & \land\ SetTimer(t, ubTimer, Delta) \\
 & & \land\ \text{UNCHANGED}\ \langle x, now, lbTimer \rangle
\end{array}
$$

Setting an upper-bound timer to a real value τ prevents *now* from advancing more than τ seconds. So, an upper-bound timer must be reset to *Infinity* to turn it off when it is not in use. A $StmtB(t)$ step must therefore reset $ubTimer[t]$, so $StmtB(t)$ needs the conjunct $ResetUBTimer(t, ubTimer)$. All other actions corresponding to program statements must leave $ubTimer[t]$ unchanged.

The algorithm has one lower-bound constraint—that a thread may not execute statement c until at least ϵ seconds after it executes statement b. We use $lbTimer[t]$ to assert that $StmtC(t)$ may not be executed until at least *Epsilon* seconds after control reaches c. Since a $StmtB(t)$ step sets $pc[t]$ to “c”, action $StmtB(t)$ must set $lbTimer[t]$. Remembering that $StmtB(t)$ must also reset $ubTimer[t]$, we have:

$$
\begin{array}{lll}
StmtB(t) & \triangleq & \land\ GoFromTo(t, \text{“b”}, \text{“c”}) \\
 & & \land\ x' = t \\
 & & \land\ RestUBTimer(t, ubTimer) \\
 & & \land\ SetTimer(t, lbTimer, Epsilon) \\
 & & \land\ \text{UNCHANGED}\ now
\end{array}
$$

We express the requirement that a $StmtC(t)$ step may occur only after $lbTimer[t]$ has timed out by conjoining to $StmtC(t)$ the enabling condition $TimedOut(t, lbTimer)$. A lower-bound timer is turned off when it times out and reaches 0, so there is no need for $StmtC(t)$ to change *lbTimer*. The definition is therefore

$$
\begin{array}{lll}
StmtC(t) & \triangleq & \land\ At(t, \text{“c”}) \\
 & & \land\ TimedOut(t, lbTimer) \\
 & & \land\ \text{IF}\ \ x \neq t\ \ \text{THEN}\ \ GoTo(t, \text{“a”}) \\
 & & \qquad\qquad\quad\ \text{ELSE}\ \ GoTo(t, \text{“cs”}) \\
 & & \land\ \text{UNCHANGED}\ \langle x, now, lbTimer, ubTimer \rangle
\end{array}
$$

Actions $NCS(t)$, $CS(t)$, and $StmtD(t)$ are the same as in the untimed specification, except that they also leave *now*, *lbTimer*, and *ubTimer* unchanged.

A *Tick* step increments *now* by some positive number d, which must be less than the value of all upper-bound timers. It decreases all timers

by d, except that lower-bound timers must stop at 0. So, a lower-bound timer is set to 0 if its value is less than d. (An upper-bound timer cannot become negative because d must be less than its current value.) The *Tick* action leaves the "ordinary" variables x and pc unchanged. Its definition is therefore

$$
\begin{array}{l}
\mathit{Tick} \;\triangleq \\
\quad \exists\, d \in \{r \in \mathit{Real} : r > 0\} : \\
\qquad \land\; \forall\, t \in \mathit{Thread} : \mathit{ubTimer}[t] > d \\
\qquad \land\; \mathit{now}' = \mathit{now} + d \\
\qquad \land\; \mathit{ubTimer}' = [t \in \mathit{Thread} \mapsto \\
\qquad\qquad\qquad \text{IF } \mathit{ubTimer}[t] = \mathit{Infinity} \text{ THEN } \mathit{Infinity} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } \mathit{ubTimer}[t] - d] \\
\qquad \land\; \mathit{lbTimer}' \;= [t \in \mathit{Thread} \mapsto \mathit{Max}(0,\, \mathit{lbTimer}[t] - d)] \\
\qquad \land\; \text{UNCHANGED } \langle x,\, pc \rangle
\end{array}
$$

This completes the definition of *Next*, the algorithm's next-state action. The safety part of the specification is the formula

$$\mathit{SafetySpec} \;\triangleq\; \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$$

That the algorithm implements mutual exclusion is expressed formally by the assertion that the predicate *MutualExclusion* is true throughout every behavior satisfying this specification. In TLA$^+$, this assertion is written

$$\text{THEOREM } \mathit{SafetySpec} \Rightarrow \Box \mathit{MutualExclusion}$$

In addition to satisfying mutual exclusion, we want an algorithm to ensure progress. For the Fischer algorithm, progress means that, if some thread is waiting to enter its critical section, then some thread (not necessarily the same one) will eventually enter. In the standard terminology of mutual exclusion, Fischer's algorithm is deadlock free but not starvation free. A thread is waiting to enter its critical section iff control is at statement a, b, or c. The progress condition can therefore be expressed in temporal logic as

$$
\begin{array}{l}
\mathit{Progress} \;\triangleq\; (\exists t \in \mathit{Thread} : pc[t] \in \{\text{“a”}, \text{“b”}, \text{“c”}\}) \\
\qquad\qquad\qquad\qquad \leadsto (\exists t \in \mathit{Thread} : pc[t] = \text{“cs”})
\end{array}
$$

where $P \leadsto Q$ (read P *leads to* Q) asserts that, if P ever becomes true, then Q will be true then or at some later point in the execution. Observe that this condition is satisfied if some thread ever enters its critical section and remains there forever.

To ensure that the algorithm makes progress, we must conjoin some liveness assumption to the safety specification. Our first specification does this by placing weak fairness assumptions on program statements a, c, and d. (The upper-bound constraint implies that b must be executed within δ seconds of when control reaches it, so we don't need a fairness assumption on statement b.) Weak fairness of an action A means that, if A remains continuously enabled, then an A step must eventually occur. It is usually expressed in TLA by the formula $\mathrm{WF}_v(A)$, where v is the tuple of all specification variables. To anyone familiar with reasoning about fairness, it is clear that the conjunction of weak fairness of each of the three actions $StmtA(t)$, $StmtC(t)$ and $StmtD(t)$ is equivalent to weak fairness of their disjunction. Hence, the fairness assumption can be expressed as

$$\forall t \in Thread : \mathrm{WF}_{vars}(StmtA(t) \lor StmtC(t) \lor StmtD(t))$$

We must also require *now* to keep advancing. (Otherwise, the lower-bound constraint could keep a $StmtC(t)$ step from ever happening.) This assumption can be expressed as:[6]

$$\forall r \in Real : \Diamond(now > r)$$

Deadlock freedom of Fischer's algorithm is shown by proving that the conjunction of *SafetySpec* and these two liveness assumptions imply formula *Progress*.

The definition of *FSpec*1, the specification of Fischer's algorithm with these liveness assumptions, and the statement of its correctness appear in module *Fischer*1 of Figure 3. As indicated, some definitions that appear in the text are omitted.

Observe that the variable *now* appears only in the conjunct of *Tick* that increments it and in UNCHANGED conjuncts. It acts as a history variable, recording the passage of time but not affecting the values of other variables. Eliminating *now* would not materially change the specification. The ability to eliminate *now* is important for model checking, but there is no reason to remove it from the specification now.

Although I have written the specification in TLA$^+$, it could be written in other languages. Many languages might have difficulty expressing a *Tick* action that allows *now* to be incremented by an arbitrary real number.

[6] Had we restricted *now* to have integral values, then we could instead have required strong fairness of the *Tick* action. However, since *now* can assume arbitrary real values, its value can remain bounded despite an infinite sequence of *Tick* steps. Therefore, fairness of the *Tick* action does not assure that *now* increases without bound.

However, as mentioned above, little is lost by letting *now* assume discrete values, in which case *Tick* can just increment *now* by 1. The specification can then be written in any language with global variables that can describe multithreaded algorithms.

2.5 Specification *FSpec*2: Progress Through Timing Bounds

Specification *FSpec*1 of module *Fischer*1 uses fairness assumptions on the execution of statements a, c, and d to ensure progress. We now write a specification *FSpec*2 that ensures progress by placing upper bounds on the execution times of these statements. More precisely, we place an upper bound on the length of time that the action representing the statement's execution can be enabled without its "being executed". For statements c and d, that is simply the maximum length of time between control reaching the statement and the statement being executed. For statement a, it means that, when control in thread t is at a, it is an upper bound on the length of time that x can continuously equal *NotAThread* without a $StmtA(t)$ step occurring.

For convenience, we use the same upper bound δ for statements a and d that we have already used for statement b. To avoid contradictory timing conditions, we must make the upper bound on the execution time for c larger than its lower bound ϵ, which is greater than or equal to δ. So we let the upper bound on the execution time of c be a new parameter γ, which we assume to be greater than ϵ.

We use the timers $ubTimer[t]$ to express these upper bounds. As in specification *FSpec*1, we could let *ubTimer* be set by the actions representing the individual program statements. However, this is more complicated than before because execution of statement d by one thread must set the upper-bound timer in any other thread that is waiting to execute statement a. Instead, to specify how an action of thread t changes the upper-bound timers, we conjoin an action expression to the entire next-state action of thread t. For uniformity, we describe the setting of the lower-bound timers the same way. The specification's next-state action is therefore

$$Next \triangleq Tick \lor (\exists t \in Thread : TNext(t) \land SetTimers(t))$$

where *Tick* is the same as in *FSpec*1; action $TNext(t)$ is almost the same as in the untimed specification; and $SetTimers(t)$ describes how an action of thread t changes *lbTimer* and *ubTimer*, as well as asserting that *now* is

left unchanged. The definition of $SetTimers(t)$ therefore has the form

$$SetTimers(t) \triangleq \land\ lbTimer' = \ldots \\ \land\ ubTimer' = \ldots \\ \land\ \text{UNCHANGED}\ now$$

An action of thread t changes only the t component of $lbTimer$, setting it to $Epsilon$ if the new value of $pc[t]$ is "c" and to $Infinity$ otherwise. So a $SetTimers(t)$ step should set $lbTimer$ to

$$[lbTimer\ \text{EXCEPT}\ ![t] = \text{IF}\ pc'[t] = \text{"c"}\ \text{THEN}\ Epsilon\ \text{ELSE}\ Infinity]$$

A $SetTimers(t)$ step should change the value of $ubTimer[s]$ as follows:

- If $s = t$, so this is a step that changes $pc[s]$, then $ubTimer'[s]$ should equal:
  - $Delta$ if the new value of $pc[s]$ is "b" or "d", or if it is "a" and the new value of x equals $NotAThread$.[7]
  - $Gamma$ if the new value of $pc[s]$ is "c".
  - $Infinity$ otherwise.
- If $s \neq t$, so $ubTimer[s]$ is a timer for a different thread's action, then $ubTimer'[s]$ should equal:
  - $Delta$ if $pc[s] = \text{"a"}$ and the action changes the value of x to $NotAThread$.
  - $Infinity$ if $pc[s] = \text{"a"}$ and the action changes the value of x to a thread identifier.
  - $ubTimer[s]$ (the current value) otherwise.

The complete definition of $SetTimers(t)$ is in Figure 4.

The next-state action $TNext(t)$ of thread t is the same as for the untimed algorithm, except that action $StmtC(t)$ has the additional conjunct $TimedOut(t, lbTimer)$ to describe the lower-bound constraint on its execution. The complete specification is in module $Fischer2$, shown in Figure 5. The module's theorem asserts that the specification satisfies mutual exclusion. The assertion and proof that it satisfies a real-time progress condition appears in Section 3.3 below.

[7] In the latter case, the step does not change x, so its new and old values are actually the same.

$$
\begin{array}{l}
\mathit{SetTimers}(t) \;\triangleq \\
\quad \land\ \mathit{lbTimer}' = [\mathit{lbTimer}\ \text{EXCEPT}\ ![t] = \text{IF}\ \mathit{pc}'[t] = \text{“c”}\ \text{THEN}\ \mathit{Epsilon} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \text{ELSE}\ \ 0] \\
\quad \land\ \mathit{ubTimer}' = [s \in \mathit{Thread} \mapsto \\
\qquad\qquad\qquad \text{IF}\ s = t \\
\qquad\qquad\qquad\quad \text{THEN IF}\ \lor\ \mathit{pc}'[s] \in \{\text{“b”}, \text{“d”}\} \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ (\mathit{pc}'[s] = \text{“a”}) \land (x' = \mathit{NotAThread}) \\
\qquad\qquad\qquad\qquad\qquad \text{THEN}\ \mathit{Delta} \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ \ \text{IF}\ \mathit{pc}'[s] = \text{“c”}\ \text{THEN}\ \mathit{Gamma} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE}\ \ \mathit{Infinity} \\
\qquad\qquad\qquad\quad \text{ELSE}\ \ \text{IF}\ \land\ \mathit{pc}[s] = \text{“a”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ (x' = \mathit{NotAThread}) \neq (x = \mathit{NotAThread}) \\
\qquad\qquad\qquad\qquad\qquad \text{THEN IF}\ x' = \mathit{NotAThread}\ \text{THEN}\ \mathit{Delta} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \text{ELSE}\ \ \mathit{Infinity} \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ \ \mathit{ubTimer}[s]] \\
\quad \land\ \text{UNCHANGED}\ \mathit{now}
\end{array}
$$

Figure 4: The action describing how a step of thread t changes *lbTimer*, *ubTimer*, and *now*.

This way of writing the specification, conjoining the action expression $\mathit{SetTimers}(t)$ to the next-state action $\mathit{TNext}(t)$ of thread t, is quite natural with TLA. It would be difficult or impossible to do it with most programming-language based specification methods. In those methods, one would have to add the timer-setting actions to the individual statements. While this makes the specification perhaps less elegant, it presents no fundamental difficulties.

Just as with specification *FSpec*1, we can eliminate the variable *now* from *FSpec*2 without affecting the algorithm. However, we will use *now* in the next section for expressing the real-time progress property that corresponds to the liveness property *Progress* of module *Fischer*1.

3 Proving Correctness of Fischer's Algorithm

Just as an explicit-time specification can use existing languages, its correctness can be proved with existing proof methods. Such methods have been around for almost three decades and should by now be well-known. So, I will just state what are essentially the fundamental lemmas necessary to prove correctness of Fischer's algorithm. The actual proofs are left to the motivated reader.

```
---------------------------- MODULE Fischer2 ----------------------------
EXTENDS FischerPreface
CONSTANT Gamma
ASSUME Epsilon < Gamma
-------------------------------------------------------------------------
NCS(t) ≜ ∧ GoFromTo(t, "ncs", "a")
         ∧ UNCHANGED x

StmtA(t) ≜ ∧ x = NotAThread
           ∧ GoFromTo(t, "a", "b")
           ∧ UNCHANGED x

StmtB(t) ≜ ∧ GoFromTo(t, "b", "c")
           ∧ x' = t

StmtC(t) ≜ ∧ At(t, "c")
           ∧ TimedOut(t, lbTimer)
           ∧ IF x ≠ t THEN GoTo(t, "a")
                      ELSE GoTo(t, "cs")
           ∧ UNCHANGED x

CS(t) ≜ ∧ GoFromTo(t, "cs", "d")
        ∧ UNCHANGED x

StmtD(t) ≜ ∧ GoFromTo(t, "d", "ncs")
           ∧ x' = NotAThread

Tick ≜   Same definition as in Module Fischer1 (page 13).

SetTimers(t) ≜   Defined on page 19

TNext(t) ≜ NCS(t) ∨ StmtA(t) ∨ StmtB(t) ∨ StmtC(t) ∨ CS(t) ∨ StmtD(t)

Next ≜ Tick ∨ (∃ t ∈ Thread : TNext(t) ∧ SetTimers(t))
-------------------------------------------------------------------------
FSpec2 ≜ Init ∧ □[Next]_vars

THEOREM FSpec2 ⇒ □MutualExclusion
=========================================================================
```

Figure 5: Module *Fischer*2, containing the specification *FSpec*2 of our second version of Fischer's algorithm.

3.1 Proof of Mutual Exclusion

In TLA, one proves that an algorithm with specification *Spec* satisfies a property F by proving the theorem $Spec \Rightarrow F$. Often, F has the same form as *Spec*. In that case, one usually says that the algorithm implements the high-level specification F, or that it implements the abstract model/algorithm described by F. To show that the Fischer algorithm satisfies mutual exclusion, we take F to be the invariance property $\Box MutualExclusion$.

The basic method of proving this invariance property was first published by Ashcroft in 1975 [5]. It involves finding an *inductive invariant* that implies *MutualExclusion*. An inductive invariant for a specification $Init \land \Box[Next]_{vars}$ is a state predicate *Inv* satisfying

I1. $Init \Rightarrow Inv$

I2. $Inv \land Next \Rightarrow Inv'$

where Inv' is obtained from *Inv* by priming all variable occurrences.

The differences between our two versions of Fischer's algorithm do not affect the basic mutual exclusion protocol, and the same inductive invariant *Inv* proves mutual exclusion for both of them. That invariant is defined in Figure 6. It contains the conjunct *TypeOK*, which is a simple type-correctness invariant asserting that each variable is an element of the appropriate set. For example, it asserts that *ubTimer* is a function from *Thread* to $Real \cup \{Infinity\}$. In a typed formalism, *TypeOK* would be implicit in the type declarations, and its invariance would be shown by type checking.[8]

We prove mutual exclusion by proving $I1$ and $I2$, which imply that *Inv* is always true, and then proving $Inv \Rightarrow MutualExclusion$, which shows that *MutualExclusion* is always true. This is all straightforward. In other formalisms, I2 is replaced by some verification condition. For example, in the Owicki-Gries method [35], *Inv* is written as a program annotation, and I2 is expressed as "sequential correctness" and "interference freedom" conditions.

3.2 Proof of Eventual Progress

We want to prove that the specification $SafetySpec \land Liveness$ of Fischer's algorithm in module *Fischer*1 satisfies property *Progress*. This property has the form $P \leadsto Q$. The basic method of proving such a property was

[8] Simple type checking proves a type-correctness invariant only if all operators are "total". For example, if the tail of an empty list is undefined, then proving a type-correctness invariant requires showing that the algorithm never takes the tail of an empty list.

$$
\begin{array}{l}
Inv \triangleq \\
\quad \land\ TypeOK \\
\quad \land\ \forall t \in Thread : \\
\qquad \land\ now \leq ubTimer[t] \\
\qquad \land\ (pc[t] = \text{"b"}) \Rightarrow (ubTimer[t] < now + Epsilon) \\
\qquad \land\ (pc[t] = \text{"c"}) \Rightarrow \\
\qquad\qquad \forall s \in Thread : (x = t) \land (pc[s] = \text{"b"}) \\
\qquad\qquad\qquad\qquad\qquad \Rightarrow (lbTimer[t] > ubTimer[s]) \\
\qquad \land\ (pc[t] \in \{\text{"cs"}, \text{"d"}\}) \Rightarrow (x = t) \land (\forall s \in Thread : pc[s] \neq \text{"b"})
\end{array}
$$

Figure 6: The inductive invariant for the proof of mutual exclusion, where *TypeOK* is a simple type-correctness invariant.

explained two decades ago [36]. One decomposes the proof of $P \leadsto Q$ into the proof of simpler $\leadsto$ formulas using a *proof lattice*. A lattice for the proof of $P \leadsto Q$ is an acyclic directed graph, whose nodes are temporal formulas, having P as the only source and Q as the only sink, and such that every path from P to Q is finite. A node G is a *successor* of a node F in the lattice iff there is an arc from F to G. A non-sink node F represents the assertion that F leads to the disjunction of its successor nodes—that is, the formula $F \leadsto (\exists i \in S : G_i)$, where $\{G_i : i \in S\}$ is the set of successors of F. The conjunction of the formulas represented by all the non-sink nodes of the proof lattice implies $P \leadsto Q$.

Formula $P \leadsto Q$ asserts that, if P ever becomes true, then Q is true then or at some later point in the execution. This is usually proved by proving $(P \land \Box\neg Q) \leadsto \text{FALSE}$, which asserts that P true and Q never again true leads to a contradiction.

A proof lattice for proving eventual progress of Fischer's algorithm appears in Figure 7. For compactness, it uses the following definitions. State predicates A, B, C, and D assert that control in some thread is at the corresponding control point—for example:

$$B \triangleq \exists t \in Thread : pc[t] = \text{"b"}$$

State predicate *Crit* asserts that some thread is in its critical section:

$$Crit \triangleq \exists t \in Thread : pc[t] = \text{"cs"}$$

State predicates ABC, $DCrit$, $BDCrit$, and X are defined by:

$$
\begin{array}{llll}
ABC \triangleq A \lor B \lor C & \qquad & BDCrit \triangleq B \lor D \lor Crit \\
DCrit \triangleq D \lor Crit & \qquad & X \triangleq x \in Thread
\end{array}
$$

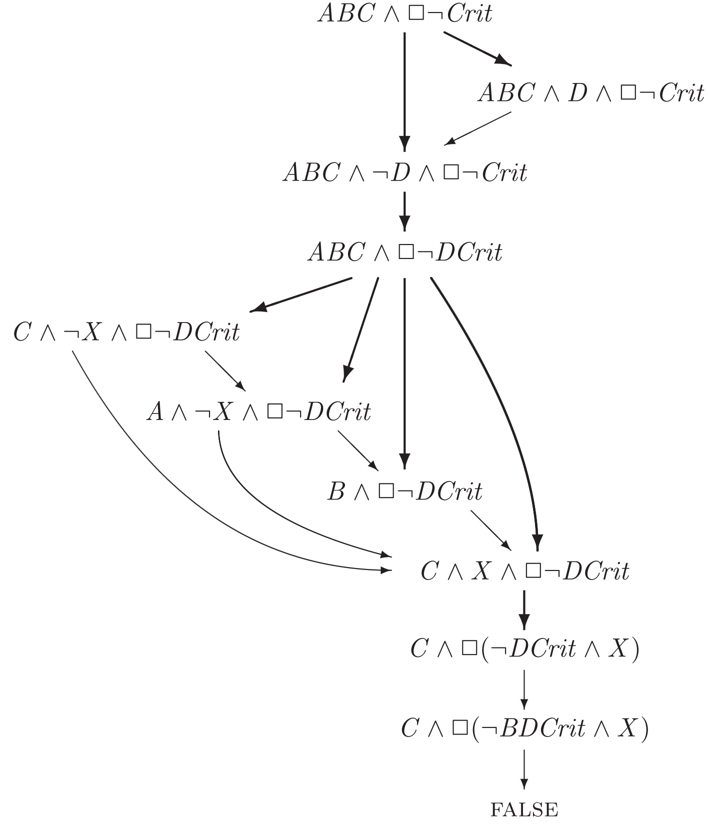


Figure 7: A proof lattice for $ABC \land \Box\neg Crit \leadsto$ FALSE, used in the proof of *SafetySpec* $\land$ *Liveness* $\Rightarrow$ *Progress*.

Hence, *Progress* equals $ABC \leadsto Crit$, which can be proved by proving

$$ABC \land \Box\neg Crit \leadsto \text{FALSE}$$

Figure 7 is a proof lattice for the latter formula. Thick arrows from a node indicate that the node implies the disjunction of its successors. (Obviously, $F \Rightarrow G$ implies $F \leadsto G$, for any formulas F and G.)

Intuitive proofs that *SafetySpec* $\land$ *Liveness* implies each of the ten formulas of this proof lattice are straightforward. Those proofs can be formalized with the proof rules of TLA [23]. They require the easily proved lemma that the following state predicate is an invariant of the specification.

$$LInv \triangleq (x \in Thread) \Rightarrow (pc[x] \in \{\text{“c”}, \text{“cs”}, \text{“d”}\})$$

Essentially the same proof should be possible in any formalism that can prove liveness properties of multithreaded algorithms.

The proof lattice of Figure 7 is fine grained, in the sense that when any non-sink node formula becomes true, it remains true until one of its successors becomes true. This implies that each thin arrow indicates a change effected by a single step. For example, once $C \wedge \neg X \wedge \Box\neg \mathit{DCrit}$ becomes true, it remains true until one of its two successors becomes true. The thin arrow from it to $C \wedge X \wedge \Box\neg \mathit{DCrit}$ indicates a state change that is caused by a $\mathit{StmtB}(t)$ step, for some thread t.

3.3 Proof of Real-Time Progress

An eventual progress property $P \leadsto Q$ asserts that, whenever P is true, Q will eventually become true. In the real-time version of this property, *eventually* is replaced by *within* Ω *seconds*, for some Ω. Proving this property can be reduced to proving an invariant by introducing a *history variable* [1]. A history variable is an auxiliary variable, meaning that adding it does not change the behavior of the other variables. Formally, *HSpec* is said to be a specification obtained by adding an auxiliary variable h to a specification *Spec* if hiding h in *HSpec* produces a specification that is equivalent to *Spec*. In TLA, this condition is expressed as

$$\text{AV.}\ \ \mathit{Spec} \equiv \exists\, h : \mathit{HSpec}$$

A history variable h is added to a specification *Spec* as follows. Suppose *Spec* equals $\mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{vars}}$, where *vars* is the tuple of all the specifications variable, and h is not one of those variables. Condition AV is then satisfied if *HSpec* equals $\mathit{HInit} \wedge \Box[\mathit{HNext}]_{\langle \mathit{vars},\, h\rangle}$, where

$$\mathit{HInit} \triangleq \mathit{Init} \wedge (h = \mathit{hIni}) \qquad\qquad \mathit{HNext} \triangleq \mathit{Next} \wedge (h' = \mathit{hNew})$$

for some *hIni* and *hNew* such that h does not occur in *hIni* and h' does not occur in *hNew*. Other formalisms also have rules for introducing history variables [35, 40].[9]

Suppose we introduce a history variable h that is set to *now* when $P \wedge \neg Q$ becomes true, is set to ∞ when Q is true, and otherwise remains unchanged. Then Q must become true within Ω seconds of when P does iff $\mathit{now} - \Omega$ is always less than h. Hence, proving the real-time progress property is then

[9]Most formalisms cannot express condition AV; they must instead define semantically what it means to add an auxiliary variable.

MODULE *HFischer*2

EXTENDS *Fischer*2
VARIABLE h

$$
\begin{aligned}
HInit \;\triangleq\;& \land\ Init \\
& \land\ h = Infinity
\end{aligned}
$$

$$
\begin{aligned}
HNext \;\triangleq\;& \land\ Next \\
& \land\ h' = \text{IF } \exists\, t \in Thread : pc'[t] = \text{“cs”} \\
& \qquad\quad \text{THEN } Infinity \\
& \qquad\quad \text{ELSE IF } \land\ h = Infinity \\
& \qquad\qquad\qquad\quad \land\ \exists\, t \in Thread : pc'[t] \in \{\text{“a”}, \text{“b”}, \text{“c”}\} \\
& \qquad\qquad\qquad \text{THEN } now \\
& \qquad\qquad\qquad \text{ELSE } h
\end{aligned}
$$

$$HFSpec2 \;\triangleq\; HInit \land \Box[HNext]_{\langle vars,\, h\rangle}$$

THEOREM $HFSpec2 \Rightarrow \Box(now - (4 * Delta + 2 * Gamma - Epsilon) < h)$

Figure 8: The specification *HFSpec*2, obtained by adding the history variable h to specification *Spec* of module *Fischer*2.

reduced to proving the invariance of $now - \Omega < h$. This particular history variable h is defined as indicated above by letting

$$
\begin{aligned}
hIni \;&\triangleq\; \text{IF } P \land \neg Q \text{ THEN } now \text{ ELSE } \infty \\
hNew \;&\triangleq\; \text{IF } Q' \text{ THEN } \infty \\
&\qquad\qquad\;\; \text{ELSE IF } P' \land (\neg P \lor Q) \text{ THEN } now \text{ ELSE } h
\end{aligned}
$$

A little thought reveals that *hNew* can also be defined by:

$$
\begin{aligned}
hNew \;&\triangleq\; \text{IF } Q' \text{ THEN } \infty \\
&\qquad\qquad\;\; \text{ELSE IF } P' \land (h = \infty) \text{ THEN } now \text{ ELSE } h
\end{aligned}
$$

For the Fischer algorithm, P asserts that some thread is at statement a, b, or c, and Q asserts that some thread is in its critical section. Figure 8 contains the TLA$^+$ specification *HFSpec*2 obtained by adding the history variable h to specification *FSpec*2 of Fischer's algorithm.

Proving real-time progress requires showing that $now - \Omega < h$ is an invariant of *HFSpec*2, for a suitable constant Ω. Although this is a standard invariance problem, few people have experience finding Ω and constructing the necessary inductive invariant. So, I will show how it is done.

Specification *FSpec*2 satisfies essentially the same progress property as *FSpec*1, except with "eventually" replaced by "within some period of time". The proof that it does is also essentially the same as for *FSpec*1, except with each eventuality assertion replaced by a real-time progress assertion. We obtain the proof of the real-time property from the proof of the eventuality property by converting the proof lattice of Figure 7 into the *timing graph* of Figure 9. A timing graph is a directed graph whose nodes are state predicates, labeled with certain timing information. To obtain this timing graph, we first modify the proof lattice as follows:

- We replace every formula of the form $\Box F$ with F. (A state predicate describes only the current state, not future behavior.)
- We make gray the sink and every state predicate that implies the disjunction of its successors—that is, the sources of the thick arrows in the lattice. (Any state satisfying such a grayed node satisfies one of its successors.)

We then transform this in the obvious way into a graph containing only the black nodes—that is, with an edge from node F to node G iff there is a path from F to G in the original graph with no intermediate black nodes. To keep the correspondence with the proof lattice clear, I have omitted this transformation in Figure 9.

Throughout an execution of the algorithm, a black predicate of Figure 9 is true iff $h < \infty$. Suppose that, as the algorithm executes, we move a token according to the following rules.

1. The token is initially off the graph.
2. When h is set to a finite value, the token is placed on some black predicate that is true.
3. As soon as one or more black successors (in the transformed graph) of a predicate containing the token becomes true, the token is moved to one of those successors.
4. When all black predicates become false (so h is changed to ∞), the token is removed from the graph.

Obviously, the token will be on the graph iff $h < \infty$. Because the original proof lattice was fine grained, a successor of a formula F in the timing graph must become true before F can become false. This ensures that the token will never be on a predicate that is false.

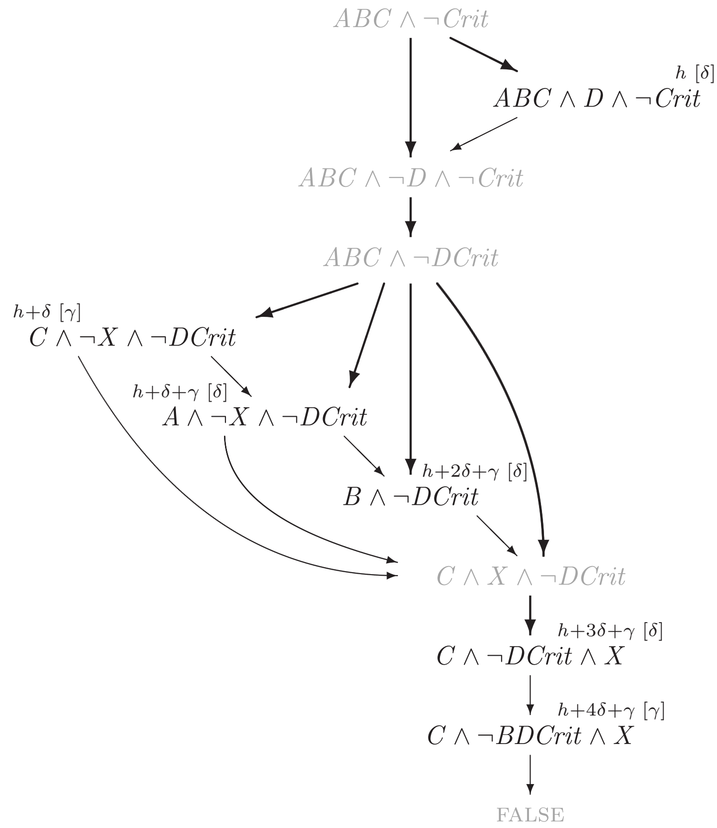


Figure 9: The timing graph for Fischer's algorithm obtained from the proof lattice of Figure 7.

The invariant $now - \Omega < h$ asserts that the token can remain on the graph for a period of at most Ω seconds. To calculate Ω, we simply have to know how long the token can remain on each black predicate. This length of time is indicated by the number (either δ or γ) in square brackets besides the node. It is obtained from the upper-bound constraints on the actions that must make the predicate false. When the token is on the graph, h equals the value *now* had when it was last moved onto the graph. From the numbers in brackets and the graph structure, it is easy to calculate, as a function of h, the greatest value that *now* can have when the token first reaches each node. Those values are also placed next to the nodes. The sum of the two numbers attached to a node is the largest value that *now* can have while the token is still on that node. The largest of those sums, in this case, $h + 4\delta + 2\gamma$, is the largest value *now* can have while the token is on the graph. Hence, $4\delta + 2\gamma$ is the desired value Ω.

We can now use this timing graph to construct the inductive invariant *HInv* used to prove the invariance of $now - 4\delta + 2\gamma < h$. The interesting conjunct of this invariant equals

$$(h = \infty) \lor TD(F_1) \lor \ldots \lor TD(F_k)$$

where $F_1, \ldots, F_k$ are the (black) nodes of the timing graph, and $TD(F)$ is constructed from F as follows. Suppose $h + \alpha\ [\beta]$ is the timing information associated with node F, asserting that $now < h + \alpha + \beta$ holds when the token is on that node. The state predicate $TD(F)$ essentially implies F and $now < h + \alpha + \beta$. It implies the bound on *now* by implying $now + ubTimer[t] \leq h + \alpha + \beta$, for some upper-bound timer $ubTimer[t]$. This implies the bound on *now* because *HInv* will also imply $0 < ubTimer[t]$.

For example, consider the predicate $A \land \neg X \land \neg DCrit$, which has the associated timing information $h + \delta + \gamma\ [\delta]$. The upper-bound timer used to imply $now < h+2\delta+\gamma$ is the one for the thread whose control is at statement a. (The existence of such a thread is asserted by the state predicate A.) We define $TCond(t, ctl, tau)$ to assert that thread t is at control point ctl, and its upper-bound timer will time out before time $h + tau$.

$$TCond(t, ctl, tau) \ \triangleq\ (pc[t] = ctl) \land (now + ubTimer[t] < h + tau)$$

The predicate $TD(A \land \neg X \land \neg DCrit)$ can then be written:

$$\begin{aligned} &\land \neg X \land \neg DCrit \\ &\land \exists t \in Thread : TCond(t, \text{“a”}, 2 * Delta + Gamma) \end{aligned}$$

The first conjunct does not assert A because A is implied by the second conjunct.

One slight complication is that the strict inequality of $TCond$ does not hold for the source node of the graph, since h is set to now and $ubTimer[t]$ is set to δ when thread t exits the critical section. Its condition is expressed with the operator

$$\begin{array}{l} TCondIni(t, ctl, tau) \triangleq \\ \quad (pc[t] = ctl) \land (now + ubTimer[t] \leq h + tau) \end{array}$$

The definition of $HInv$ appears in Figure 10. The fourth conjunct is the interesting one. The first two conjuncts are the invariants Inv and $LInv$, defined in Figure 6 and Section 3.2, respectively. The third conjunct asserts bounds on the values of upper-bound timers as a function of the threads' control states. These bounds hold because, if an upper-bound timer was last set to time out in τ seconds, its current value is at most $now + \tau$. The two penultimate disjuncts of the fourth conjunct, which correspond to the nodes $B \land \neg DCrit$ and $C \land \neg DCrit \land X$ of Figure 9, contain an additional conjunct asserting a property of the upper-bound timers of all threads with control at statement b.

The observant reader will have noted that the value of Ω derived from the timing diagram of Figure 9 is larger than the one that appears in the theorem of module $HFischer2$ in Figure 8. Fischer's algorithm actually satisfies a smaller bound on the waiting time than the one proved with the invariant $HInv$. The proof of the stronger bound requires a more complicated invariant; it is left as an exercise.

4 A Simple Distributed Algorithm

Because they are subtle and hard to debug, distributed algorithms are an important application domain for specification and verification. Such algorithms typically have features not present in the simple Fischer algorithm. These features include queues or sets of messages in transit—each message with a separate upper bound on its delivery time—and dynamically computed timeout delays. Uppaal [27] is the only real-time model checker I know of that can handle algorithms with such features. But Uppaal's modeling language lacks the high-level data structures of TLA$^+$, so it must use a lower-level encoding of these algorithms. Language limitations can be a significant barrier to the practical verification of complex distributed algorithms.

We now consider an algorithm that, while very simple, exhibits the interesting features of more complicated distributed algorithms. It is inspired by a classic algorithm of Radia Perlman [37]. The original algorithm constructs

$$
\begin{array}{l}
HInv \triangleq \\
\quad \land Inv \\
\quad \land LInv \\
\quad \land \forall t \in Thread : \\
\qquad \land (pc[t] \in \{\text{"b"}, \text{"d"}\}) \Rightarrow (ubTimer[t] \leq Delta) \\
\qquad \land (pc[t] = \text{"a"}) \land (x = NotAThread) \Rightarrow (ubTimer[t] \leq Delta) \\
\qquad \land (pc[t] = \text{"c"}) \Rightarrow (ubTimer[t] \leq Gamma) \\
\quad \land \lor \land h = Infinity \\
\qquad \lor \land ABC \land \neg Crit \\
\qquad\quad \land \exists t \in Thread : TCondIni(t, \text{"d"}, Delta) \\
\qquad \lor \land \neg X \land \neg DCrit \\
\qquad\quad \land \exists t \in Thread : TCond(t, \text{"c"}, Delta + Gamma) \\
\qquad \lor \land \neg X \land \neg DCrit \\
\qquad\quad \land \exists t \in Thread : TCond(t, \text{"a"}, 2 * Delta + Gamma) \\
\qquad \lor \land \neg DCrit \\
\qquad\quad \land \forall t \in Thread : (pc[t] = \text{"b"}) \Rightarrow TCond(t, \text{"b"}, 4 * Delta + Gamma) \\
\qquad\quad \land \exists t \in Thread : TCond(t, \text{"b"}, 3 * Delta + Gamma) \\
\qquad \lor \land X \land \neg DCrit \quad \text{This conjunct and } LInv \text{ imply } C \\
\qquad\quad \land \forall t \in Thread : (pc[t] = \text{"b"}) \Rightarrow TCond(t, \text{"b"}, 4 * Delta + Gamma) \\
\qquad\quad \land \exists t \in Thread : TCond(t, \text{"b"}, 4 * Delta + Gamma) \\
\qquad \lor \land \neg DCrit \land X \land \neg B \\
\qquad\quad \land TCond(x, \text{"c"}, 4 * Delta + 2 * Gamma)
\end{array}
$$

Figure 10: The inductive invariant *HInv* for proving real-time progress of Fischer's algorithm.

a spanning tree and maintains that tree by having the root periodically propagate an "I am alive" message down it. A new tree is constructed if a failure caused some node to time out before receiving the message.

Our simple algorithm assumes an arbitrary network of nodes. Each node can send messages to its neighbors. The network need not be connected. Define the *leader* of a connected component to be the lowest-numbered node in the component. The goal of the algorithm is for each node n to learn its leader—that is, the leader of its connected component.

As in Perlman's algorithm, the leader of a component maintains its leadership by periodically sending messages that are forwarded to all nodes in its component. A message contains a *hops* field, indicating how many times it has been forwarded. A node uses that field to determine its distance to the leader. (The distance between two nodes is the number of links in the shortest path joining them.) A node forwards messages that have reached it via a shortest path from the leader, and it ignores all other messages.

After sending a message, the leader sets a timer that will "awaken" it to send the next message. A node that receives a message from the leader sets its timer to awaken it if the leader's next message does not arrive when it should. When a node is awakened by the timeout, it assumes itself to be the leader and sends the appropriate message to its neighbors. Initially and after failure of a node or communication link, nodes can have a mistaken idea of who their leaders are. However, within a fixed period of time, every non-failed node learns who its leader is.

A node that believes itself to be the leader sets its timer to awaken it *Period* seconds after sending a message. We assume that the node can be awakened up to *TODelay* seconds after the timeout. (By letting *Period* be the minimum timeout interval, this models both delay in reacting to a timeout and variation in the running rate of physical timers.) We also assume that a message is received at most *MsgDelay* seconds after it is sent. A simple calculation shows that the algorithm achieves stability if, upon receiving a message from its leader, a node n sets its timer to time out no sooner than $\mathit{Period} + \mathit{TODelay} + \mathit{dist}[n] * \mathit{MsgDelay}$ seconds in the future, where $\mathit{dist}[n]$ is the distance from n to its leader.

Correctness of this algorithm means that if no failure or repair has occurred for a sufficiently long period of time, then every node knows its leader. Stating this condition formally requires adding a history variable to record the time of the last failure or repair. To simplify the algorithm's description and the statement of its correctness, we assume that nodes do not fail. Correctness then means that, by a certain time, every node n knows its leader.

For simplicity, the TLA$^+$ specification assumes that there are N nodes, numbered from 1 to N. The set *Node* of node numbers equals $1 \dots N$. The network topology is described by a constant parameter *Nbrs*, where *Nbrs*(n) is the set of neighbors of node n—that is, the set of nodes with a link to n.

The specification *LSpec* is defined in module *Leader* of Figure 11, on pages 33–34. A node n maintains the following variables:

ldr$[n]$ The node that n believes to be its leader.

dist$[n]$ What n believes to be its distance to *ldr*$[n]$.

timer$[n]$ A countdown timer for node n's timeout action. To further simply the specification, we use this timer to express both upper- and lower-bound constraints by allowing the action to occur only after *timer*$[n]$ becomes negative and requiring it to occur before *timer*$[n]$ reaches $-TODelay$.

The variable *msgs* represents the messages in transit. A message has the following fields:

src The sender.

dest The destination node.

ldr The leader that originated the message.

hops The number of times the message has been forwarded.

rcvTimer A countdown timer used to express the upper-bound constraint on message-delivery time.

There could be multiple copies of the same message in transit at the same time. The value of *msgs* is therefore a bag. A bag (also called a multiset) is like a set, except that it can contain more than one copy of an element. The following operators on bags are defined in the standard *Bags* module:

BagToSet(B) The set of distinct elements in bag B.

SetToBag(S) A bag containing one copy of each element in set S.

$B_1 \oplus B_2$ The union of bags B_1 and B_2.

$B_1 \ominus B_2$ Bag B_1 with elements of bag B_2 removed, one copy of an element being removed from B_1 for every copy of the same element in B_2.

```
─────────────────────── MODULE Leader ───────────────────────
EXTENDS Reals, Bags

CONSTANTS N, Nbrs(_), MsgDelay, TODelay, Period
Node ≜ 1 .. N

VARIABLES ldr, dist, timer, msgs, now

Init ≜ ∧ ldr   = [n ∈ Node ↦ n]
       ∧ dist  = [n ∈ Node ↦ 0]
       ∧ timer = [n ∈ Node ↦ Period]
       ∧ msgs  = EmptyBag
       ∧ now   = 0
──────────────────────────────────────────────────────────────
MsgsSent(n, S) ≜ SetToBag([src : {n},  dest : S,  ldr : {ldr′[n]},
                           hops : {dist′[n]},  rcvTimer : {MsgDelay}])

TimeOut(n) ≜ ∧ timer[n] < 0
             ∧ ldr′   = [ldr  EXCEPT ![n] = n]
             ∧ dist′  = [dist EXCEPT ![n] = 0]
             ∧ msgs′  = msgs ⊕ MsgsSent(n, Nbrs(n))
             ∧ timer′ = [timer EXCEPT ![n] = Period]
             ∧ UNCHANGED now

RcvMsg(n) ≜
  ∧ ∃ ms ∈ BagToSet(msgs) :
       ∧ ms.dest = n
       ∧ IF  ∨ ms.ldr  < ldr[n]
             ∨ ∧ ms.ldr = ldr[n]
               ∧ ms.hops + 1 ≤ dist[n]
           THEN ∧ ldr′   = [ldr EXCEPT ![n] = ms.ldr]
                ∧ dist′  = [dist EXCEPT ![n] = ms.hops + 1]
                ∧ msgs′  = (msgs ⊖ SetToBag({ms}))
                                ⊕  MsgsSent(n, Nbrs(n) \ {ms.src})
                ∧ timer′ =
                     [timer EXCEPT ![n] = Period + TODelay +
                                          (ms.hops + 1) ∗ MsgDelay]
           ELSE ∧ msgs′ = msgs ⊖ SetToBag({ms})
                ∧ UNCHANGED ⟨ldr, dist, timer⟩
  ∧ UNCHANGED now
```

Figure 11a: Module *Leader* (beginning).

$$
\begin{aligned}
Tick \triangleq\ & \exists\, d \in \{r \in Real : r > 0\} : \\
& \quad \land \forall\, n \in Node : timer[n] + TODelay \geq d \\
& \quad \land \forall\, ms \in BagToSet(msgs) : ms.rcvTimer \geq d \\
& \quad \land now' = now + d \\
& \quad \land timer' = [n \in Node \mapsto timer[n] - d] \\
& \quad \land msgs' = \text{LET } Updated(ms) \triangleq \\
& \qquad\qquad\qquad [ms \text{ EXCEPT } !.rcvTimer = ms.rcvTimer - d] \\
& \qquad\qquad \text{IN } BagOfAll(Updated, msgs) \\
& \quad \land \text{UNCHANGED } \langle ldr, dist \rangle
\end{aligned}
$$

$$Next \triangleq (\exists\, n \in Node : TimeOut(n) \lor RcvMsg(n)) \lor Tick$$

$$LSpec \triangleq Init \land \Box[Next]_{\langle ldr,\, dist,\, msgs,\, timer,\, now \rangle}$$

We now state the assumptions about the constants, and define the predicate *Correctness* whose invariance asserts correctness of the algorithm.

$$
\begin{aligned}
\text{ASSUME}\ & \land N \in Nat \\
& \land \forall\, n \in Node : \land Nbrs(n) \subseteq Node \\
& \qquad\qquad\qquad\quad \land \forall\, m \in Nbrs(n) : n \in Nbrs(m) \\
& \land \{MsgDelay, TODelay, Period\} \subseteq \{r \in Real : r > 0\}
\end{aligned}
$$

$Ball(i, n) \triangleq$ The set of nodes a distance of at most i from node n.

$$
\begin{aligned}
& \text{LET } B[j \in 0 \,..\, i] \triangleq \text{IF } j = 0 \\
& \qquad\qquad\qquad\qquad \text{THEN } \{n\} \\
& \qquad\qquad\qquad\qquad \text{ELSE } B[j-1] \cup \text{UNION } \{Nbrs(m) : m \in B[j-1]\} \\
& \text{IN } B[i]
\end{aligned}
$$

$$Min(S) \triangleq \text{CHOOSE } i \in S : \forall\, j \in S : i \leq j$$

The minimum of a non-empty set S of numbers

$$Dist(m, n) \triangleq Min(\{i \in 0 \,..\, N : m \in Ball(i, n)\})$$

The distance between nodes m and n, if it is finite.

$$
\begin{aligned}
& Correctness \triangleq \\
& \text{LET } Ldr(n) \triangleq Min(Ball(N, n)) \quad \text{The leader of node } n. \\
& \text{IN } \forall\, n \in Node : \\
& \qquad (now > Period + TODelay + Dist(n, Ldr(n)) * MsgDelay) \\
& \qquad\quad \Rightarrow (ldr[n] = Ldr(n))
\end{aligned}
$$

$$\text{THEOREM } LSpec \Rightarrow \Box Correctness$$

Figure 11b: Module *Leader* (end).

The actions of node n are $TimeOut(n)$, which is enabled by a timeout, and $RcvMsg(n)$, which describes the receipt of a message. The operator *MsgsSent* is used in these two actions to describe the bag of messages being sent. The *Tick* action advances *now* and and decreases the timers. Its first two conjuncts (the enabling conditions) enforce the upper bounds on message delay and on the execution of $TimeOut(n)$.

The last part of module *Leader* asserts the algorithm's correctness, starting with the required assumptions about the parameters. The state predicate *Correctness* asserts that, for each node n, if *now* is large enough, then $ldr[n]$ equals n's leader. (The definition relies on the observation that, in a graph of N nodes, the distance between any two nodes is less than N.) Invariance of *Correctness* implies that, whenever enough time has elapsed, every node knows its leader.

An inductive invariance proof of correctness is straightforward and is left as an exercise for the reader.

5 Avoiding Zeno Specifications

If *now* is just another variable, nothing prevents us from writing specifications in which it does not behave like time. It is easy to convince ourselves that the value of *now* is always a real number that never decreases. However, more subtle unphysical behaviors are possible. For example, our second specification of Fischer's algorithm allows "Zeno" behaviors, in which *now* remains bounded. We could disallow such behaviors by conjoining to the specification the formula

$$NZ \triangleq \forall r \in Real : \Diamond(now > r)$$

requiring that *now* increase without bound. But there is no need to do that. We don't care what happens in behaviors in which time is bounded, because such behaviors do not represent actual executions of the algorithm. We showed that both the physically possible behaviors and the Zeno behaviors satisfy mutual exclusion and the real-time progress property.

While allowing Zeno behaviors is not a problem, forcing them is. A specification would be incorrect, in the sense of not being physically implementable, if it ever required time to remain bounded—that is, if it could reach a state from which time was unable to increase without bound. A *nonZeno* specification is defined to be one such that any finite behavior that satisfies it can be extended to an infinite behavior satisfying it in which *now* is unbounded [2]. A sensible real-time specification must be nonZeno.

As an example of Zeno (non-nonZeno) and nonZeno specifications, consider our second specification of Fischer's algorithm. Statement c must be executed when less than *Gamma* seconds and more than *Epsilon* seconds has elapsed after control reaches it. This implies that, if $Gamma \leq Epsilon$, then *now* can advance by at most *Gamma* seconds after control reaches c. The specification is therefore Zeno if $Gamma \leq Epsilon$. It is nonZeno if $Gamma > Epsilon$.

Conjoining formula *NZ* does not solve the problem of Zeno specifications. If a specification is Zeno, requiring time to be unbounded simply rules out all finite behaviors that reach states in which *now* must remain bounded. But those behaviors were probably allowed because of an error in the specification. If $Gamma \leq Epsilon$, conjoining *NZ* to our second specification of Fischer's algorithm asserts that thread t cannot execute statement b when $x = t$.

The problem of avoiding Zeno specifications exists for implicit-time as well as explicit-time specifications. Implicit-time languages can be constrained to permit only nonZeno specifications, but at a cost to their expressiveness. Such constraints would probably turn out to be instances of a general theorem for showing that specifications written in a conjunctive style are nonZeno [2, Theorem 1]. However, that theorem does not apply to the kind of simple explicit-time specifications considered here.

Being nonZeno means that for any real number r, from any reachable state it is possible to reach a state in which *now* is greater than r. This assertion can be expressed and proved in some logics, but not in the linear-time logic underlying TLA. There is a general method of using TLA to prove that a specification is nonZeno. Let *Next* be the specification's next-state action and let $s \xrightarrow{A} t$ mean that the pair s, t of successive states is an action A step. Define a *subaction* of the specification to be an action A satisfying the following condition: for any reachable state s, if there exists a state t such that $s \xrightarrow{A} t$, then there exists a state u such that $s \xrightarrow{A \land Next} u$. In particular, A is a subaction if A implies *Next*. To prove that a specification is nonZeno, one finds weak and/or strong fairness conditions on subactions of the next-state action such that they and the specification imply *NZ* [2]. (This is an instance of a general method for proving possibility properties [24].) For specification *FSpec*1 of Fischer's algorithm, it suffices to take weak fairness of $StmtB(t)$ for every thread t and strong fairness of $Tick \land (now' \geq now + 1)$.

An analogous method for proving that a specification is nonZeno should be possible with other formalisms. However, in most other formalisms, actions are linguistic constructs rather than formulas, so the corresponding

proof may require using semantic reasoning to rewrite the specification.

I expect that in most applications, the intended specification will be obviously nonZeno—as is the case with our examples. However, when writing a formal specification, it is easy to make small mistakes that yield a specification quite different from the intended one. It's therefore a good idea to check a specification in as many ways as we can. Verifying that an "obviously" nonZeno specification is really nonZeno provides a useful check.

6 Model Checking

6.1 General Observations

Model checking a specification consists of mechanically verifying that all possible executions satisfy the desired correctness properties. Specifications typically contain unspecified parameters, such as the number of processes or the size of a buffer. Ordinary model checking is performed for specific instances of the specification obtained by substituting actual values for the parameters. The likelihood that model checking has missed an error in the specification depends on the variety of different instances that have been checked. There are more sophisticated forms of model checking that employ abstraction techniques, sometimes with simple mechanical theorem checking, to verify the specification for all values of the parameters. However, these approaches are still primarily topics of research and are not widely used. I will restrict my attention to ordinary, naive model checking algorithms.

Model checking requires that the set of reachable states be finite. In practice, not only must that set be finite, but it must not be too large. The size of the state space is often an exponential function of the parameters. This usually means that one cannot check large enough instances of the specification to obtain complete confidence in its correctness. However, checking even small instances usually catches many bugs.

Real-time specifications have an infinite set of reachable states because time is unbounded. A simple method for checking infinite-state specifications is to restrict model checking to a finite subset of the set of reachable states. The TLC model checker can be instructed to limit itself to examining states that satisfy a *constraint*, which can be an arbitrary state predicate. For a discrete-time specification that starts with $now = 0$, we can remove the infinite number of times by using the constraint $now \leq MaxNow$ for some constant $MaxNow$. Of course, the model checker can then find only errors that manifest themselves within $MaxNow$ seconds.

There is a better way to model-check real-time specifications than by explicitly bounding time. A clue to how it can be done is provided by the specifications of Fischer's algorithm. As observed above, we could simply eliminate *now* from those specifications without changing the algorithm. The resulting specifications then have a finite number of reachable states and can easily be model checked. Fischer's algorithm illustrates the fact that almost all real-time specifications are symmetric with respect to time translation. What a system does next generally depends on the amount of time that has elapsed since other events have occurred, not on the actual time. To explain how to take advantage of this time symmetry, I first explain what symmetry means and how it can be used in model checking.

6.2 Model Checking with Symmetry

6.2.1 Specifications and Temporal Properties

For now, I take a semantic view in which a state is an assignment of values to the sequence *vars* of all the specification's variables[10]. The *state space* of a specification is the set of all such states. Semantically, a state predicate is a predicate (Boolean function) on states, and an action is a predicate on pairs of states. The formula $s \xrightarrow{A} t$ asserts that action A is true on the pair s, t of states.

A *behavior* is a sequence of states. A *temporal property* is a predicate on behaviors. Temporal properties are represented syntactically as temporal formulas. We usually conflate the property and the formula that represents it.

I assume a specification $\mathcal{S}$ that consists of an initial predicate *Init*, a next-state action *Next*, and a liveness assumption L. (If the specification has no liveness assumption, then $L = \text{TRUE}$.) For TLA, *Next* is an action of the form $[N]_{vars}$ that allows stuttering steps. The initial predicate and next-state action form the *safety part* of the specification $\mathcal{S}$, which I write $\overline{\mathcal{S}}$. A behavior $s_1, s_2, \ldots$ satisfies $\overline{\mathcal{S}}$ iff s_1 satisfies *Init* and $s_i \xrightarrow{Next} s_{i+1}$ for all i. The behavior satisfies $\mathcal{S}$ iff it satisfies both $\overline{\mathcal{S}}$ and the liveness assumption L.

6.2.2 Symmetry

A *symmetry* is an equivalence relation on states. A state predicate P is *symmetric with respect to* a symmetry $\sim$ iff, for any states s and t with

[10] This is different from the usual semantics of TLA in which a state is an assignment of values to *all* variables.

$s \sim t$, predicate P is true in state s iff it is true in state t. An action A is *symmetric with respect to* $\sim$ iff for any states s_1, s_2, and t_1,

$$\begin{array}{ccc} s_1 & \xrightarrow{A} & t_1 \\ \wr & & \\ s_2 & & \end{array} \quad \text{implies there exists } t_2 \text{ such that} \quad \begin{array}{ccc} s_1 & \xrightarrow{A} & t_1 \\ \wr & & \wr \\ s_2 & \xrightarrow{A} & t_2 \end{array}$$

In other words, for any states s_1 and s_2 with $s_1 \sim s_2$ and any state t_1, if $s_1 \xrightarrow{A} t_1$ then there exists a state t_2 with $t_1 \sim t_2$ such that $s_2 \xrightarrow{A} t_2$. I usually omit *with respect to* $\sim$ when it is clear what the relation $\sim$ is.

A symmetry $\sim$ is extended to an equivalence relation on behaviors in the obvious way by letting two behaviors be equivalent iff they have the same length and their corresponding states are equivalent. A temporal property is *symmetric* (with respect to $\sim$) iff, for every pair of behaviors σ and τ with $\sigma \sim \tau$, the property is true of σ iff it is true of τ.

A temporal formula is constructed from state predicates and actions by applying temporal operators, logical connectives, and ordinary (non-temporal) quantification. The formula is obviously symmetric if each of its component state predicates and actions is symmetric. The converse is not true. For example, the formula $\Box P \lor \Diamond \neg P$ is symmetric even if the predicate P is not symmetric, because it is true for all behaviors.

6.2.3 Model Checking

An explicit-state model checker such as TLC works by computing the directed graph $\mathcal{G}$ of a specification's reachable states. The nodes of $\mathcal{G}$ are states, and $\mathcal{G}$ is the smallest graph satisfying the following two conditions: (i) $\mathcal{G}$ contains all states satisfying the initial predicate *Init*, and (ii) if state s is a node of $\mathcal{G}$ and $s \xrightarrow{Next} t$, then $\mathcal{G}$ contains the node t and an edge from s to t. Paths through $\mathcal{G}$ (which may traverse the same node many times) starting from an initial state correspond to behaviors satisfying the specification's safety part $\overline{\mathcal{S}}$. Those behaviors that also satisfy its liveness assumption are the ones that satisfy the specification.

The model checker constructs $\mathcal{G}$ by the following algorithm, using a set $\mathcal{U}$ of unexamined reachable states.

- Let $\mathcal{U}$ equal the set of states satisfying *Init* and let $\mathcal{G}$ be the graph with set of nodes $\mathcal{U}$ and no edges. More precisely, start with $\mathcal{U}$ and $\mathcal{G}$ empty, sequentially enumerate the states satisfying *Init*, and add each state not already in $\mathcal{G}$ to both $\mathcal{U}$ and $\mathcal{G}$.

- While $\mathcal{U}$ is nonempty, choose some state s in $\mathcal{U}$ and enumerate all states t satisfying $s \xrightarrow{Next} t$. For each such t: (i) if t is not in $\mathcal{G}$ then add it to $\mathcal{G}$ and to $\mathcal{U}$; (ii) if there is no edge from s to t in $\mathcal{G}$, then add one.

When model checking under a constraint, a state is added to $\mathcal{U}$ and $\mathcal{G}$ only if it satisfies the constraint.

Model checking under a symmetry $\sim$ consists of constructing a smaller graph $\mathcal{E}$ by adding a state to $\mathcal{U}$ and $\mathcal{E}$ only if $\mathcal{E}$ does not already contain an equivalent state. The graph $\mathcal{E}$ constructed in this way satisfies the following properties:

- $s \not\sim t$ for every distinct pair of nodes s, t of $\mathcal{E}$.
- For every state s satisfying *Init*, there is a node t in $\mathcal{E}$ such that t satisfies *Init* and $s \sim t$.
- For every node s of $\mathcal{E}$ and every state t such that $s \xrightarrow{Next} t$, the graph $\mathcal{E}$ contains a node t' with $t \sim t'$ and an edge from s to t'.

The model checker then checks the specification as if $\mathcal{E}$ were the reachable-state graph.

A real model checker can execute such an algorithm only if the graph it constructs is finite. Otherwise, it will never finish and will eventually run out of storage. However, we can define a theoretical model checker that performs these algorithms even for an infinite state space. Such a model checker can check all the specifications in Sections 2–4, with or without symmetry, despite their infinite state spaces. I will show in Section 6.3 how these specifications can be checked with a real model checker. For now, I ignore practical concerns and assume a theoretical model checker that can handle an infinite state graph. All the results apply *a fortiori* if the state graph is finite.

We would like model checking with symmetry to be equivalent to ordinary model checking. For this to be the case, the following condition must hold:

SS. A behavior satisfies $\overline{\mathcal{S}}$ iff it is equivalent (under $\sim$) to a behavior described by a path through $\mathcal{E}$ starting from an initial state.

This condition does not imply that the behaviors described by paths through $\mathcal{E}$ satisfy $\overline{\mathcal{S}}$. It asserts only that those behaviors are equivalent to ones that satisfy $\overline{\mathcal{S}}$.

A simple induction argument shows that condition SS is true if the specification satisfies two properties:

S1. (a) *Init* is symmetric, or
    (b) No two states satisfying *Init* are equivalent.

S2. *Next* is symmetric.

Let us call the specification *safety symmetric* (with respect to $\sim$) iff it satisfies S1 and S2.

An explicit-state model checker checks that a correctness property holds for every behavior described by a path through the state graph starting from an initial state. More precisely, if the specification's liveness assumption is L, a property F is checked by checking that $L \Rightarrow F$ holds for every such behavior. A symmetric property is true of a behavior iff it is true of any equivalent behavior. Condition SS therefore implies that if L is symmetric, then model checking with symmetry is equivalent to ordinary model checking for verifying a symmetric property F. Thus, model checking and model checking with symmetry are equivalent for a safety symmetric specification with a symmetric liveness assumption.

The simplest kind of temporal property is a state predicate P, which as a temporal formula asserts that P is true initially. It is obvious that if the specification satisfies S1(b), then model checking with symmetry is equivalent to ordinary model checking for verifying that P is satisfied, even if P is not symmetric.

6.2.4 Expressing Symmetry

TLC provides two ways of describing symmetries. The first is symmetry under a set Π of permutations of a constant set C. States s and t are equivalent under this symmetry iff there is a permutation π in Π such that replacing every c in C by $\pi(c)$ transforms s to t. The general definition of symmetry under a set of permutations is difficult, but its meaning is fairly obvious for the permutation sets that TLC handles.

In the Fischer algorithm, we can let Π be all permutations of the set *Thread* of threads. It is easy to see that the initial condition and the next-state action of the two Fischer algorithm specifications *FSpec*1 and *FSpec*2 are symmetric under this set of permutations, so those specifications are safety symmetric. The invariant *MutualExclusion* is also symmetric. Hence, we can (theoretically) check that our specifications of Fischer's algorithm guarantee mutual exclusion by model checking with symmetry under permutations of threads.

The liveness property *Progress* of the specification *FSpec*1 is symmetric under this symmetry relation, since it has the form $P \leadsto Q$ where predicates P and Q are symmetric. However, the liveness assumption *Liveness* is not. This is not obvious, since performing a permutation of the threads throughout a behavior does not change whether or not the behavior satisfies *Liveness*. However, the definition of symmetry requires that the truth of the property not change even if we permute the threads differently in different states of the behavior. Property *Liveness* is the conjunction of fairness conditions on all the individual threads. By choosing the permutation for the threads separately in each state, we can transform a behavior satisfying *Liveness* into an equivalent one in which some particular thread never takes a step. (There is no requirement that the equivalent behavior satisfy the safety part of the specification.) In general, we can be sure that the conjunction of fairness conditions for different actions is symmetric only if each of those actions is symmetric. This is not the case for property *Liveness*, since each of its fairness conditions is for an action of a particular thread. We therefore cannot check liveness property of *FSpec*1 by model checking with symmetry under permutations of threads.

The second method TLC provides for describing a symmetry is *view symmetry*. A view symmetry is defined by an arbitrary state function called a *view*. (A state function is an expression that contains only constants and unprimed variables.) Two states are equivalent under a view V iff the value of V is the same in the two states. Many explicit-state model checkers test if a state s is in the state graph $\mathcal{G}$ constructed so far by keeping the set of fingerprints of nodes in $\mathcal{G}$ and testing if $\mathcal{G}$ contains a node with the same fingerprint as s. Such a checker is easily modified to implement checking under view symmetry by keeping fingerprints of the views of states rather than of the states themselves.

For the Fischer algorithm, we let V consist of the tuple of all the specification's variables except *now*. This means that two states are equivalent iff they differ only in the value of *now*. It is easy to see that our two specifications of the algorithm are safety symmetric under this symmetry. (The *Init* predicates are not symmetric, but condition S1(b) holds.) The invariant *MutualExclusion* is symmetric, so we can use model checking with symmetry under this view to verify mutual exclusion. Both the property *Progress* and the liveness assumption *Liveness* of module *Fischer*1 are symmetric under this view. We can therefore use model checking under this view to check that *FSpec*1 satisfies its liveness property.

We can try using this same idea for any real-time specification, defining a view to consist of the tuple of all variables except *now*. The specification

LSpec of the leader algorithm in Section 4 is safety symmetric, but its invariant *Correctness* is not symmetric because it depends on the value of *now*. The specification *HFSpec*2 of the Fischer algorithm with history variable h in Section 3.3 is not safety symmetric under this view because its next-state action can set h to a value that depends on the value of *now*. Its correctness condition is the invariance of a state predicate that is also not symmetric under this view because it depends on the value of *now*.

View symmetry is equivalent to abstraction [12, 15] for a symmetric specification $\mathcal{S}$. Abstraction consists of checking $\mathcal{S}$ by model checking a different specification $\mathcal{A}$ called an abstraction of $\mathcal{S}$. The view corresponds to the abstraction mapping from states of $\mathcal{S}$ to states of $\mathcal{A}$. For our specifications of Fischer's algorithm, view symmetry under the view V defined above is equivalent to an abstraction in which $\mathcal{A}$ is obtained from $\mathcal{S}$ by eliminating the variable *now*.

A model checker may support checking under view symmetry or abstraction. If not, one must construct the abstract specification $\mathcal{A}$ by hand.

6.2.5 Symmetry Under Time Translation

We have seen above that our two versions of Fischer's algorithm are safety symmetric under the view consisting of the tuple of all variables except *now*. That symmetry is a special case of time-translation symmetry, in which two states are equivalent iff they are the same except for absolute time. I now define what this means, using the notation that $s.v$ is the value of variable v in state s.

A *time translation* is a family of mappings T_d on the state space of the specification $\mathcal{S}$ that satisfies the following properties, for all states s and all real numbers d and e.

- $T_d(s).now = s.now + d$
- $T_0(s) = s$
- $T_{d+e}(s) = T_d(T_e(s))$

Specification $\mathcal{S}$ is *invariant under* this time translation iff it satisfies the following two conditions, for all real numbers d.

T1. (a) A state s satisfies *Init* iff $T_d(s)$ does, or
(b) $s.now = t.now$ for any states s and t satisfying *Init*.

T2. $s \xrightarrow{Next} t$ iff $T_d(s) \xrightarrow{Next} T_d(t)$, for any states s and t.

Given a time translation, we define the *time-translation symmetry* $\sim$ by $s \sim t$ iff $s = T_d(t)$ for some d. It is easy to check that T1 and T2 imply S1 and S2 for this symmetry. Hence, a specification that is invariant under a time translation is symmetric under the corresponding time-translation symmetry. Invariance under time translation is stronger than time-translation symmetry because, in addition to implying SS, it implies the following property.

TT. Let $s_1, \ldots, s_k$ and $t_1, t_2, \ldots$ be two behaviors satisfying $\overline{\mathcal{S}}$ (the second behavior may be finite or infinite). If $s_k = T_d(t_j)$, then the behavior $s_1, \ldots, s_k, T_d(t_{j+1}), T_d(t_{j+2}), \ldots$ also satisfies $\overline{\mathcal{S}}$.

To define a time translation, we must define $T_d(s).v$ for every real number d, state s, and variable v. Explicit-time specifications have three kinds of variables: *now*, timer variables, and "ordinary" variables that are left unchanged by the *Tick* action. We know that $T_d(s).now$ equals $s.now + d$. Time translation should not change the value of an ordinary variable v, so we should have $T_d(s).v = s.v$ for such a variable. For a timer variable t, we should define $T_d(s).t$ so that the number of seconds in which t will time out is the same in s and $T_d(s)$. We have defined three kinds of timer variables: countdown timers, count-up timers, and expiration timers. The value of a countdown or count-up timer directly indicates the number of seconds until it times out, so $T_d(s).ct$ should equal $s.ct$ for such a timer ct. Whether or not an expiration timer et has timed out depends on the value of $et - now$. The time translation T_d preserves the number of seconds until et times out iff $T_d(s).et - T_d(s).now$ equals $s.et - s.now$. Since $T_d(s).now = s.now + d$, this is true iff $T_d(s).et = s.et + d$.

With this definition of the T_d, any explicit-time specification is invariant under time translation, and hence safety symmetric under time-translation symmetry, if it expresses real-time requirements only through timer variables. Let $v_1, \ldots, v_m$ be the specification's ordinary variables and countdown and count-up timer variables, and let $et_1, \ldots, et_n$ be its expiration timer variables. Then symmetry under time translation is the same as view symmetry with the view

$$\langle v_1, \ldots, v_m, et_1 - now, \ldots, et_n - now \rangle$$

(In case an expiration timer can have the value ∞ or $-\infty$, we define $\pm\infty - r$ to equal $\pm\infty$ for any real number r.)

Since the specifications *FSpec*1 and *FSpec*2 of Fischer's algorithm and *LSpec* of the leader algorithm use only countdown timers, time symmetry is

the same as view symmetry with the view consisting of the tuple of all variables other than now. In the specification $HFSpec2$ of module $HFischer2$, the variable h is an expiration timer, being set to now when a certain condition becomes true and reset to $Infinity$ when it becomes false. Variables $ubTimer$ and $lbTimer$ are countdown timers and pc and x are ordinary variables, so this specification is invariant under time translation, and time symmetry is the same as view symmetry under the view

$$\langle pc,\ x,\ ubTimer,\ lbTimer,\ h - now \rangle$$

This is not quite correct in TLA$^+$ because the standard *Reals* module defines *Infinity* only to satisfy $-Infinity < r < Infinity$ for any real r, not to satisfy $Infinity - r = Infinity$. We therefore define $\ominus$ by

$$s \ominus r \ \triangleq\ \text{IF}\ s = Infinity\ \text{THEN}\ Infinity\ \text{ELSE}\ s - r$$

and write the view as

$$\langle pc,\ x,\ ubTimer,\ lbTimer,\ h \ominus now \rangle$$

Specification $HFSpec2$ is safety symmetric under this view. Moreover, its correctness property

$$\Box(now - (4 * Delta + 2 * Gamma - Epsilon) < h)$$

simply asserts that the timer h never times out, so it is also symmetric under this view. It is easy to see this symmetry directly, since the condition is equivalent to

$$\Box(-(4 * Delta + 2 * Gamma - Epsilon) < h \ominus now)$$

which depends only on $h \ominus now$. We can therefore use model checking under this view symmetry to check that $HFSpec2$ satisfies its correctness property.

6.2.6 Periodicity and Zeno Behaviors

A nonZeno behavior is one that satisfies property NZ, which asserts that time increases without bound. Property NZ is not symmetric under time translation. By replacing states of a behavior with ones translated back to the behavior's starting time, we can construct an equivalent behavior in which now never changes.

A specification $\mathcal{S}$ is nonZeno iff every finite behavior satisfying $\mathcal{S}$ can be extended to an infinite one satisfying $\mathcal{S}$ and NZ. Since NZ is not symmetric under time translation, model checking with time-translation symmetry

cannot be used to check that a specification is nonZeno. However, we can take advantage of time-translation invariance when using ordinary model checking to show that a specification is nonZeno. I now explain how this is done.

Let $\mathcal{S}$ be a specification that is invariant under time translation. For simplicity, let's assume that the initial condition of $\mathcal{S}$ asserts that *now* equals 0, so $s.now \geq 0$ for all reachable states s. For any reachable state s, let $LeastTime(s)$ be the greatest lower bound of the values $t.now$ for all states t equivalent to s (under time-translation symmetry). The *period* of $\mathcal{S}$ is defined to be the least upper bound of the values $LeastTime(s)$ for all reachable states s of $\mathcal{S}$. Intuitively, if a system's specification has a finite period λ, then all its possible behaviors are revealed within λ seconds. More precisely, any λ-second segment of a system behavior is the time translation of a segment from the first λ seconds of some (possibly different) behavior.

Let us define the condition NZ_λ as follows, where λ is a positive real number.

NZ_λ. Every finite behavior satisfying $\overline{\mathcal{S}}$ that ends in a state s with $s.now \leq \lambda$ can be extended to a behavior satisfying $\overline{\mathcal{S}}$ that ends in a state t with $t.now \geq \lambda + 1$.

Assume that specification $\mathcal{S}$ is time-translation invariant, has a period less than or equal to the real number λ, and satisfies NZ_λ. The following proof shows that $\mathcal{S}$ is then nonZeno.

1. ASSUME: Any finite behavior σ satisfying $\overline{\mathcal{S}}$ with final state s can be extended to a behavior τ satisfying $\overline{\mathcal{S}}$ with final state t such that $t.now \geq s.now + 1$.

   PROVE: $\mathcal{S}$ is nonZeno.

   1.1. For any natural number k, any finite behavior σ satisfying $\overline{\mathcal{S}}$ with final state s can be extended to a behavior τ_k satisfying $\overline{\mathcal{S}}$ with final state t such that $t.now \geq s.now + k$.

   PROOF: By simple induction from the step 1 assumption.

   1.2. Q.E.D.

   PROOF: To prove that $\mathcal{S}$ is nonZeno, we must show that any finite behavior σ satisfying $\overline{\mathcal{S}}$ can be extended to an infinite behavior τ satisfying $\overline{\mathcal{S}}$ in which the value of *now* grows without bound. We can let τ equal the limit as $k \to \infty$ of the behaviors τ_k whose existence is asserted by step 1.1. The behavior τ satisfies $\overline{\mathcal{S}}$ because every finite prefix of it does.

2. ASSUME: σ is a finite behavior satisfying $\overline{\mathcal{S}}$ with final state s.

PROVE: There exists a behavior τ satisfying $\overline{\mathcal{S}}$ that extends σ and has final state t with $t.now \geq s.now + 1$.

2.1. Choose real number d and reachable state u with $u.now \leq \lambda$ and $s = T_d(u)$.

PROOF: d and u exist by the reachability of s and the assumption that $\mathcal{S}$ has period at most λ.

2.2. Let ρ be a behavior satisfying $\overline{\mathcal{S}}$ that ends in u.

PROOF: ρ exists by step 2.1, which asserts that u is reachable.

2.3. Extend ρ by appending a sequence of states $w_1, \ldots, w_n$ to obtain a behavior satisfying $\overline{\mathcal{S}}$ such that $w_n.now \geq \lambda + 1$.

PROOF: This can be done because ρ satisfies $\overline{\mathcal{S}}$ by 2.2 and $\mathcal{S}$ is assumed to satisfy NZ_λ.

2.4. The behavior τ obtained by appending $T_d(w_1), \ldots, T_d(w_n)$ to σ satisfies $\overline{\mathcal{S}}$.

PROOF: Since $s = T_d(u)$ (step 2.1) and ρ has final state u (step 2.2), TT and 2.3 imply that τ satisfies $\overline{\mathcal{S}}$. (TT holds because $\mathcal{S}$ is assumed to be invariant under time translation.)

2.5. Q.E.D.

Since $T_d(w_n)$ is the final state of τ, 2.4 implies that to complete the proof of step 2, we need only show that $T_d(w_n).now \geq s.now + 1$. This follows from $u.now \leq \lambda$ (step 2.1) , $w_n.now \geq \lambda + 1$ (step 2.3), and

$$\begin{aligned} T_d(w_n).now &= w_n.now + d \\ &\geq u.now + d + 1 \quad \text{[by } w_n.now \geq \lambda + 1 \geq u.now + 1\text{]} \\ &= s.now + 1 \quad \text{[because } s = T_d(u) \text{ by 2.1]} \end{aligned}$$

3. Q.E.D.

PROOF: Steps 1 and 2 trivially imply that $\mathcal{S}$ is nonZeno.

Let us review what has just been proved. Under the assumption that the initial states of $\mathcal{S}$ all have $now = 0$, I showed that if $\mathcal{S}$ is invariant under time translation, has a period of at most λ, and satisfies NZ_λ, then it is nonZeno. To use model checking to prove that $\mathcal{S}$ is nonZeno, the checker must be able to verify that $\mathcal{S}$ has a period of at most λ and that it satisfies NZ_λ.

Here is how we can use model checking under time-translation symmetry to find an upper bound on the period of $\mathcal{S}$. Let $\mathcal{E}$ be the state graph constructed by model checking under this symmetry. Because every reachable state is equivalent to a node in $\mathcal{E}$, the period of $\mathcal{S}$ is less than or equal

to the least upper bound of the values $s.now$ for all nodes s of $\mathcal{E}$. (Since all initial states have $now = 0$, the period of most specifications will equal this least upper bound if the model checker uses a breadth-first construction of the state graph.) Debugging features allow the TLC user to insert in the specification expressions that always equal TRUE, but whose evaluation causes TLC to perform certain operations. Using these features, it is easy to have TLC examine each state s that it finds and print the value of $s.now$ iff $s.now > t.now$ for every state t it has already found.[11] This makes computing an upper bound on the period of $\mathcal{S}$ trivial, if the graph $\mathcal{E}$ is finite. An explicit-state model checker that lacks the ability to compute the upper bound can verify that λ is an upper bound on the period by performing model checking under time-translation symmetry to verify the invariance of $now \leq \lambda$.

To check that $\mathcal{S}$ satisfies NZ_λ, we must show that from every reachable state with $now \leq \lambda$, it is possible to reach a state with $now \geq \lambda + 1$. We can do this by model checking with the constraint $now \leq \lambda + 1$, in which the model checker ignores any state it finds with $now > \lambda + 1$. This is easy to do with a model checker that can check possibility properties. With one that checks only linear-time temporal properties, we must show that $\mathcal{S}$ together with fairness assumptions about subactions of its next-state action imply that the value of now must eventually reach $\lambda + 1$. That is, we add fairness assumptions on certain actions and check the property $\Diamond(now \geq \lambda + 1)$ under the constraint $now \leq \lambda + 1$.

There is one tricky point to checking a liveness property F under a constraint. The liveness assumption L might be violated by all behaviors satisfying the constraint, in which case the checker would decide that the property F holds because it finds $L \Rightarrow F$ to be trivially true. In particular, a fairness assumption on the $Tick$ action could imply that now grows without bound, which would be false for every path through the state graph constructed under the constraint $now \leq \lambda + 1$. Model checking with a constraint P is equivalent to ordinary model checking of the specification obtained by conjoining the condition P' to the next-state action. For our specifications, model checking with the constraint $now \leq \lambda + 1$ is equivalent to changing the $Tick$ action to $Tick \land (now' \leq \lambda + 1)$. The fairness conditions we use to check NZ_λ must be on subactions of the modified next-state action. Recall that we can prove that the specification $FSpec1$ of Fischer's algorithm is nonZeno by proving NZ under the assumptions of weak fairness on $StmtB(t)$ for every thread t and strong fairness of $Tick \land (now' \geq now + 1)$.

[11] One of the features needed was added to TLC after publication of [25].

For verifying NZ_λ by model checking with the constraint $now \leq \lambda + 1$, we must replace the latter assumption with strong fairness of the action $Tick \wedge (\lambda + 1 \geq now' \geq now + 1)$.

All of this, including the definition of *period*, has been under the assumption that $now = 0$ for all initial states. Extending the definition of *period* to the general case is not hard, but there is no need to do it. Invariance under time translation (in particular, condition T1) requires that either (a) the set of initial states is invariant under time translation, or (b) the value of *now* is the same in all initial states. In case (b), that value will probably either be 0 or else a parameter of the specification that we can set equal to 0. In case (a), we conjoin the requirement $now = 0$ to the initial predicate. Invariance under time translation implies that, in either case, modifying the specification in this way does not affect whether or not it is nonZeno.

6.2.7 Checking Inductive Invariance

Model checking can at best verify specific instances of a specification. Attaining sufficient confidence in a specification's correctness may require a proof. Hand proofs are error-prone, but mechanical verification is usually too time-consuming to be practical. A compromise is to make hand proofs more reliable by using a model checker to find errors in it.

At the heart of any assertional proof is an inductive invariant—an invariant *Inv* such that, for any state s (not necessarily reachable) satisfying *Inv*, every state t satisfying $s \xrightarrow{Next} t$ also satisfies *Inv*. A model checker can easily check that *Inv* is an invariant of the specification. In principle, it can just as easily check that *Inv* is inductive by checking that it is an invariant of the specification obtained by replacing the initial predicate with *Inv*.

While this works in principle, it seldom works in practice. There are usually too many states that satisfy the inductive invariant—many more than are reachable. Moreover, even if the set of states satisfying the invariant is not too large, computing it may take too long. Most inductive invariants start with conjuncts that express type correctness. To compute the set of states satisfying such an invariant *Inv*, TLC enumerates all type-correct states and throws away the ones not satisfying the rest of *Inv*. This process can sometimes be made more efficient by using a more sophisticated type-correctness invariant that takes advantage of relations among the variables. However, it is still usually feasible only for very small instances of the specification.

For a time-translation invariant specification and a time-symmetric inductive invariant, model checking can use time symmetry. However, the

type invariant for a discrete-time specifications asserts only that *now* is a natural number. To bound the state space, we must modify the type invariant to assert that $0 \leq now \leq \mu$ for some sufficiently large μ. The value of μ is large enough if every reachable state found by the model checker is an initial state. TLC finds reachable states by a breadth-first search and prints the depth of the search tree, which equals 1 iff every reachable state is an initial state. TLC's ability to print the maximum value of *now* allows one to find a large enough value of μ. For a specification that uses only countdown or count-up timers and no expiration timers, μ can be taken to equal 0. Even if checking cannot be performed with a large enough μ to ensure that the invariant is inductive, it is still a good way to try to find errors.

An explicit-state model checker can seldom check inductive invariance on large enough instances of a specification to gain much confidence in the invariant's correctness. However, because they don't enumerate states, symbolic model checkers based on Boolean decision diagrams or satisfiability solving may be better at checking inductive invariance. I don't know if they have been used in this way. It is worth testing an inductive invariant with a model checker even on tiny instances of the specification because any kind of mechanical checking usually reveals errors. It's best to correct all errors the model checker can find before trying to write a proof.

6.3 Model Checking Our Specifications

6.3.1 Modifying the Specifications

TLA$^+$ was not designed with model checking in mind. It is much too expressive for every possible specification to be model checked. However, the natural way of writing TLA$^+$ specifications of systems and algorithms yields specifications that TLC can almost always check—at least for small instances. TLC was designed so that, in most cases, one can check the specification without having to modify it. For example, one can check particular instances by instructing TLC to substitute specific constants for constant parameters. In practice, one usually writes a "test harness" module that imports the specification module and adds things like the definition of the view.

Our specifications use continuous time. For model checking, we must modify them to make time discrete. We do this by modifying their *Tick* actions to increment *now* by 1. In all our specifications, this can be done by simply replacing

$$Tick \;\triangleq\; \exists d \in \{r \in Real : r > 0\} : \;\ldots$$

with

$$Tick \triangleq \text{LET } d = 1 \text{ IN } \ldots$$

I could have avoided having to make this change by writing

$$PosReal \triangleq \{r \in Real : r > 0\}$$
$$Tick \triangleq \exists d \in PosReal : \ldots$$

and then instructing TLC to substitute $\{1\}$ for *PosReal*. However, that would have been "cheating", since I would have written the specification that way only for this purpose.

Had we allowed *now* to assume any real value in the initial state, then we would also have to modify the initial predicate by replacing the conjunct $now \in Real$ with $now = 0$.

Specification *FSpec*1 of Fischer's algorithm has as liveness assumption formula *NZ*, which equals $\forall r \in Real : \Diamond(now > r)$. TLC cannot handle such a conjunction over an infinite set of values, so we must change this condition for model checking. Since *now* is advanced only by a *Tick* action, a behavior that satisfies *NZ* must take infinitely many *Tick* steps. Hence, *NZ* implies strong fairness of a *Tick* action. Therefore, to verify a liveness property, it suffices to replace the assumption *NZ* by $\text{SF}_{vars}(Tick)$. I have done this for model checking. In discrete-time specifications, strong fairness of *Tick* is better than *NZ* for asserting that time advances, since it allows one to ensure syntactically that the fairness properties are on subactions of the specification.

6.3.2 Measurements

The execution-time results reported here for TLC were obtained on a dual processor 2.4 GHz PC running Windows XP. TLC is written in Java, and it was run under the BEA WebLogic JRockit™ version 1.4.2_04 Java Virtual Machine (JVM). TLC uses multiple threads when building the state graph and checking safety properties. With a thread-friendly JVM, TLC achieves speedups of close to N with N processors. With 2 processors, the speedup under the BEA JVM is roughly a factor of 1.5. When checking a liveness property on the completed state graph, TLC is single threaded.

The execution times include a fixed startup time of a little over 3 seconds. For example, model checking Fischer's algorithm with an empty set of threads, so there is a single reachable state, takes about 3.2 seconds.

I made no effort to achieve great precision in the timing measurements, often using the computer concurrently for other tasks like editing and reading mail that were not processor intensive. The time to model check the same instance of a specification could vary by several seconds on short runs and several percent on longer ones.

TLC stores on disk the state graph and the list of states whose next states it has not yet examined. This means that, for checking safety properties, it is usually limited in the size of the instance it can handle only by time and not by space. When checking liveness properties, TLC uses a data structure that must fit in memory. For most of the instances tested here, disk was not needed. TLC was run with the Java runtime's maximum memory allocation parameter set to 700 MBytes, but most instances used much less memory than that.

6.3.3 Specification *FSpec*1

Fischer's algorithm is so simple that model checking it should be easy. If not, it would be unlikely for model checking real specifications to be feasible.

To model check specification *FSpec*1, we must choose specific values for the parameters *Delta* and *Epsilon* and for the set of threads. Clearly, only the number of threads matters. Since *Delta* is an upper-bound timing constraint, decreasing its value just eliminates possible behaviors. The algorithm assumes $Delta \leq Epsilon$, so it makes most sense to let $Delta = Epsilon$ for model checking. I have done that for all the tests reported, so only the value of *Delta* is mentioned.

I have used TLC to check the invariance of the predicate *MutualExclusion* defined in module *FischerPreface* and the predicates *Inv* and *LInv* used in the correctness proof. These invariants were checked by TLC using symmetry under both time translation and permutations of threads. Time-translation symmetry is expressed in TLC with the view described in Section 6.2.5.

The number of reachable states, and hence the execution time, increases with the number of threads and the value of *Delta*. For complicated algorithms, one is lucky to be able to test an instance with as many as 3 threads. Fischer's algorithm is so simple that TLC can check invariance properties for small values of *Delta* with 6 threads in less than a minute. With N threads, we expect the number of states to be roughly proportional to $Delta^N$. The graph of Figure 12 shows the dependence of the number of states on *Delta* for 4 threads. With this number of threads, TLC checks the specification at an asymptotic rate of about 2700 reachable states per second, taking 102

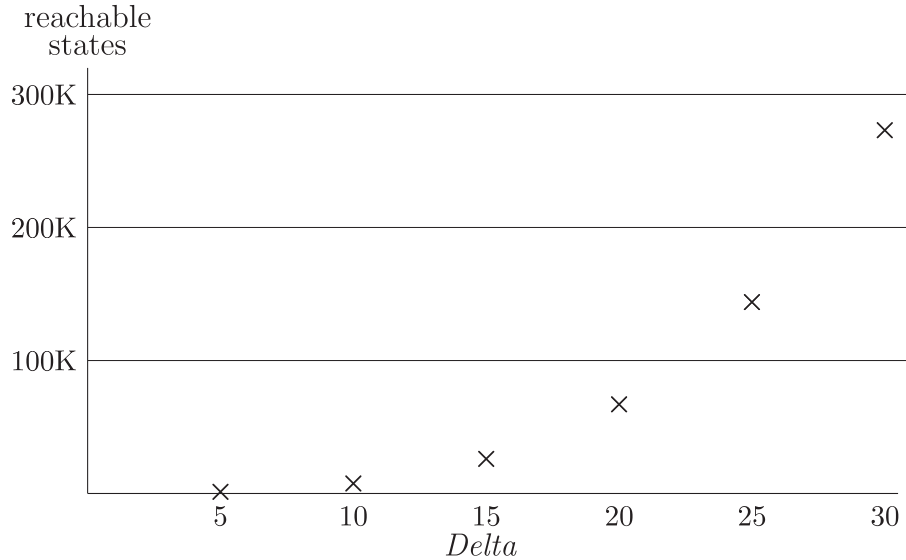


Figure 12: Number of reachable states for the specification in module *Fischer*1 with 4 threads, under time-translation and thread-permutation symmetries.

seconds to find the 273134 reachable states for $Delta = 30$.

Figure 13 shows how the number of states grows with the number of threads when $Delta = 5$. The execution time increases much more dramatically than the number of reachable states:

5 threads	3311 states	7 seconds	950 states/second
6 threads	8213 states	35 seconds	260 states/second
7 threads	18530 states	556 seconds	35 states/second

where the number of states per second is obtained by subtracting the estimated 3.2 seconds overhead from the execution times. (Execution times for fewer than 4 threads are too short for the timing measurements to be meaningful.) There are two reasons for this increase:

- Although symmetry under permutations reduces the total number of states in the state graph, it does not reduce the number of states t satisfying $s \xrightarrow{Next} t$ for each state s in the graph. That number increases with the number of threads that can take a step. As a result, the number of times TLC generates and examines each reachable state increases with the number of threads—from an average of 1.7 for a single thread to 4.6 for 7 threads.

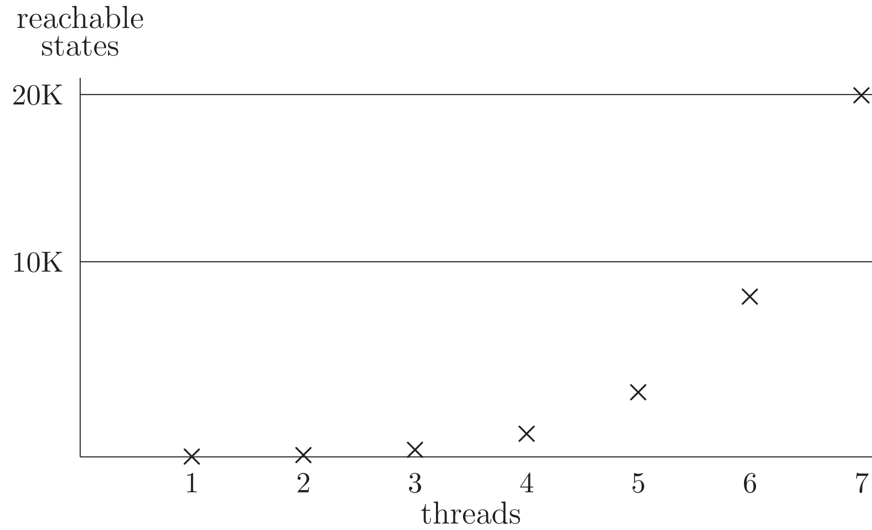


Figure 13: Number of reachable states for the specification in module *Fischer*1 with *Delta*=5, under time-translation and thread-permutation symmetries.

- The invariants are universally quantified over the set of threads, so the time required to check that they hold in an individual state increases with the number of threads.

For all instances that I have checked having more than one thread, the period equals $2 * Delta - 1$. In light of the many instances tested, it would be remarkable if this were not true in general.

Symmetry under thread permutations is very effective at reducing the number of reachable states. For N threads, this symmetry reduces the number of reachable states by a factor approaching $N!$ as *Delta* goes to infinity. For smaller values of *Delta*, a larger proportion of the states are symmetric under some permutations of threads, so the number of reachable states is reduced by a smaller factor. With 4 threads ($N! = 24$), the number of reachable states is reduced by a factor of 14.1 for $Delta = 5$ and by a factor of 20.7 for $Delta = 20$.

Since we cannot use thread permutation under symmetry to check the liveness property, we cannot check liveness for as large an instance of the specification. The largest instance I have tested has 6 threads and $Delta = 5$, which produces 2037987 reachable states and for which it takes TLC almost 3 hours to check liveness. However, TLC can check liveness with 6 threads and $Delta = 10$ (138644 reachable states) in about $6\frac{1}{2}$ minutes. It is unlikely

for the specification to have an error that does not manifest itself in this instance.

As explained in Section 6.2.6, we check that a time-translation invariant specification with period at most λ is nonZeno by checking that it satisfies condition NZ_λ. This is done by adding a suitable fairness assumption to the safety part and checking the property $\Diamond(\mathit{now} \geq \lambda + 1)$ under the constraint $\mathit{now} \leq \lambda + 1$. For arbitrary theoretical model checking, the fairness assumption includes strong fairness of the action $\mathit{Tick} \wedge (\lambda + 1 \geq \mathit{now}' \geq \mathit{now} + 1)$. For a discrete-time specification in which the *Tick* action increments *now* by 1, this is equivalent to strong fairness of $\mathit{Tick} \wedge (\lambda \geq \mathit{now})$. For this version of Fischer's algorithm, we also need weak fairness of the action $\mathit{StmtB}(t)$ for each thread t to ensure that t resets its upper-bound timer and allows *now* to advance.

Because the property to be checked is not symmetric under time translation and the liveness assumption is not symmetric under thread permutation, we can use neither of those two symmetries. The set of reachable states is bounded by the constraint $\mathit{now} \leq \lambda + 1$. However, since states that differ only in the value of *now* yield distinct nodes in the state graph, we expect the graph to have about $\lambda + 1$ times as many nodes as under time-translation symmetry. Because the lower-bound timing constraint rules out some combinations of values of *now* and the other variables, there are only about 60-80% that many nodes in the graph for checking that $\mathit{FSpec}1$ satisfies NZ_λ. Checking that an instance with four threads is nonZeno takes $1\frac{1}{2}$ minutes for $\mathit{Delta} = 5$ and 30 minutes for $\mathit{Delta} = 10$.

The Fischer algorithm is simple enough that we can hope to check its inductive invariant on large enough instances to gain confidence in its correctness—especially since we can use symmetry under thread permutations as well as time symmetry. The limiting factor in the size of instance that TLC can handle is that the current implementation gives up if it finds more than one million initial states. (We have not yet encountered an engineer who writes inductive invariants, and real specifications are unlikely to have more than a few hundred initial states, so there has been little incentive to remove this limit.) The largest instances for which TLC can check that $\mathit{Inv} \wedge \mathit{LInv}$ is an invariant of this specification are

2 threads	$\mathit{Delta} = 12$	490347 states	checked in $1\frac{1}{2}$ minutes
3 threads	$\mathit{Delta} = 3$	99372 states	checked in 1 minute
4 threads	$\mathit{Delta} = 1$	7140 states	checked in $\frac{1}{2}$ minute

(TLC can also check it for a single thread with *Delta* in the hundreds.) These instances are large enough to give me much more confidence in my hand proof.

6.3.4 Specification *FSpec*2

Specification *FSpec*2 has the additional parameter *Gamma*, which must be greater than *Epsilon*. It is an upper-bound constraint, so increasing its value increases the set of possible behaviors. As with *FSpec*1, I checked instances with *Epsilon* equal to *Delta*.

The additional upper-bound timing constraints of *FSpec*2 give it more reachable states than *FSpec*1 for comparable parameters. For example, using the same symmetries under time translation and thread permutation, and letting *Gamma* equal $Delta + 5$, specification *FSpec*2 has roughly 3 times as many reachable states as *FSpec*1 for values of *Delta* ranging from 5 to 25. Invariance checking takes a little longer per state—2400 states per second instead of 2700. For 6 threads and $Delta = 5$, specification *FSpec*2 with $Gamma = 12$ has about 4 times as many states and takes about 50% longer per state. For two or more threads, the period of *FSpec*2 appears to equal the maximum of $2 * Delta - 1$ and $Gamma - 1$.

To verify that *FSpec*2 is nonZeno, we check condition NZ_λ using the fairness property

$$\begin{array}{l} \land \forall t \in Thread : \\ \quad \mathrm{WF}_{vars}(\land StmtA(t) \lor StmtB(t) \lor StmtC(t) \lor StmtD(t) \\ \qquad\qquad \land SetTimers(t)) \\ \land \mathrm{SF}_{vars}((now \leq \lambda) \land Tick) \end{array}$$

which clearly implies the specification's next-state action. Its larger number of reachable states makes checking that *FSpec*2 is nonZeno correspondingly harder than checking *FSpec*1, though still not very hard. For example, with 4 threads, $Delta = 5$ and $Gamma = 8$, so the period is 9, checking that *FSpec*2 is nonZeno produces a state graph with 248489 states and takes about $5\frac{1}{4}$ minutes.

Checking that *Inv* is an inductive invariant of *FSpec*2 is also quite feasible. Maximal instances for which TLC can check it are

2 threads	$Delta = 10$	$Gamma = 15$	492305 states	$1\frac{1}{2}$ minutes
2 threads	$Delta = 12$	$Gamma = 13$	467670 states	$1\frac{1}{2}$ minutes
3 threads	$Delta = 2$	$Gamma = 5$	121088 states	$1\frac{1}{2}$ minutes

6.3.5 Specification *HFSpec*2

Specification *HFSpec*2 is obtained from *FSpec*2 by adding the history variable h. As observed in Section 6.2.5, we check that

$$now - (4 * Delta + 2 * Gamma - Epsilon) < h$$

is an invariant of *HFSpec*2 using view symmetry under the view

$$\langle pc,\ x,\ ubTimer,\ lbTimer,\ h \ominus now \rangle$$

We also use symmetry under permutations of threads.

The invariant asserts that the value of h can vary from *now* to $now + 4 * Delta + 2 * Gamma - Epsilon - 1$. Hence, the number of reachable states of *HFSpec*2 should be larger than the number for *FSpec*2 by at most a factor of $4 * Delta + 2 * Gamma - Epsilon - 1$. In the tests I've run, the actual factor lies between *Delta* and $2 * Delta$. The running time per reachable state seems to be roughly the same for the two specifications. For example, with 6 threads, $Delta = 5$, and $Gamma = 10$, specification *HFSpec*2 had 5.1 times as many reachable states (175071) and took 7.6 times as long (20 minutes 10 seconds).

As with *FSpec*1 and *FSpec*2, the period of *HFSpec*2 appears to be independent of the number of threads. However, it does not seem to be a very simple function of *Delta* and *Gamma*.

We can check that *HFSpec*2 is nonZeno using the same fairness assumption as for *FSpec*2. (As explained in Section 5, fairness must be on a subaction of the specification. In general, if A is a subaction of a specification $\mathcal{S}$, then it is also a subaction of the specification obtained from $\mathcal{S}$ by adding a history variable.) In addition to having a larger state space than *FSpec*2 for the same parameter values, *HFSpec*2 also has a larger period. With 4 threads, $Delta = 3$ and $Gamma = 5$, verifying NF_{15} for *HFSpec*2 required examining 6 times as many reachable states (444638) and took 7.5 times as long (9.5 minutes) as verifying NF_9 for *FSpec*2.

As expected, inductive invariance checking for *HFSpec*2 can be performed only on smaller instances than for *FSpec*2. The maximal values of parameters for which TLC can check that *HInv* is an inductive invariant are

2 threads	$Delta = 2$	$Gamma = 8$	18182 states	3 minutes
2 threads	$Delta = 3$	$Gamma = 6$	20414 states	$3\frac{1}{2}$ minutes
3 threads	$Delta = 1$	$Gamma = 3$	3514 states	$2\frac{1}{2}$ minutes

Although small, these instances were large enough to reveal an error in an earlier version of *HInv*.

6.3.6 Specification *LSpec*

The specification *LSpec* of the leader algorithm uses only countdown timers and is invariant under time translation. It has no other symmetries; even if

the node graph is symmetric, the specification uses the node identifiers and is therefore not symmetric under permutations of nodes.

Although *LSpec* is time-translation symmetric, the invariant *Correctness* is not because it explicitly mentions *now*. We could add a timer as a history variable and restate the correctness property in terms of it, but that is not necessary. Instead, we model check under a symmetry other than simple time translation. Formula *Correctness* has the form

$$\forall n \in Node : (now > c(n)) \Rightarrow (ldr[n] = Ldr(n))$$

for a constant expression $c(n)$ independent of *now*. Let Σ be the maximum of the $c(n)$ for all nodes n. Define a symmetry $\sim$ by $s \sim t$ iff $s.now$ and $t.now$ are either equal or are both greater than Σ. It is easy to see that *Correctness* is symmetric under $\sim$. It is a little less obvious, but also true, that specification *LSpec* is symmetric under $\sim$. This symmetry is described by the view

$$\langle ldr,\ dist,\ timer,\ msgs,\ \text{IF}\ \ now > \Sigma\ \ \text{THEN}\ \ \Sigma + 1\ \ \text{ELSE}\ \ now \rangle$$

We check the invariance of *Correctness* by model checking under this view symmetry.

The parameters of the specification are N and *Nbrs*, which describe the graph, and the timing constants *Period*, *TODelay*, and *MsgDelay*. The latter two are upper-bound constraints, so the number of reachable states is an increasing function of their values. Figure 14 shows the results of checking the invariance of *Correctness* on three different graphs, with 3–5 nodes, for some haphazardly chosen values of the timing bounds. Some of those results are followed with the results of nonZeno checking for the same instance, indicating the value of λ for which NZ_λ was verified. In about a dozen instances checked for each, the periods are $2 * Period + MsgDelay + TODelay + 1$ for the 3-node graph and $Period + 3 * MsgDelay + 2 * TODelay + 1$ for the 4-node graph.

We expect that increasing a timing bound will increase the number of reachable states, since it increases the number of possible values of the timer variables. However, the first three rows for $N = 3$ and $N = 4$ show that increasing *Period* decreases the number of states. By slowing the system down, increasing a lower-bound constraint can sometimes reduce the set of possible behaviors. Increasing *Period* decreases the rate at which messages are sent. Since *Period* is a lower bound on the time between the sending of messages and *MsgDelay* is an upper bound on how long a message can remain in the multiset *msgs* before being delivered, the maximum number of messages that can be in transit at any time depends on the ratio

MsgDelay/*Period*. The following table gives some idea of what's going on, where the results are for the 3-node graph.

Period	*MsgDelay*	*TODelay*	$\frac{MsgDelay}{Period}$	states	msgs in transit max	mean
2	2	1	1	6579	6	3.46
1	2	1	2	240931	12	6.57
3	2	2	.67	20572	6	3.69
10	3	5	.33	247580	6	3.85

(The maximum and mean number of messages in transit were measured using TLC's debugging features.) The first two rows show the dramatic effect of changing *Period* and leaving the other parameters the same. The second two rows show that the *MsgDelay*/*Period* ratio is just one of the factors determining the number of messages in transit and the number of reachable states.

Checking inductive invariance seems to be infeasible for *LSpec*, even on a 2-node graph.

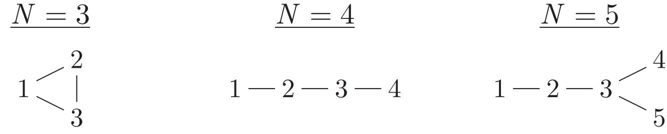
6.4 Comparison With Uppaal

A number of algorithms have been developed and implemented in model checkers for real-time systems [3, 18, 27, 43]. We would like to know how they compare with the simple method described here.

Most real-time model checkers use continuous-time models and employ clock-region constructions to check them that are more expensive than the simpler algorithms of ordinary model checkers. On the other hand, their execution speed depends only on the ratios of the timing parameters, not on the absolute values. With the simple discrete-time method described here, multiplying the parameters by a constant c usually increases the number of reachable states by a power of c. So the two methods are not directly comparable, and it would be easy to bias a comparison by the proper choice of parameters. So we can at best hope for a qualitative comparison of how the two approaches might work in practice.

Most real-time model checkers require the system to be described in timed-automata languages that are not expressive enough to describe the leader algorithm of Section 4. The only real-time model checker I know of that can handle this algorithm is Uppaal [27]. I have therefore restricted my attention to Uppaal, one of the most widely used real-time model checkers.

This section describes the use of Uppaal on two specifications—the leader algorithm and a version of Fischer's algorithm. The leader algorithm is the



N	*Period*	*MsgDelay*	*TODelay*	states	time	states/sec
3	3	2	1	5760	7 sec	800
	2	2	1	6579	10 sec	625
	1	2	1	240931	$4\frac{1}{2}$ min	900
	5	2	5	82105	70 sec	1150
			NZ_{18}	83890	140 sec	590
	5	3	5	264225	$4\frac{1}{2}$ min	1000
			NZ_{19}	270109	$8\frac{1}{2}$ min	530
	5	4	5	836152	16 min	850
	3	2	2	20572	20 sec	1050
			NZ_{11}	21260	35 sec	630
	10	3	5	247580	4 min	975
			NZ_{29}	251708	8 min	530
4	3	2	1	5606	12 sec	475
	2	2	1	6656	13 sec	490
	1	2	1	172531	$6\frac{1}{2}$ min	440
	5	2	5	179860	$6\frac{1}{2}$ min	460
			NZ_{22}	185228	$6\frac{1}{2}$ min	480
	5	3	5	728411	29 min	410
			NZ_{25}	749163	$28\frac{1}{2}$ min	440
	5	4	5	2974572	1125 min	440
	3	2	2	27576	45 sec	620
			NZ_{15}	29859	50 sec	600
	10	3	5	586504	23 min	425
			NZ_{35}	604620	$22\frac{1}{2}$ min	450
5	3	1	1	20961	75 sec	280
	5	3	1	331292	34 min	160
	3	2	2	691394	69 min	170

Figure 14: Checking that *Correctness* is an invariant of *LSpec* for the indicated graphs with 3, 4, and 5 nodes, together with nonZeno checking for some instances.

more interesting example, because it is representative of the class of systems for which one would most likely want to use existing languages and tools that are not specialized for handling real time. However, since Fischer's algorithm is so popular a benchmark, I include it as well.

Uppaal and TLC differ not only in their basic model-checking algorithms, but also in the level of their input languages. Like most model checkers, Uppaal uses a lower-level modeling language that can be compiled into efficient code. TLA$^+$ is a very high-level language, so TLC must "execute" a specification interpretively. TLC is therefore significantly slower than conventional model checkers for verifying simple systems. To explore the difference this makes, I also present data for Fischer's algorithm obtained with two popular model checkers, Spin [20] and SMV [31], whose models are written in lower-level languages.

Uppaal maintains all its working data in memory. Its use of memory appears to cause the Windows XP memory management system to thrash. I therefore ran Uppaal (with its default settings) under Linux, on a 3.1 GHz uniprocessor with 3 GBytes of memory. For the leader algorithm, Uppaal was also run at Aalborg University on a 30-node network of 2.6 GHz processors, each with 1 GByte of memory. Uppaal can easily be used to check that a specification is nonZeno, but such checking was not performed on either of the examples.

6.4.1 The Leader Algorithm

Arne Skou, an experienced Uppaal user at Aalborg University, with the assistance of Gerd Behrmann and Kim Larsen, translated *LSpec* to an Uppaal model. Since Uppaal's system modeling language is not as expressive as TLA$^+$, this required some encoding. In particular, Uppaal cannot represent a potentially unbounded multiset, so the Uppaal model encodes the TLA$^+$ variable *msgs* in a fixed-length array. Uppaal checks that this array does not overflow to ensure that the model is a faithful representation of the algorithm.

I ran the Uppaal specification with the same 3- and 4-node graphs on which I ran TLC. Uppaal proved to be very sensitive to the *MsgDelay*/*Period* ratio. As observed above, this ratio is related to the maximum number of messages in transit at any time. On a single computer, Uppaal usually fails by running out of memory at a *MsgDelay*/*Period* ratio around .6. (Whether it runs out of memory also depends on the value of *TODelay*.) To get a better picture of what was happening, Skou also ran Uppaal on the Aalborg University 30-processor network. The results are tabulated in Figure 15.

N	$Period$	$MsgDelay$	$TODelay$	$\frac{MsgDelay}{Period}$	TLC	Uppaal	30-proc Uppaal
3	10	3	5	.3	255	9.4	2.9
	3	1	1	.33	4	9.4	13.4
	5	2	5	.5	70	11.2	2.9
	5	3	1	.6	13	30.8	3.0
	5	3	5	.6	265	fail	20.9
	3	2	1	.67	7	10.2	3.0
	3	2	2	.67	20	fail	16.6
	5	4	1	.8	27	32.5	9.2
	5	4	5	.8	980	fail	fail
	2	2	1	1	11	fail	fail
	1	2	1	2	270	fail	fail
	1	2	2	2	1280	fail	fail
4	10	3	5	.3	1385	42.2	2.5
	3	1	1	.33	6	43.9	2.7
	5	2	2	.4	42	48.3	4.2
	5	2	5	.4	390	93.0	4.3
	2	1	1	.5	6	48.2	3.7
	5	3	1	.6	28	72.8	3.8
	5	3	5	.6	1770	fail	84.6
	3	2	1	.67	12	73.1	9.8
	3	2	2	.67	44	fail	73.1
	5	4	5	.8	6760	fail	fail
	2	2	1	1	13	fail	fail
	1	2	1	2	390	fail	fail
	1	2	2	2	1650	fail	fail

Figure 15: Comparison of Uppaal and TLC execution times in seconds for the same graphs with 3 and 4 nodes as in Figure 14.

For *MsgDelay*/*Period* ratios significantly less than .6, Uppaal's execution time depends almost entirely on the graph and not on the other parameters. TLC's execution time depends on the magnitude of the parameters as well as on this ratio. Hence, if Uppaal succeeds, it is usually faster than TLC for small values of the parameters and much faster for larger values. Using 30 processors extends the range of parameters for which Uppaal succeeds. TLC can be run on multiple computers using Java's RMI mechanism. Tests have shown that using C computers typically speeds it up by a factor of about $.7C$. This suggests that, run on a network of processors, TLC's execution speed is comparable to Uppaal's for the range of instances tested. However, TLC will be slower than Uppaal for large enough values of the timing-constraint parameters.

The overall result is that Uppaal can check models with larger timing-constraint parameters, and hence with a finer-grained choice of ratios between the parameters. However, TLC can check a wider range of ratios among the parameters. For finding bugs, the ability to check parameter ratios of both 1:2 and 2:1 is likely to be more useful than the ability to check ratios of both 1:2 and 11:20.

Skou and his colleagues subsequently rewrote the Uppaal model to improve its performance. Because the TLA$^+$ specification was written to be as simple and elegant as possible, with no consideration of model-checking efficiency, the fairest comparison seems to be with the first, unoptimized Uppaal model. When checking the new model on a single computer, Uppaal fails on only four of the instances of Figure 15. It is an average of 4.5 times faster for the $N = 3$ instances and 50 times faster for the $N = 4$ instances. However, it still fails when *MsgDelay*/*Period* is greater than about 1. The new model therefore does not alter the basic result that Uppaal is faster than TLC for the range of parameter ratios it can handle, but it cannot handle as wide a range.

It is possible that these results reflect some special property of this example. However, the sensitivity to the *MsgDelay*/*Period* ratio suggests that it is the messages in transit that pose a problem for Uppaal. Each message carries a timer, and the performance of real-time model checkers tends to depend on the number of concurrently running timers. Perhaps the most common use of real time in systems is for timing constraints on message transmission—constraints that are modeled by attaching timers to messages. This suggests that Uppaal might have difficulty checking such systems if there can be many messages in transit. However, more examples must be tried before we can draw any such conclusion.

6.4.2 Fischer's Algorithm

Comparisons with Uppaal for Fischer's algorithm were made with a version of the algorithm described by a model distributed with Uppaal. It is similar to the version described by *FSpec*1, except that each thread has 4 control points instead of 6. The Uppaal version has no explicit liveness assumption, but Uppaal has a built-in assumption that time advances. The model has a single parameter K, related to the parameters of *FSpec*1 by $Epsilon = Delta = K + 1$.

The safety properties checked were mutual exclusion and absence of deadlock. The liveness property checked is that a process with control at statement b eventually reaches statement c. (This seems to be the only liveness property satisfied by the Uppaal model.) TLC and Uppaal were run as described above. For checking safety, TLC was run both with and without symmetry under permutations of threads. (The liveness property is not symmetric.)

Spin is an explicit-state model checker developed by Gerard Holzmann. The Spin model was written by Holzmann, who executed it on a 3 GHz, 3 GByte uniprocessor. For checking liveness, Spin uses a separate model that contains an extra process. This increases the number of states by a factor of 2.3–2.7, depending on the value of K.

SMV is a symbolic model checker, based on binary decision diagrams, that was developed by Ken McMillan. The SMV model was written by McMillan, who executed it on a 3 GHz uniprocessor with 2 GBytes of memory.

The representations of Fischer's algorithm used in all four model checkers are essentially the same. For both Spin and SMV, there are other ways to model the algorithm that can be checked more efficiently.

All the models were tested for 6 threads, which is the smallest number for which Uppaal takes a significant amount of time. The results for different values of K are shown in Figure 16. Checking for deadlock is essentially free for explicit-state model checkers, and TLC and Spin do it unless explicitly instructed not to. Uppaal and symbolic model checkers like SMV must be instructed to check for deadlocks. The SMV model was run without deadlock checking. About 49 seconds of Uppaal's execution times was spent checking for deadlock. TLC's liveness tests also checked for safety; checking only for liveness would reduce the execution times by a few percent.

Since Uppaal's execution time is independent of K, we know that it will be faster than a model checker whose running time depends on K. All of the model checkers could check the specification for large enough

		Safety				Liveness		
K	states	TLC^{s}	TLC	Spin	SMV	TLC	Spin	SMV
2	155976	9	29	.7	1.3	128	3.7	2.5
3	450407	10	78	2.4	3.8	385	13	6.3
4	1101072	16	194	6.9	6.5	1040	49	10
5	2388291	26	399	19	10	3456	171	16
6	4731824	47	784	51	14	5566	468	22
7	8730831	78	1468	142	25	13654	1317	40
8	15208872	132	2546	378	35		3593	54
9	25263947	244	4404	977	46		5237	73
10	40323576	446	7258	2145	62			95
Uppaal		82				135		

Figure 16: Execution times in seconds for a simple version of Fischer's algorithm with 6 threads, where TLC^{s} is TLC with symmetry under thread permutations.

values of K to provide reasonable confidence of its correctness, though the numbers do not bode well for the ability of TLC and Spin to check liveness for more complicated examples. We do not expect TLC's performance on liveness checking to be good enough for large applications. But because Fischer's algorithm is so simple, it would be dangerous to infer from these numbers that the performance of Uppaal and SMV would be good enough. For example, SMV does much better than Spin, even though explicit-state model checkers generally perform better than symbolic model checkers for this kind of asynchronous algorithm.

One observation from which we can generalize is the dependence of execution time on the number of reachable states. For TLC without symmetry, this dependence is linear because the time to compute possible next states and to evaluate the invariant is independent of K. We expect the same to be true for any explicit-state model checker. (Spin's execution time increased faster than the number of states because it was run with naive default settings; this should not occur in practice when run by a knowledgeable user.) SMV's execution time increases more slowly than the number of reachable states. Execution time for a symbolic model checker does not depend directly on the number of states, and I expect that it typically increases more slowly with increasing values of a specification's parameters than does the number of states.

When using symmetry under permutations of threads, TLC does quite

well, even running faster than Spin for larger values of K. (Spin and SMV cannot use symmetry under permutations in this way.) However, these results should not be taken very seriously. In real examples, symmetry sets usually contain only 2 or 3 elements, and the resulting speedups are much more modest.

It might also be a mistake to extrapolate from TLC's relatively poor performance on this simple example. The inefficiency of interpreting a specification may be less important than other factors for large problems. I know of only one case in which TLC and a more conventional model checker were applied to a large industrial specification—one that required days to check. The conventional model checker was Murphi [13], which also runs many times faster than TLC on small examples. On the large specification, the execution times of TLC and Murphi were comparable. Of course, the only conclusion we can draw from *that* example is that we should not try to draw conclusions from any one example.

7 Conclusion

7.1 Objections

I have encountered three objections to explicit-time specifications. The first is that they model system operations as happening at a particular instant of time. Some people feel that, since an operation takes a finite length of time, it should have a beginning and an ending time. But these people generally find nothing wrong with the usual practice of modeling the execution of an untimed system's operation as a single atomic event. We typically model execution of the statement $y := 0$ as one event, even though its actual execution involves voltages changing continuously over some interval of time. A discrete system is by definition one whose execution can be modeled as a sequence of atomic events. Discrete real-time systems are no different. If we need to distinguish between the starting and stopping times of the execution of an operation, we can model that execution by separate start and stop events.

The second objection to explicit-time specifications is that they express upper-bound timing constraints with a timer that prevents *now* from becoming too large. Some people feel that a timer should not be allowed to prevent time from advancing. They seem to think that a program's specification is *causing* the program's actions to occur. In fact, a specification causes nothing to happen. It just *describes* the program's possible executions. When describing allowed behaviors, saying that time is not allowed to

reach noon unless an event has occurred is completely equivalent to saying that the event must occur before noon. A specification does not express causality.

The third objection is that, because time advances in discrete steps, a behavior may skip over an error state—allowing us to prove correctness of an incorrect specification. For example, suppose correctness requires a property P to hold during some time interval I. We verify this by showing that $(\mathit{now} \in I) \Rightarrow P$ is an invariant of the system. A behavior could satisfy this specification by skipping over the interval I when P is false, so the value of *now* is never in I. This objection fails to take into account that proving invariance of $(\mathit{now} \in I) \Rightarrow P$ shows that it holds in all states of *all* behaviors. If there is a behavior in which *now* advances past the interval I in a single step, then there is also a behavior in which *now* advances past I in two steps, the first step assigning to it a value in I. In general, proving correctness in a model in which time advances in discrete steps is sufficient if time may advance in small enough steps.

7.2 Hybrid-System Specifications

A hybrid-system specification relates the behavior of a system to the values of physical quantities in its environment. A real-time specification is a special case of a hybrid-system specification in which time is the only relevant physical quantity. Like time, any physical quantity can be represented by an ordinary specification variable. The basic idea behind explicit-time specifications can therefore be applied as well to all hybrid-system specifications. The changes to variables representing physical quantities can be specified as solutions to differential equations [25]. Although the resulting TLA$^+$ specifications are straightforward [22], TLC may not be able to handle them unless the equations describing the evolution of the physical quantities are very simple—for example, if they are linear. (Most model checking algorithms for hybrid systems assume linear equations of evolution [17].)

7.3 Concluding Remarks

The main reason for using an explicit-time approach is to be able to use existing languages and tools, instead of having to develop new ones. There is no reason to develop new languages and tools unless they offer some advantages over existing ones. Implicit-time specifications are not inherently any easier to read or write than explicit-time ones. Nor are they any easier to reason about mathematically. One justification for implicit-time languages

is to take advantage of special algorithms for model checking real-time specifications. However, the results of Section 6 suggest that conventional model checking will work fairly well.

There are practical reasons for using a higher-level language like TLA$^+$ instead of one designed expressly for model checking. As one industrial user observed, "The prototyping and debug phase through TLA$^+$/TLC is so much more efficient than in a lower-level language."

Acknowledgements

I wish to thank Gerd Behrmann, Gerard Holzmann, Kim Larsen, Ken McMillan, and Arne Skou for the help they provided that is described in Section 6.4. I also wish to thank Tom Henzinger for his helpful comments; a number of them have found their way into my discussion of model checking. Lee Pike informed me of the prior work by Dutertre and Sorea, and Frits Vaandrager informed me of the work by Brinksma, Mader, and Fehnker.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[3] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Control*, 104(1):2–34, May 1993.

[4] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, London, Paris, Tokyo, Hong Kong, Barcelona, 1990.

[5] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[6] Brannon Batson and Leslie Lamport. High-level specifications: Lessons from industry. To appear in the Proceedings of the First International Symposium on Formal Methods for Components and Objects, held 5-8 November 2002 in Leiden, The Netherlands, 2003.

[7] Arthur Bernstein and Paul K. Harter, Jr. Proving real time properties of programs with temporal logic. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 1–11, New York, 1981. ACM. *Operating Systems Review 15*, 5.

[8] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[9] Ed Brinksma, Angelika Mader, and Ansgar Fehnker. Verification and optimization of a PLC control schedule. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):21–33, 2002.

[10] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Enginnering and Methodology*, 2(1):1–46, 1993.

[11] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts, 1988.

[12] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[13] David L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–393, 1996.

[14] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2004.

[15] Susanne Graf and Claire Loiseaux. Property preserving abstractions under parallel composition. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 644–657. Springer, 1993.

[16] Thomas A. Henzinger and Orna Kupferman. From quantity to quality. In Oded Maler, editor, *Proceedings of the International Workshop on*

Hybrid and Real-Time Systems (HART '97), volume 1997 of *Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, 1997.

[17] Thomas A. Henzinger and Rupak Majumdar. Symbolic model checking for rectangular hybrid systems. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of the Sixth International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Lecture Notes in Computer Science, pages 142–156. Springer-Verlag, 2000.

[18] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Control*, 111(2):193–244, June 1994.

[19] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[20] Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, Boston, 2004.

[21] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2(3):175–206, December 1982.

[22] Leslie Lamport. Hybrid systems in TLA$^+$. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102, Berlin, Heidelberg, 1993. Springer-Verlag.

[23] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[24] Leslie Lamport. Proving possibility properties. *Theoretical Computer Science*, 206(1–2):341–352, October 1998.

[25] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. A link to an electronic copy can be found at `http://lamport.org`.

[26] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, September 1994.

[27] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal of Software Tools for Technology Transfer*, 1(1/2):134–152, December 1997.

[28] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, January 1995. Technical Report CS-TR-95-03.

[29] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Symposium on the Principles of Distributed Computing*, pages 137–151. ACM, August 1987.

[30] Nancy Lynch and Frits Vaandrager. Forward and backward simulations ii. timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[31] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[32] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[33] Jonathan S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *The Journal of Systems and Software*, 18(1):33–60, April 1992.

[34] Jonathan S. Ostroff. Composition and refinement of discrete real-time systems. *ACM Transactions on Software Engineering and Methodology*, 8(1):1–48, January 1999.

[35] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.

[36] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[37] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended LAN. In *Proceedings of the Ninth Symposium on Data Communications*, pages 44–53. SIGCOMM, ACM Press, 1985.

[38] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[39] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.

[40] Fred B. Schneider. *On Concurrent Programming.* Graduate Texts in Computer Science. Springer, 1997.

[41] Fred B. Schneider, Bard Bloom, and Keith Marzullo. Putting time into proof outlines. In J. W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 618–639, Berlin, Heidelberg, New York, 1992. Springer-Verlag.

[42] Jiacun Wang. *Timed Petri Nets: Theory and Application.* Kluwer Academic Publishers, Boston, 1998.

[43] Sergio Yovine. KRONOS: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, December 1997.