Formal Methods for Dependable Real-Time Systems*

Reprint of an invited paper presented at an International Symposium on Real-Time Embedded Processing for Space Applications organized by CNES, the French Space Agency, Les Saintes-Maries-de-la-Mer, France, 3–6 November, 1992, Published by Cépaduès-Éditions, Toulouse, France

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

We outline the motivation for using formal methods to specify and reason about real-time properties, and sketch some of the approaches that have been proposed and used.

1 Introduction

"Time is precious, but truth is more precious than time" [Fortune cookie].

1.1 Formal Methods

"Formal methods" are the use of mathematical techniques in the design and analysis of computer hardware and software. Engineers in established disciplines are fully familiar with the benefits that mathematical analysis can provide: for example, computational fluid dynamics (CFD) allows safe and relatively inexpensive exploration of alternative airfoil designs, and accurate prediction of the behavior of the selected design. Wind tunnel and flight tests are performed primarily to validate accuracy of the CFD model, and to measure final performance parameters. With computer

*The work described in the paper was partially supported by the National Aeronautics and Space Administration Langley Research Center under contracts NAS1-18266 and NAS1-18969 and by the US Naval Research Laboratory under contract N00014-92-C-2177.

software, however, prototyping and testing remain the principal methods for exploring designs and validating implementations. These are expensive and possibly dangerous, and can provide only partial coverage of the range of behaviors that a piece of software may exhibit.

The problem, of course, is that computer hardware and software are discrete systems: their behavior is determined by a succession of discrete state changes. The succession of states need not produce behavior that varies smoothly or continuously; instead it can exhibit discontinuities and abrupt transitions. The ability to select alternative courses of action is a source of the power and flexibility provided by computer systems, but it is also the reason why their behavior is hard to predict and to validate. Tests provide information on only the state sequences actually examined; without continuity there is little reason to suppose the behavior of untested sequences will be "close" to tested ones, and therefore little justification for extrapolating from tested cases to untested ones.

The fallibility of testing becomes particularly acute in the case of distributed and real-time systems, such as the embedded systems used for spacecraft control. Coordination of the various subsystems and redundant components can depend on delicate timing relationships that can be difficult to explore fully in tests. Consequently, a number of major spacecraft anomalies have been due to timing problems. For example, Magellan lost Earth lock several times in August 1990 (soon after entering Venus orbit)—events that "threatened to cause [it] to go into an uncontrollable sequence of spins which would have destroyed the spacecraft" [27]. It took over six months to identify the source of the problem, which was a timing error in the flight software.

The difficulty of predicting and testing timing behavior is compounded by the possibility that hardware faults may occur at any time. For example, some observations were lost when the heavy radiation environment at Jupiter caused one of the clocks on the Voyager spacecraft to jump several seconds [5]. More robust clock synchronization was provided for later encounters, but the mechanisms of fault tolerance and recovery are complex and compound still further the difficulty of validating behavior by testing. Thus, for example, the failure of the first attempt to launch the Space Shuttle was due to a synchronization problem among the redundant computers [12]. Experience with digital flight controls for aircraft indicates that poorly organized redundancy management (in particular, inadequate attention to synchronization issues) can lead to the fault-tolerance mechanisms becoming themselves the *primary* cause of system failure [22].

Formal methods confront the discrete behavior of computer systems by using *discrete mathematics* to model it. Discrete mathematics builds directly on mathematical logic, and proofs of theorems take the place of the numerical calculations that are familiar in most other engineering mathematics. That is, instead of using a mathematical model to calculate a value for some numerical quantity such as lift or drag, in formal methods for computer science we prove theorems about the modeled behavior—such as "for any behavior whatsoever on the part of faulty clocks, the following algorithm keeps the nonfaulty clocks synchronized, provided less than a third of the total are faulty."

In formal methods, the problem of discontinuity and the unsoundness of extrapolating from a finite number of tests are overcome using methods of proof based on mathematical induction, so that an infinite (or finite but very large) number of possible behaviors is fully covered in a finite proof.

Formal methods offer much more to computer science than just "proofs of correctness" for programs and digital circuits, however. Many of the problems in software and hardware design are due to imprecision, ambiguity, incompleteness, misunderstanding, and just plain mistakes in the statement of top-level requirements, in the description of intermediate designs, or in the specifications of components and interfaces. Some of these problems can be attributed to the difficulty of describing large and complex artifacts in natural language, but others may be due to the lack of a suitably precise conceptual framework for describing and reasoning about certain classes of behavior. Real-time behavior seems to fall into this latter class. With the exception of scheduling theory, there is relatively little foundation for an engineering discipline of real-time systems.

Formal methods attempt to supply precise techniques for specifying and reasoning about system behavior using techniques based, mainly, on the axiomatic method of mathematics. The basic idea of the axiomatic method is to specify properties required or assumed[1] as axioms in some formal language. These axioms, plus the rules of inference associated with the chosen language, supply everything that can be used in reasoning about the specified artifact. To show that the specification has some property not explicitly mentioned in the axioms, we must prove that that property is a logical consequence of the axioms. Thus a formal specification can be probed and tested by posing and proving putative theorems that we call *challenges*: "if this specification says what it should, then the following ought to follow." Similarly, to show that a design meets its requirements, we must prove that each requirement is derivable from the axioms that specify the design.

In a formal proof, everything we need to know about the particular problem area is encoded in the statement of the theorem and the axiomatization of its premises, and the truth of the theorem can be established by simply "pushing the symbols around" according to the rules of inference; the *form* of the proof determines its validity—it does not depend on unrecorded knowledge of the problem area, nor on

[1] Whether the specification is of requirements or assumptions depends on whether one is specifying an artifact to be constructed, or some external entity (such as a device, or the environment) with which it is to interact. Usually both are required simultaneously, as in formalizations of statements such as "this artifact is required to accomplish the following, assuming its environment behaves in the manner described..."

intuition about what the theorem says. Thus, checking a formal proof is a purely mechanical exercise that can, in principle, be done by a computer program, and with complete accuracy.

This is not to say that knowledge and intuition are superfluous in formal methods—on the contrary, they are essential to the invention of useful axiomatizations and theorems, to their interpretation and application in the real world, and to the *discovery* of proofs. Use of purely formal reasoning to *check* proofs, however, can be beneficial because it forces a complete enumeration of assumptions and inferences, and can catch faulty reasoning that would otherwise be overlooked. Even without mechanical checking, the "symbol-pushing" character of formal proofs can help prevent faults due to flawed intuition, jumping to conclusions, or simple carelessness. Above all, it is a *repeatable* exercise: others can check our work in greater detail than is feasible for informal reviews.

1.2 Real Time

A real-time system is one whose correctness depends not only on values of its outputs, but also on the times at which they are produced. Generally, a real-time system executes a collection of tasks subject to both *deadline* and *jitter* constraints: once activated (either by a timer, if it is a periodic task, or by some external event if it is aperiodic) a task is required to produce its outputs before its deadline, and with low variability (jitter) from one activation to another. Tight constraints on jitter arise in feedback control, where the stability of the transfer function may depend on consistent timing in the feedback loop, and in some systems dependent on I/O devices that require precise timing between inputs and outputs [20]. In some systems it can be important not to produce results too early, so that deadlines are often best treated as intervals, rather than points, in time. *Hard* real time refers to circumstances where a missed deadline is potentially catastrophic; for *soft* real-time constraints, there is still some (diminished) utility in the results of a task that has missed its deadline [36]. Thus, to be considered correct or useful, real-time systems must deliver results within specified time intervals, either without exception (hard real time), or with high probability (soft real time) [13].

There are two main issues in the development of real-time systems: the derivation of the timing constraints, and the construction of a system structure (particularly a scheduling regimen) that guarantees to satisfy those constraints. Traditionally, the second of these—especially scheduling theory—has received the most attention, but it can be argued that the first is the more fundamental.

A real-time system usually begins with some qualitative timing requirements, such as constraints on the simultaneity and ordering of events. For example, "cars and trains must not be on the railroad crossing at the same time," or "the valve must be closed before the hose is disconnected." Sometimes, quantitative constraints are present, too: for example, "close the valve, wait 10 seconds, and then check the pressure" or "the control loop must execute at a frequency of 40 Hz, with no more than 2 msec jitter on sensor sampling or actuator output."[2] In yet other cases, time may not be mentioned at all in the top level statements of requirements; for example, "achieve reliable communication over this channel." Here, explicit concern with time might arise only in the implementation, where timeouts could be used as part of a reliable data-transfer protocol.

Lutz and Wong [21] enumerate some of the scheduling constraints that can be required for spacecraft operations:

1. *Forbidden-combination constraint.* For example, "At most one of the two optics heaters can be on at one time."

2. *Minimum-interval constraint.* For example, "a *scan* command shall not be sent within 10 seconds of a *slew* command."

3. *Maximum-interval constraint.* For example, "A maximum of 6 minutes delay between spectrometer power-off and heater power-on can be tolerated."

4. *Inside-interval constraint.* For example, "Each low-gain antenna motor power-off command follows the power-on command no sooner than 9 seconds and no later than 30 seconds."

5. *Outside-interval constraint.* For example, "The time separation between powering on the S-band transmitter and powering on the X-band transmitter shall be either less than 30 seconds or greater than 6 minutes."

In all these cases, detailed timing requirements emerge as the top level requirements are elaborated into designs. For example, one way to avoid cars and trains occupying a railroad crossing at the same time is to lower a gate across the road shortly before a train arrives on the crossing. We can calculate how long it takes to lower the gate, and the minimum time that can elapse between a train passing a sensor and reaching the crossing, and from that we can deduce timing constraints on the gate controller. The specification problem at this level is to describe the combination of functional and timing constraints that the gate controller is to satisfy; the verification problem is to prove that this specification achieves the requirement that trains and cars are not present on the crossing simultaneously.

Once a detailed functional and timing specification is available, the next job is to implement it. The issues here are, firstly, how to organize a system implementation

[2] It can be argued that these are really derived requirements, following from qualitative higher-level requirements or constraints (e.g., "it takes a few seconds for the pressure sensor to stabilize," or "here are the equations for the control laws").

so that its timing behavior is predictable and, secondly, how to use that organization to guarantee the particular timing behavior required. The largest body of work in real-time systems concerns these issues, particularly the behavior of preemptive scheduling algorithms. There is some opportunity to apply formal methods in the analysis of these algorithms and their implementations. For example, Burns and Wellings [6] present a formal specification and informal proof that priority inheritance averts priority inversions. However, the larger opportunity appears to lie in the application of formal methods to the earlier problem: namely showing that functional and timing constraints (i.e., specifications) at the implementation level are sufficient to achieve higher level requirements.

For example, a problem that seems appropriate for formal study is that of showing that specific spacecraft command schedules satisfy sets of constraints such as those shown earlier. The more demanding problem considered by Lutz and Wong is to show that the constraints remain satisfied, even when detection of errors causes fault-protection and recovery schedules to be invoked in the middle of an ongoing command schedule.

2 Formal Specification of Real-Time Properties

Formal specification notations for timing constraints such as those of Lutz and Wong can be developed either from temporal logics, or from classical first- and higher-order logics.

Temporal logics provide two symbols (called modalities) $\Box$ and $\Diamond$ that range over a temporal ordering of states. $\Box A$ ("always" A) says that the temporal assertion A is true in all future states, and $\Diamond A$ ("eventually" A) asserts that A eventually becomes true at some future state. Pnueli [28] was the first to recognize that these logics could be used to reason about distributed computations and, in particular, about liveness properties. The distinction between liveness and safety properties is an important one in system modeling because the two types of properties require different specification and assurance techniques. A safety property stipulates that specific "bad things" do not happen during execution, while a liveness property stipulates that certain "good things" do happen (eventually). Formally, a safety property S is one such that for all execution traces τ not satisfying S, there is a prefix α of τ such that for no β does $\alpha \circ \beta$ satisfy S (where $\circ$ denotes concatenation of sequences). In other words, a safety property is one such that if a trace "goes bad," there is an identifiable point at which it happens [2]. A liveness property L is one such that for all α, there exists a β such that $\alpha \circ \beta$ satisfies L. In other words, a liveness property is one such that every trace can be extended to satisfy the property. It has been shown that any system property can be expressed as the conjunction of a safety property and a liveness property [2].

There are two families of temporal logics: linear time and branching time [16]. Both families have their adherents, and both have led to effective specification techniques. It is usually necessary to embellish the basic logics of either family with additional operators in order to achieve a comfortable degree of expressiveness; examples include "next state," "until," and backwards-time operators. Interval logics are temporal logics specialized for reasoning over intervals of activity [34]. The Temporal Logic of Actions (TLA) [19] is a temporal logic in which the modal operators are generalized from states to pairs of states (actions); it achieves considerable expressiveness with very little mechanism.

Despite their name, temporal logics do not provide ways to reason about "time" in a direct or quantitative (i.e., "real-time") sense. Several extensions to temporal logic have been proposed for reasoning about real-time properties. The simplest extension is Metric Temporal Logic (MTL) [15] where bounded versions of the temporal operators are introduced. Thus $\Box_{>3}A$ asserts that A is true in every state that is more than 3 time units in the future, and $\Diamond_{\leq 3}A$ asserts that A will eventually hold within 3 time units. A maximum interval constraint such as the one given earlier can be expressed in MTL as

$$\Box(S_{\mathrm{p\_off}} \supset \Diamond_{\leq 6} S_{\mathrm{h\_on}}),$$

where $S_{\mathrm{p\_off}}$ indicates "spectrometer power off" and $S_{\mathrm{h\_on}}$ "heater on" (and $\supset$ is the symbol for "implies").

The *explicit-clock* temporal logics such as Real-Time Temporal Logic (RTTL) [25] use a special variable T to indicate the value of time at any state and use first-order quantification over that variable to make real-time assertions. The previous example could be expressed in an explicit clock logic as

$$\forall x \Box((S_{\mathrm{p\_off}} \wedge x = T) \supset \Diamond(S_{\mathrm{h\_on}} \wedge T \leq x + 6))$$

(where $\forall$ means "for all," and $\wedge$ means "and"). The idea here is that x takes a "snapshot" of the clock in a state at which $S_{\mathrm{p\_off}}$ is true, so that the right hand side of the implication can say that there must eventually be a state in which $S_{\mathrm{h\_on}}$ is true and the clock has advanced at most 6 minutes.

Mixing first-order quantification (i.e., $\forall$ and $\exists$) with the temporal modalities can lead to complexity. Timed Propositional Temporal Logic (TPTL) [3] is an extension to temporal logic that provides just enough mechanism to take "snapshots" of the (implicit) clock, without adding the full power of first-order quantification. In TPTL, the temporal modalities bind a variable that "freezes" the current value of time, which can then appear in inequalities—so that our example then appears as:

$$\Box x.S_{\mathrm{p\_off}} \supset \Diamond y.S_{\mathrm{h\_on}} \wedge y \leq x + 6.$$

Alur and Henzinger [4] survey these and many other real-time variants of temporal logic; they identify six dimensions of choice that must be made in creating such a logic, and describe current knowledge concerning expressiveness and decidability for various choices. It seems that no single logic is uniformly superior to the others (for example, there are properties that are much more difficult to specify in MTL than TPTL, and also vice-versa). Their survey does not include interval temporal logics, which have also been extended to real-time systems [9, 24].

Classical logics can also be used to reason about time. Jahanian and Mok's Real-Time Logic (RTL) [14] was one of the earliest attempts; it uses an occurrence operator $@(i, e)$ to denote the time of the i'th occurrence of event e, so that our example would appear as

$$\forall i : @(i, S_{\text{p\_off}}) + 6 \geq @(i, S_{\text{h\_on}}).$$

A graphical framework for state machine specifications called "modecharts" has been based on RTL [13], and a verifier allows certain real-time properties to be proved of such specifications [37]. However, the examples that have been subjected to mechanically-supported analysis in this way are relatively straightforward.

3 Formal Verification of Real-Time Properties

The only genuinely difficult examples of real-time reasoning that have been subjected to mechanized analysis are fault-tolerant clock synchronization algorithms. The goal of these algorithms is to maintain the clocks of the nonfaulty redundant channels of a fault-tolerant control system in approximate synchrony, despite arbitrary failures among the faulty channels. In reasoning about these algorithms, time is modeled in a very application-specific way; $c_p^{(i)}(T)$ denotes the "real" time (a rational number) at which channel p's clock reads T (another number) during the i'th frame. The synchronization requirement is then stated as

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta$$

for nonfaulty channels p and q. The idea is that if a certain action is to be triggered when a channel's clock reads T, then we want the different channels to perform the action within δ real-time units of each other.

Lamport and Melliar-Smith's Interactive Convergence Algorithm (ICA) solves this clock-synchronization problem, provided less than a third of the channels are faulty [17]. ICA assumes that all nonfaulty channels have reasonably accurate clocks with a maximum rate of drift from real time given by ρ, and are synchronized within some bound δ_0 initially. The slight differences in their running rates will cause the clocks gradually to drift apart, so that they must be resynchronized periodically. Each channel resynchronizes by determining the differences between its own clock and those of other channels, forming a "fault-tolerant average" of those differences, and adjusting its own clock by that amount. Trivial solutions, such as those in which all clocks are reset to zero at every resynchronization must be excluded, so there is a small bound Σ on the size of the adjustment that may be made at each resynchronization. Determining the differences between their clock readings may require cooperation among the channels, so they must all engage in the synchronization protocol at approximately the same time. To do this, each channel engages in the synchronization protocol every R seconds, and for a duration of S seconds, according to its own clock. Channels may not be able to determine the differences between their clocks with absolute accuracy: the quantity ϵ is assumed to bound the error in $\Delta_{qp}^{(i)}$, which denotes the difference between the clocks of channels q and p, as determined by p during its ith resynchronization.

The key to making a clock synchronization algorithm resistant to Byzantine failures among its components is the use of a "fault-tolerant average" in the adjustment step [33]. ICA is characterized by use of the *egocentric mean* as its averaging function. To compute the egocentric mean, a channel replaces all differences $\Delta_{qp}^{(i)}$ larger than a fixed quantity Δ by zero, and then calculates the arithmetic mean of the resulting set of differences.

To gain an idea of why this works, consider two nonfaulty channels p and q. For simplicity, assume that these channels perform their synchronization calculations simultaneously and instantaneously. If r is also a nonfaulty channel, then the estimates that p and q form of r's clock value can differ by at most 2ϵ. If r is a faulty channel, however, p and q could form estimates of its clock value that differ by as much as $2\Delta + \delta$ (since r could indicate a value as large as Δ different from each of p and q without being disregarded, and these channels could themselves have clocks that are δ apart). Assuming there are n channels, of which m are faulty, the egocentric means formed by p and q can therefore differ from each other by as much as

$$\frac{(n-m)2\epsilon + m(\delta + 2\Delta)}{n}.$$

Thus, provided

$$\delta \geq 2\epsilon + \frac{2m\Delta}{n-m},$$

this procedure will maintain the clocks of p and q within δ of each other, as required.

The sketch we have just given neglects many important details (for example, the channels do not perform the algorithm simultaneously and instantaneously, and their clocks may drift further apart in the periods between resynchronizations). Since the clock synchronization algorithm is a potential source of single point failure in fault-tolerant control systems, we require an unusually high degree of confidence in the analysis of ICA. In 1988, we formally specified and verified ICA using our verification system EHDM [31]. In doing so, we discovered that the proofs or statements of four of the five lemmas, and the proof of the main synchronization theorem, were flawed in the published journal presentation [17]. In developing our mechanically-checked verification we eliminated the approximations used by Lamport and Melliar-Smith and streamlined the argument. We were able to derive a journal-style presentation from our mechanized verification that is not only more precise than the original, but is simpler, more uniform, and easier to follow [29,30]. Our mechanized verification took us a couple of months to complete and required about 200 lemmas (most of which are concerned with "background knowledge" such as summation and properties of the arithmetic mean that are assumed in informal presentations); EHDM requires about 10 minutes on a Sun SparcStation 2 to check all the proofs mechanically.

There are many clock synchronization algorithms apart from ICA. Schneider gives a uniform treatment that includes almost all the known algorithms [33]. Our colleague Shankar has performed a mechanically checked verification of Schneider's analysis [35]. As with our verification of ICA, he found some errors in the original analysis.

As noted, we used an application-specific representation of time in reasoning about these algorithms. These representations are easily described in the higher-order logic that underlies the specification language of EHDM. Indeed, higher-order logic with arithmetic is so expressive that temporal logics, and their real-time variants, can also be defined within it. The advantage of embedding such formalisms within a single logic is that it is easier to combine them with others and easier to share common theories, such as datatypes, arithmetic, and other prerequisite mathematics. This becomes important when real-time properties must be combined with others. For example, one purpose of clock synchronization is to support an approach to fault tolerance and transient recovery based on exact-match majority voting. To show that such an approach works, we need to reason about fault tolerance (and hierarchical development) as well as clock-synchronization. A propositional real-time temporal logic is insufficiently expressive to deal with such larger challenges, whereas Butler and Di Vito were able to accomplish this example relatively straightforwardly in EHDM [11].

Because numbers provide the underlying representation for real time (either the natural numbers or the integers for discrete time, and the rationals or reals for dense time), mechanized proof checking of arguments involving real time ultimately requires facility in arithmetic—particularly addition and subtraction (for accumulating delays) and inequality chaining (for dealing with bounds and intervals). The logical theory known as Presburger Arithmetic[3] is ideal for this purpose, since it is decidable—this means there is an algorithm that can tell whether any such expression is valid or not. The theorem prover of EHDM includes a decision procedure for Presburger arithmetic and this was essential to our ability to complete the formal

[3]That is arithmetic with constants and variables, addition and subtraction, but with multiplication restricted to the linear case (i.e., multiplication by literal constants only), together with the relations $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$.

verifications of clock synchronization algorithms described earlier; it would be tedious beyond the point of feasibility to undertake such verifications using systems that provide no built-in automation of arithmetic reasoning.

The apparent advantage of the real-time temporal and first-order logics that have been proposed is that they are optimized to capture the relevant characteristics of real-time reasoning. In many circumstances, however, this advantage is not realized. Many real-time applications have a cyclic structure, so that the natural way to argue about their correctness is in terms of the last time something happened, or the next time that it will occur. The real-time logics, on the other hand, treat time as a linear structure, with no special treatment for "this time" and "last time." A symptom of this mismatch is that formal proofs of quite simple real-time properties can become excessively long, tedious, and inperspicuous in real-time logics. Because they concern deadlines or allowable intervals, most real-time requirements are safety properties (for example, all six of Lutz and Wong's requirements are safety properties). The natural way to prove safety properties is by an inductive invariance argument; that is by assuming the property holds in some state (and was true in the initial state), prove that it is true in its successor state. There is a choice whether "some state" should be "now" and its successor therefore the "next" state, or whether the successor should be "now," so that "some state" is the previous state. It seems generally more convenient to reason about the time that has elapsed since the last time a property held (as opposed to the delay until it will next hold), so the latter of these alternatives seems the most attractive.

Much recent work that incorporates real time into logics for reasoning about computations follows this approach. Abadi and Lamport present what they call "an old-fashioned recipe for real time" [1] where they employ a process that advances time in discrete increments to model time, and counter variables are used to bound certain time intervals. Schneider, Bloom, and Marzullo [32] have extended "Proof Outline Logic" to handle real time. The logic contains control predicates to indicate where the control is in a program. They employ an operator that records the time when a control predicate last became true. Carruth and Misra [7] employ a similar extension to Unity [8] where for any assertion p, $\bar{p}$ ("punch p") records the absolute time (i.e., "punches the clock") when p last went from being false to true. In this approach, $\bar{p}$ is initially equal to the time in the initial state if p is true in the initial state, otherwise, $\bar{p}$ is some negative value (since time only ranges over nonnegative values). Maler, Manna, and Pnueli [23] introduce yet another interesting operator called the duration operator δ, where for a temporal formula ϕ, the value of $\delta(\phi)$ at any state s in an execution of a program yields the largest time duration ending in s for which ϕ has continuously held. If ϕ is false at s, then $\delta(\phi)$ is 0.

Our colleague Shankar has developed an approach along these lines that uses a "since" operator, denoted by vertical bars: at any state in the program behavior, $|p|$ records the time that has elapsed since the last state in the past at which p was true. If p was never true in the past $|p| = \infty$. Since $|p|$ is concerned only about the past, its value is unaffected by whether p is true *now*.

As an example of real-time reasoning using this operator, we consider Fischer's real-time mutual exclusion algorithm [18]. This example is something of a test piece in real-time reasoning, since it is not obvious that the algorithm works. The idea is that several processes, each identified by a unique number, share a state variable x, which is initially 0. Whenever any process wishes to enter its critical section, it waits until it sees that $x = 0$. With a delay at most hi, it then sets x to its own number and goes to the "check" action. In the check action, it waits for at least lo units and then looks to see if the value of x is still its own number (some other process could have previously seen $x = 0$ and be in a race with this processor); if it is, the process immediately enters its critical section (otherwise it goes back to waiting for x to return to 0). On leaving its critical section, the process resets x to 0. The claim is that this algorithm achieves mutual exclusion (i.e., no two processes are ever in their critical sections simultaneously), provided $hi < lo$. Notice that the specification for this algorithm (i.e., mutual exclusion) is a pure safety property, not involving real time. Real-time reasoning is needed only to show correctness of the algorithm.

We give an informal verification of a simplified version of the algorithm that neglects the looping behavior: that is, we consider processes that simply stall at the check action if they find x different to their own number. The three steps in the algorithm can be described as follows, where i is the number of the process concerned.

init: if $x = 0$, goto try;

try: $x := i$, goto check;

check: if $x = i$, enter critical section.

We write $try(i)$ to indicate that the i'th process is executing the *try* action, and so on for the other actions; $cs(i)$ means that process i is in its critical section. Using the *since* operator, the two timing constraints can be expressed as follows.

1. $\forall i : try(i) \supset |init(i)| \leq hi$.
   When process i is executing its *try* action, the time since it was in its *init* action is at most hi.

2. $\forall i : cs(i) \supset |check(i)| + lo \leq |try(i)|$.
   When process i is in its critical section, the time since it was in its *try* action exceeds the time when it was last in its *check* action by at least lo.

The correctness requirement is

$$cs(i) \wedge cs(k) \supset i = k.$$

The argument for correctness follows from four invariants.

1. Whenever $cs(k)$, the time since $try(k)$ is at most the time since $x = 0$. This follows because the *try* action sets x to k at the same time it enters the *check* action.

2. Whenever $try(i)$, the time since $x = 0$ is at most the time since $init(i)$. This is because $x = 0$ is a precondition for the action at *init*.

3. Whenever $cs(k)$, then $\neg try(i)$. Otherwise the time since $x = 0$ is at most *hi* (by Invariant 2 and Timing Constraint 1) and at least *lo* (by Invariant 1 and Timing Constraint 2), contradicting $hi < lo$.

4. Whenever $cs(k)$, then $x = k$. Only the action $try(i)$ could falsify $x = k$, but Invariant 3 rules this out.

Shankar has formalized the *since* operator in our PVS verification system [26] and has mechanically checked the correctness argument given above for Fischer's mutual exclusion algorithm, and also the timing constraints on a simple railroad crossing example. These mechanically-checked formal verifications are little longer than a comparably detailed informal argument.

4 Conclusion

The formal verifications of clock synchronization algorithms show that mechanically-supported reasoning about complex real-time behavior is feasible, though something of a *tour-de-force*. However, there has been significant increase in the effectiveness of verification systems since those verifications were performed, and we can expect verifications of comparable difficulty to become fairly routine within another couple of years.

The current challenge, then, lies not so much in increasing the power or effectiveness of the deductive apparatus needed for mechanically-supported verification of real-time properties, but in developing perspicuous and economical approaches to the formalization and specification of such properties. The last few years have seen a surge of interest in applying formal methods to real-time problems, and many new and interesting approaches have been proposed. It is too early to identify which of these will be most successful; what is needed are experimental applications of the most promising approaches to problems of significant difficulty and interest.

Real-time properties, along with fault tolerance, are among the most interesting and potentially valuable targets for formal methods since they tend to be mission and safety-critical, yet hard to analyze by informal arguments, and difficult to validate by testing. The next few years promise much progress in formal methods for these applications, and we may expect formal methods to provide real benefit to system developments in these areas before too long.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In de Bakker et al. [10], pages 1—27.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] R. Alur and T. A. Henzinger. A really temporal logic. In *30th IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[4] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In de Bakker et al. [10], pages 74—106.

[5] Anonymous. Reprogramming capability proves key to extending Voyager 2's journey. *Aviation Week and Space Technology*, page 72, August 7, 1989.

[6] A. Burns and A. J. Wellings. Priority inheritance and message passing communication: A formal treatment. *Real-Time Systems*, 3(1):19–44, March 1991.

[7] John Alen Carruth and Jayadev Misra. Proof of a real-time mutual-exclusion algorithm. Notes on UNITY 32–92, Department of Computer Sciences, University of Texas at Austin, Austin, TX, September 1992.

[8] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[9] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[10] 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*, REX Workshop, Mook, The Netherlands, June 1991. Springer Verlag.

[11] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In *3rd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 85–97, Mondello, Sicily, Italy, September 1992. IFIP WG 10.4. Preprint proceedings.

[12] John R. Garman. The "bug" heard 'round the world. *ACM Software Engineering Notes*, 6(5):3–10, October 1981.

[13] C. L. Heitmeyer, B. G. Labaw, P. C. Clements, and A. K. Mok. Engineering CASE tools to support formal methods for real-time software development. In *Fifth International Workshop on Computer-Aided Software Engineering*, pages 110–113, Montreal, Quebec, Canada, July 1992. IEEE Computer Society.

[14] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[15] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.

[16] L. Lamport. Sometime is sometimes not never. In *10th ACM Symposium on Principles of Programming Languages*, pages 174–185, Austin, TX, January 1983.

[17] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[18] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[19] Leslie Lamport. The temporal logic of actions. Technical Report 57, DEC Systems Research Center, Palo Alto, CA, April 1990. A substantially modified version is available dated January 1991.

[20] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[21] Robyn R. Lutz and Johnny S. K. Wong. Detecting unsafe error recovery schedules. *IEEE Transactions on Software Engineering*, 18(8):749–760, August 1992.

[22] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[23] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In de Bakker et al. [10], pages 447—484.

[24] P. M. Melliar-Smith. Extending interval logic to real time systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, pages 224–242, Altrincham, UK, April 1987. Volume 398 of *Lecture Notes in Computer Science*, Springer Verlag.

[25] Jonathan S. Ostroff. A logic for real-time discrete event processes. *IEEE Control Systems Magazine*, 10(4):95–102, June 1990.

[26] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, June 1992. Volume 607 of *Lecture Notes in Artificial Intelligence*, Springer Verlag.

[27] James F. Peters, III and Sheela Ramanna. Verifying command sequences for satellite systems. *IEEE Aerospace and Electronic Systems Magazine*, 7(10):14–21, October 1992. See corrections of "substantial errors of fact," in Vol. 8, No. 4, April 1993, page 40.

[28] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.

[29] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239, June 1989.

[30] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *SIGSOFT '91: Software for Critical Systems*, pages 1–15, New Orleans, LA, December 1991. Expanded version to appear in *IEEE Transactions on Software Engineering*, 1993.

[31] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[32] F. B. Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. In de Bakker et al. [10], pages 618—639.

[33] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[34] R. L. Schwartz, P. M. Melliar-Smith, and F. Vogt. An interval logic for higher-level temporal reasoning. In *ACM Symposium on Principles of Distributed Computing*, pages 173–186, August 1983.

[35] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 217–236, Nijmegen, The Netherlands, January 1992. Volume 571 of *Lecture Notes in Computer Science*, Springer Verlag.

[36] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, November 1990. (Editorial).

[37] Douglas A. Stuart. Implementing a verifier for real-time systems. In *Real Time Systems Symposium*, pages 62–71, Lake Buena Vista, FL, December 1990. IEEE Computer Society.