Real-Time Model Checking is Really Simple

Leslie Lamport
Microsoft Research

13 June 2005

To be presented at Charme 2005, to be held 3-6 October 2005, in Saarbrücken, Germany.

This version is formatted differently, with different line and page breaks, from the version to appear in the Charme proceedings.

Contents

Real-Time Model Checking is Really Simple

Leslie Lamport
Microsoft Research

13 June 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 and expresses timing requirements with special timer variables. The resulting specifications can be verified with an ordinary model checker. This basic idea and some less obvious details are explained, and results are presented for two examples.

1 Introduction

Numerous special languages and logics have been proposed for specifying and verifying real-time algorithms. There is an alternative that I call the *explicit-time* approach, in which the current time is represented as the value of a variable *now* and the passage of time is modeled by a *Tick* action that increments *now*. Timing constraints are expressed with timer variables.

Hardly anything has been written about the explicit-time approach, perhaps because it is so simple and obvious. As a result, most people seem to believe that they must use special real-time languages and logics. It has already been shown that an explicit-time approach works fine for specifying and proving properties of real-time algorithms [1]. Here, I consider model checking explicit-time specifications.

The major advantage of the explicit-time approach is that it can be used with any language and logic for describing concurrent algorithms. This is especially important for complex algorithms that can be quite difficult to represent in the lower-level, inexpressive languages typical of real-time model checkers. For example, distributed message-passing algorithms have queues or sets of messages in transit, each with a bound on its delivery time. Such

algorithms are difficult or impossible to handle with most real-time model checkers. Section 2 briefly explains the explicit-time approach with a simple distributed algorithm. A complete specification of the algorithm in TLA$^+$ [8], a high-level mathematics-based language, appears in [9].

Explicit-time descriptions can use either continuous or discrete time. Section 3 shows that when discrete time is used, these descriptions can be checked with ordinary model checkers. This simple fact has been known for quite a while and is implicit in several published results [5]. However, a direct statement of it does not seem to have appeared before in print. Moreover, there are some aspects of model checking explicit-time specifications that may not be obvious, including the use of view symmetry and a method for checking that a specification is nonZeno [1].

Section 4 describes the result of checking the algorithm described in Section 2 with TLC, a model checker for TLA$^+$ specifications, and with Uppaal [10], the only real-time model checker I know of that can handle this example. It also compares TLC, Spin [6], and SMV [11] with Uppaal on the Fischer mutual exclusion algorithm [13]. More details appear in [9].

2 Writing Explicit-Time Specifications

In an explicit-time specification, time is represented with a variable *now* that is incremented by a *Tick* action. For a continuous-time specification, *Tick* might increment *now* by any real number; for a discrete-time specification, it increments *now* by 1. Timing bounds on actions are specified with one of three kinds of timer variables: a *countdown* timer is decremented by the *Tick* action, a *count-up* timer is incremented by *Tick*, and an *expiration* timer is left unchanged by *Tick*.[1] A countdown or count-up timer expires when its value reaches some value; an expiration timer expires when its value minus *now* reaches some value. An upper-bound timing constraint on when an action A must occur is expressed by an enabling condition on the *Tick* action that prevents an increase in time from violating the constraint; a lower-bound constraint on when A may occur is expressed by an enabling condition on A that prevents it from being executed earlier than it should be.

I illustrate how one writes explicit-time specifications using the example of a simple version of a classic distributed algorithm of Radia Perlman [12]. The original algorithm constructs a spanning tree rooted at the lowest-numbered node, called the *leader*. The tree is maintained by having the

[1] Dutertre and Sorea [3] use a different kind of timer variable that predicts the time at which an action will occur.

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

Figure 1: The *Tick* action's definition for the leader algorithm.

leader periodically propagate an *I'm Leader* message down it that informs each node of its distance to the leader. A new tree is constructed if a failure causes some node to time out before receiving the *I'm Leader* message. I have simplified it by eliminating failures, so correctness means simply that every node learns the leader within some fixed length of time. A complete TLA$^+$ specification of the algorithm appears in [9]. Here, I describe only the TLA$^+$ specification of the *Tick* action.

The algorithm has three timing parameters, *Period*, *MsgDelay*, and *TODelay*. Each node n has a countdown timer $\mathit{timer}[n]$. Setting $\mathit{timer}[n]$ to τ causes a timeout to occur between τ and $\tau + \mathit{TODelay}$ seconds later. By letting τ be the minimum timeout interval, this models both delay in reacting to a timeout and variation in the running rate of physical timers. When its timeout occurs, node n sends an *I'm Leader* message and sets $\mathit{timer}[n]$ to *Period*. If n receives an *I'm Leader* message from a lower-numbered node, it resets $\mathit{timer}[n]$ to a suitable value. A message is assumed to be received at most *MsgDelay* seconds after it is sent, a constraint enforced with a *rcvTimer* countdown timer field in the message. The algorithm achieves stability if, upon receiving a message from its leader, a node n sets $\mathit{timer}[n]$ to a value no smaller than $\mathit{Period} + \mathit{TODelay} + \mathit{dist}[n] * \mathit{MsgDelay}$, where $\mathit{dist}[n]$ is the distance from n to the leader.

Figure 1 contains the definition of the *Tick* action from the TLA$^+$ specification. It can't be completely understood without seeing the rest of the specification and having some knowledge of TLA$^+$ (including the definitions of the operators *BagToSet* and *BagOfAll* from the standard *Bags* module). However, it will indicate how timing constraints are specified and also give an idea of the high-level nature of TLA$^+$. This version is for a continuous-time specification, in which *now* is incremented by some real value d. We obtain

a discrete-time specification by replacing "$\exists d \in \{r \in \mathit{Real} : r > 0\} :$" with "LET $d \triangleq 1$ IN".

The action's first two conjuncts enforce the upper-bound constraints. The first prevents $\mathit{timer}[n]$ from becoming less than $-\mathit{TODelay}$, for each node n. The second prevents the timer $\mathit{ms.rcvTimer}$ from becoming negative, for all messages ms in the bag (multiset) msg of messages in transit.

The action's remaining conjuncts assert how the variables are changed. The third conjunct asserts that now is incremented by d. The fourth and fifth conjuncts assert that all the timers are decremented by d, the fourth for each $\mathit{timer}[n]$ and the fifth for the timer component $\mathit{ms.rcvTimer}$ of each message ms. The final conjunct asserts that the specification's other variables are unchanged.

The complete specification asserts the additional timing constraint that a timeout action of node n cannot occur before $\mathit{timer}[n]$ has counted down past 0. This constraint is expressed by the conjunct $\mathit{timer}[n] < 0$ in that action's definition.

3 Model Checking Explicit-Time Specifications

Most real-time system specifications are symmetric under time translation, meaning that system actions depend only on the passage of time, not on absolute time values. This section explains what symmetry and model checking under symmetry mean and describes a simple method of model checking explicit-time specifications that are symmetric under time translation.

3.1 Specifications and Temporal Properties

Let a *state* of a specification be an assignment of values to all the specification's variables, and let its *state space* be the set of all such states. 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.

Assume a specification $\mathcal{S}$ that consists of an initial predicate Init, a next-state action Next, and a liveness assumption L that is a temporal property, possibly equal to TRUE. The initial predicate and next-state action form the *safety part* $\overline{\mathcal{S}}$ of specification $\mathcal{S}$. A behavior $s_1, s_2, \ldots$ satisfies $\overline{\mathcal{S}}$ iff s_1 satisfies Init and $s_i \xrightarrow{\mathit{Next}} s_{i+1}$ for all i; it satisfies $\mathcal{S}$ iff it satisfies both $\overline{\mathcal{S}}$ and L.

3.2 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 $s \sim t$, 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 $s_2 \xrightarrow{A} t_2$.

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 symmetric if each of its component state predicates and actions is symmetric.

3.3 Model Checking

An explicit-state model checker works by computing the directed graph $\mathcal{G}$ of a specification $\mathcal{S}$'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 *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 $\overline{\mathcal{S}}$. Those behaviors that also satisfy its liveness assumption are the ones that satisfy $\mathcal{S}$.

The model checker constructs $\mathcal{G}$ by the following algorithm, using a set $\mathcal{U}$ of unexamined reachable states. Initially, $\mathcal{G}$ and $\mathcal{U}$ are both empty. The checker first sequentially enumerates the states satisfying *Init*, adding each state not already in $\mathcal{G}$ to both $\mathcal{G}$ and $\mathcal{U}$. It does the following, while $\mathcal{U}$ is nonempty. It chooses some state s in $\mathcal{U}$ and enumerates all states t satisfying $s \xrightarrow{Next} t$. For each such t: (i) if t is not in $\mathcal{G}$ then it adds t to $\mathcal{G}$ and to $\mathcal{U}$; (ii) if there is no edge from s to t in $\mathcal{G}$, then it adds one.

Model checking under a constraint P is performed by constructing a subgraph of $\mathcal{G}$ containing only states that satisfy the state predicate P. To compute the subgraph, this procedure is modified to add a state to $\mathcal{G}$ and $\mathcal{U}$ only if the state satisfies P.

Model checking under a symmetry $\sim$ consists of constructing a smaller graph $\mathcal{E}$ by adding a state to $\mathcal{E}$ and $\mathcal{U}$ only if $\mathcal{E}$ does not already contain an equivalent state. The graph $\mathcal{E}$ constructed in this way satisfies the following properties: (i) $s \not\sim t$ for every distinct pair of nodes s, t of $\mathcal{E}$; (ii) for every state s satisfying *Init*, there is a node t in $\mathcal{E}$ such that t satisfies *Init* and $s \sim t$; (iii) 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 specification is then checked as if $\mathcal{E}$ were the reachable-state graph.

Here, I ignore practical concerns and assume a theoretical model checker that can perform this algorithm even if the state graph is infinite. All the results apply *a fortiori* if the state graph is finite.

For model checking with symmetry to be equivalent to ordinary model checking, 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 $\overline{\mathcal{S}}$, just that they are equivalent to ones that satisfy $\overline{\mathcal{S}}$. Condition SS is true if the specification satisfies the following two properties:

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

S2. *Next* is symmetric.

The specification is defined to be *safety symmetric* iff it satisfies S1 and S2.

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

The simplest kind of temporal property is a state predicate P, which as a temporal formula asserts that P is true initially. 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.

3.4 View Symmetry

A view symmetry is defined by an arbitrary function on states called a *view*. 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. TLC supports view symmetry as well as symmetry under permutations of a constant set.

View symmetry is equivalent to abstraction [2, 4] 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 $\mathcal{A}$.

3.5 Symmetry Under Time Translation

Time-translation symmetry is a special kind of 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: (i) $T_d(s).now = s.now + d$, (ii) $T_0(s) = s$, and (iii) $T_{d+e}(s) = T_d(T_e(s))$. Specification $\mathcal{S}$ is defined to be *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. 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 $T_d(s)$. 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 $s.et - s.now$, which 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 $\langle v_1, \ldots, v_m, et_1 - now, \ldots, et_n - now \rangle$.

3.6 Periodicity and Zeno Behaviors

Let *NZ* be the temporal property asserting that time increases without bound. 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* [1]. 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. Thus, 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 as follows to use ordinary model checking to show that a specification is nonZeno.

Let $\mathcal{S}$ be a specification that is invariant under time translation. For simplicity, we 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.

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 $t.now \geq \lambda + 1$.

It can be shown that if a specification $\mathcal{S}$ is time-translation invariant, has a period less than or equal to the real number λ, and satisfies NZ_λ, then it is nonZeno. Therefore, we can check that $\mathcal{S}$ is nonZeno by verifying 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 for a model checker that, like TLC, 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.[2] This makes computing an upper bound on the period of $\mathcal{S}$ easy. 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 verifying the invariance of $now \leq \lambda$, using time-translation symmetry.

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 $now > \lambda + 1$. It is easy to verify NZ_λ under this constraint with a model checker that can check possibility properties. With one like TLC that checks only linear-time temporal properties, we must show that $\mathcal{S}$ together with fairness assumptions on subactions

[2] One of the features needed was added to TLC after publication of [8].

of its next-state action imply that the value of *now* must eventually reach $\lambda + 1$ [1, 7]. That is, we add fairness assumptions on certain actions and check that eventually $now \geq \lambda + 1$ holds, using the constraint $now \leq \lambda + 1$.

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 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.

4 Comparison with Uppaal

4.1 The Leader Algorithm

I have checked the TLA$^+$ specification of the leader algorithm with the TLC model checker. Although the specification is time-translation invariant, the correctness property is not. It asserts $(now > c(n)) \Rightarrow P(n)$ for each node n, where $c(n)$ is a constant expression and $P(n)$ does not contain *now*. We could add a timer variable and restate the property in terms of it. (This is what is done in the Uppaal model.) However, I instead had TLC check the property under a symmetry $\sim$ defined as follows. Let Σ be the maximum of $c(n)$ for all nodes n. Then $s \sim t$ iff $s.now$ and $t.now$ are both equal or both greater than Σ. Both the specification and the correctness property are symmetric under $\sim$. This symmetry is view symmetry under the view consisting of the tuple $\langle v_1, \ldots, v_k, \text{IF } now > \Sigma \text{ THEN } \Sigma + 1 \text{ ELSE } now \rangle$, where the v_i are all the variables except *now*.

Real-time model checkers use much lower-level modeling languages than TLA$^+$. Uppaal [10] is the only one I know of whose language is expressive enough to model this algorithm. Arne Skou, with the assistance of Gerd Behrmann and Kim Larsen, translated the TLA$^+$ specification to an Uppaal model. Since Uppaal's modeling language is not as expressive as TLA$^+$, this required some encoding. In particular, Uppaal cannot represent the potentially unbounded multiset of messages in the TLA$^+$ specification, so the Uppaal model uses a fixed-length array instead. To ensure that the model faithfully represents the algorithm, Uppaal checks that this array does not overflow.

TLC and Uppaal were run on different but roughly comparable machines. As indicated, some Uppaal executions were run on a 30-machine network. More detailed results are presented in [9].

The parameters of the specification are the number N of nodes, a constant operator that describes the graph, and the timing constants *Period*, *TODelay*, and *MsgDelay*. The latter two are upper-bound constraints, which implies that the number of reachable states is an increasing function of their values. Figure 2 shows the results of checking the correctness property on two different graphs, with 3 and 4 nodes, for some haphazardly chosen values of the timing bounds. Uppaal timings are given for a single machine and for the 30-machine network; *fail* means that Uppaal ran out of memory.

We expect that increasing a timing bound will increase the number of reachable states, and hence TLC's execution time, since it increases the number of possible values of the timer variables. The time required by Uppaal's algorithm depends only on the ratios of the timing bounds, not on their absolute value. The results show that Uppaal's execution time is strongly dependent on the ratio $MsgDelay/Period$. For 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 execution speed typically increases by a factor of about .7 times the number of computers. 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, since increasing the timing-constraint parameters increases the number of reachable states, TLC will be slower than Uppaal for large enough values of these 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.[3]

[3]The Uppaal model was subsequently rewritten to improve its performance. Because the TLA$^+$ specification was written to be as simple as possible, with no consideration of model-checking efficiency, the fairest comparison seems to be with the first Uppaal model.

$N = 3$ $N = 4$

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 2: Comparison of Uppaal and TLC execution times in seconds for the indicated graphs with 3 and 4 nodes.

Period	*MsgDelay*	*TODelay*	$\frac{MsgDelay}{Period}$	reachable 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

Figure 3: The number of messages in transit.

The dependence on the $MsgDelay/Period$ ratio can be explained as follows. Since *Period* is a lower bound on the time between the sending of messages and *MsgDelay* is an upper bound on how long it takes to deliver the message, the maximum number of messages that can be in transit at any time should be roughly proportional to this ratio. The table of Figure 3 gives some idea of what's going on, where the results are for the 3-node graph. 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.

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 constraints on message transit time—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.

TLC was also used to check that some of the instances in Figure 2 were nonZeno. For $N = 3$, this took about twice as long as checking the correctness property; for $N = 4$ the two times were about the same.

Uppaal can check the new model on a single computer an average of 4.5 times faster for the $N = 3$ instances of Figure 2 and 50 times faster for the $N = 4$ instances, but 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.

		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						135		

Figure 4: 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.

4.2 Fischer's Algorithm

I also compared the explicit-state approach to the use of Uppaal on a version of Fischer's mutual exclusion algorithm [13] that is distributed with Uppaal. Because TLA^+ is a very high-level language, TLC must "execute" a specification interpretively. It is therefore significantly slower than conventional model checkers for verifying simple systems. I also obtained data for two other popular model checkers whose models are written in lower-level languages: the explicit-state model checker Spin [6] and the symbolic checker SMV [11] that uses binary decision diagrams. The Spin model was written and checked by Gerard Holzmann, and the SMV model was written and checked by Ken McMillan. Checked were the safety properties of mutual exclusion and deadlock freedom (except for SMV) and a simple liveness property.

This version of Fischer's algorithm uses a parameter K that is both an upper- and lower-bound timing constraint. 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 4. Uppaal's execution time is independent of K. For checking safety, TLC was run both with and without symmetry under permutations of threads. (The liveness property is not symmetric.) The speedups obtained by the 6-fold symmetry should not be taken very seriously; in real examples one at best obtains only 2- or 3-fold symmetry.

Since Uppaal's execution time is independent of K, we know that for large enough values of K 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 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 is dangerous to infer from these numbers that the performance of Uppaal and SMV would be good enough.

5 Conclusion

Experts in the field will not be surprised that one can write and check explicit-time specifications using ordinary model checkers. But this is apparently not widely appreciated because it has not been stated clearly in the literature. Moreover, the use of view symmetry and the method described here for checking that a specification is nonZeno may be new even to experts.

I know of no previous comparisons of the explicit-state approach with the use of a real-time model checker. The results reported here do not tell us how the two methods will compare on other examples. But they do indicate that verifying explicit-time specifications with an ordinary model checker is not very much worse than using a real-time model checker. Indeed, the results for the leader algorithm suggest that the explicit-time approach is competitive with Uppaal for distributed algorithms. The results of using TLC to check two more complicated versions of Fischer's algorithm are reported in [9]. They too suggest that TLC can be used in practice to check explicit-time specifications.

The main advantage of an explicit-time approach is the ability to use languages and tools not specially designed for real-time model checking. There are practical reasons for using a higher-level language like TLA$^+$ instead of one designed expressly for model checking. As one industrial user remarked, "The prototyping and debug phase through TLA$^+$/TLC is so much more efficient than in a lower-level language."

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] Leslie Lamport. Real time is really simple. Technical Report MSR-TR-2005-30, Microsoft Research, March 2005.

[10] 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.

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

[12] 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.

[13] 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.