Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms

John Rushby, *Member*, *IEEE*

Abstract—Many critical real-time applications are implemented as time-triggered systems. We present a systematic way to derive such time-triggered implementations from algorithms specified as functional programs (in which form their correctness and fault-tolerance properties can be formally and mechanically verified with relative ease). The functional program is first transformed into an untimed synchronous system and, then, to its time-triggered implementation. The first step is specific to the algorithm concerned, but the second is generic and we prove its correctness. This proof has been formalized and mechanically checked with the PVS verification system. The approach provides a methodology that can ease the formal specification and assurance of critical fault-tolerant systems.

Index Terms—Formal methods, formal verification, time-triggered algorithms, synchronous systems, PVS.

1 Introduction

Synchronous systems are distributed computer systems where there are known upper bounds on the time that it takes nonfaulty processors to perform certain operations and on the time that it takes for a message sent by one nonfaulty processor to be received by another. The existence of these bounds simplifies the development of fault-tolerant systems because nonfaulty processes executing a common algorithm can use the passage of time to predict each others' progress. This property contrasts with asynchronous systems, where there are no upper bounds on processing and message delays and where it is therefore provably impossible to achieve certain forms of consistent knowledge or coordinated action in the presence of even simple faults [1], [2].

For these reasons, fault-tolerant systems for critical control applications in aircraft, trains, automobiles, and industrial plants are usually based on the synchronous approach, though they differ in the extent to which the basic mechanisms of the system really do guarantee satisfaction of the synchrony assumption.

With systems based on conventional "commercial off the shelf" (COTS) components, synchrony is merely an assumption—these systems employ scheduling algorithms that can miss deadlines, their operating systems admit the possibility of buffer overflows, they use contention buses such as Ethernet, and they have other characteristics that allow occasional violations of claimed time bounds. Violation of the synchrony assumption may lead to failure of the higher-level system components that depend on it, so adopting this assumption when it is only probabilistically valid has ramifications on overall system reliability. Notice that adding timeouts does not make an asynchronous system synchronous [3].

While probabilistic satisfaction of the synchrony assumption may be "good enough" for less critical applications, those that are truly critical must either rest on weaker assumptions, or must be specially constructed to ensure that the assumption is unconditionally valid. Those that take the latter course often build on mechanisms that are not merely synchronous, but synchronized and time-triggered: The clocks of the different processors are kept close together, processors perform their actions at specific times, and tasks and messages are globally and statically scheduled. The buses and operating systems used in these contexts are specialized and dedicated to satisfaction of the synchrony hypothesis [4]. The Honeywell SAFEbus™ [5], [6] that provides the safety-critical backplane for the Boeing 777 Airplane Information Management System (AIMS) [7], [8], the control system for the Shinkansen (Japanese Bullet Train) [9], and the Time-Triggered Protocol (TTP) for safety-critical automobile functions [10] all use this approach.

A number of basic functions have been identified that provide important building blocks in the construction of fault-tolerant synchronous systems [11], [12]; these include consensus (also known as interactive consistency and Byzantine agreement) [13], reliable and atomic broadcast [14], and group membership [15]. Numerous algorithms have been developed to perform these functions and, because of their criticality and subtlety, several of them have been subjected to detailed formal [16], [17], [18] and mechanically checked [19], [20], [21], [22], [23] verifications, as have their combination into larger functions such as diagnosis [24], and their synthesis into a fault-tolerant architecture based on active (state-machine) replication [25], [26].

Formal, and especially mechanically-checked, verification of these algorithms is still something of a *tour de force*, however. To have real impact on practice, we need to reduce the difficulty of formal verification in this domain to a routine and largely automated process. In order to achieve

• *J. Rushby is with the Computer Science Laboratory, SRI International, Menlo Park CA 94025. E-mail: Rushby@csl.sri.com.*

Manuscript received 15 Aug. 1998; revised 10 Feb. 1999.
Recommended for acceptance by W.H. Sanders.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109547.

this, we should study the sources of difficulty in existing treatments and attempt to reduce or eliminate them. In particular, we should look for opportunities for systematic treatments: These may allow aspects common to a range of algorithms to be treated in a uniform way and may even allow some of those aspects to be broken out and verified in a generic manner once and for all.

There is a wide range in the apparent level of difficulty and detail in the mechanized verifications cited above. Some of the differences can be attributed to the ways in which the problems are formalized or to the different resources of the formal specification languages and theorem provers employed. For example, Rushby [19] and Bevier and Young [23] describe mechanically checked formal verifications of the same "Oral Messages" algorithm [27] for the consensus problem that were performed using different verification systems. Young [28] argues that differences in the difficulty of these treatments (that of [19] is generally considered simpler and clearer than that of [23]) are due to choices in the way things are formalized, and not to the capabilities of the tools employed. We may assume that such differences will be reduced or eliminated as experience is gained and the better choices become more widely known.

More significant than differences due to *how* things are formalized are differences due to *what* is formalized and the level of detail considered necessary. For example, both verifications of the Oral Messages algorithm mentioned above specify the algorithm as a functional program and the proofs are conventional inductions. Following this approach, the special case of a two-round algorithm (a variant of the algorithm known as OM(1)) is specified in [22] in a couple of lines and its verification is almost completely automatic. In contrast, the treatment of OM(1) in [18] is long and detailed and quite complicated. The reason for its length and complexity is that this treatment explicitly considers the distributed, message, passing character of the intended implementation and calculates tight real-time bounds on the timeouts employed. All these details are abstracted away in the treatments using functional programs—but this does not mean these verifications are inferior to the more detailed analyses: On the contrary, I would argue that they capture the essence of the algorithms concerned (i.e., they explain *why* the algorithm is fault, tolerant) and that message-passing and real-time bounds are implementation details that ought to be handled separately. In fact, most of the papers that introduce the algorithms concerned, and the standard textbook [29], use a similarly abstract and time-free treatment. On the other hand, it is undeniably important also to verify a specification that is reasonably close to the intended implementation and to establish that the correct timeouts are used and that the concrete fault modes match those assumed in the more abstract treatment.

The natural resolution for these competing claims for abstractness and concreteness is a hierarchical approach in which the essence of the algorithm is verified in an abstract formulation and a more realistic formulation is then shown to be a refinement, in some suitable sense, of the abstract formulation. This may not always be possible (e.g., for event-based systems) but, when it is, we may hope that the refinement argument will be a routine calculation of timeouts and other concrete details.

The purpose of this paper is to present such a hierarchical treatment for the important case of time-triggered implementations of round-based algorithms and to show that most of the details of refinement to a concrete formulation can be worked out once and for all.

2 Round-Based Algorithms

In her textbook [29], Lynch identifies algorithms for the synchronous system model with those that execute in a series of "rounds." Rounds have two phases: In the first, each processor[1] sends a message to some or all of the other processors (different messages may be sent to different processors; the messages depend on the current state of the sending processor); in the second phase, each processor changes its state in a manner that depends on its current state and the collection of messages it received in the first phase. There is no notion of real-time in this model: Messages are transferred "instantaneously" from senders to recipients between the two phases. The processors operate in lockstep: All of them perform the two phases of the current round, then move on to the first phase of the next round, and so on.

Several of the algorithms of interest here were explicitly formulated in terms of rounds when first presented, and others can easily be recast into this form. For example, the Oral Messages algorithm for consensus, OM(1), requires two rounds as follows.

Algorithm OM(1)

Round 0:

Communication Phase: A distinguished processor called the *transmitter* sends a value to all the other processors, which are called *receivers*; the receivers send no messages.

Computation Phase: Each receiver stores the value received from the transmitter in its state.

Round 1:

Communication Phase: Each receiver sends the value it received from the transmitter to all the other receivers; the transmitter sends no message.

Computation Phase: Each receiver sets the "decision" component of its state to the majority value among those received from the other receivers and that (stored in its state) received from the transmitter.

In the presence of one or fewer arbitrary faults, OM(1) ensures that all nonfaulty receivers decide on the same value and, if the transmitter is nonfaulty, that value is the one sent by the transmitter.

There are two different ways to implement round-based algorithms. In the *time-triggered* approach, the implementation is very close to the model: The processors are closely synchronized (e.g., to within a couple of bit-times in the case of SAFEbus) and all run a common, deterministic

1. I refer to the participants as processors to stress that they are assumed to fail independently; the agents that perform these actions will actually be processes.

schedule that will cause them to execute specific algorithms at specific times (according to their local clocks). The sequencing of phases and rounds is similarly driven by the local clocks and communication bandwidth is also allocated as dedicated, fixed, time slots. The first (communication) phase in each round must be sufficiently long that all nonfaulty processors will be able to exchange messages successfully; consequently, no explicit timeouts are needed: A message that has not arrived by the time the second (computation) phase of a round begins is implicitly timed out.

Whereas the allocation of resources is statically determined in the time-triggered approach, in the other, *event-triggered*, approach, resources are scheduled dynamically and processors respond to events as they occur. In this implementation style, the initiation of a protocol may be triggered by a local clock, but subsequent phases and rounds are driven by the arrival of messages. In Lamport and Merz' treatment of OM(1), for example, a receiver that has received a message from the transmitter may forward it immediately to the other receivers without waiting for its clock to indicate that the next round has started (in other words, the pacing of phases and rounds is determined locally by the availability of messages). Unlike the time-triggered approach, messages may have to be explicitly timed out in the event-triggered approach. For example, in Lamport and Merz' treatment of OM(1), a receiver will not wait for relayed messages from other receivers beyond $2\delta + \epsilon$ past the start of the algorithm (where δ is the maximum communication delay and ϵ the maximum time that it can take a receiver to decide to relay a message).

Event-triggered systems are generally easier to construct than time-triggered ones (which require a big planning and scheduling effort upfront) and achieve better CPU utilization under light load. On the other hand, Kopetz [4], [10], [30] argues persuasively that time-triggered systems are more predictable (and, hence, easier to verify), easier to test, easier to compose together, make better use of broadcast communications bandwidth, can operate closer to capacity, and are generally to be preferred for truly critical applications. The previously mentioned SAFEbus for the Boeing 777, the Shinkansen train control system, and the TTP protocol for automobiles are all time-triggered.

Our goal is a systematic method for transforming round-based protocols from very abstract functional programs, whose properties are comparatively easy to formally and mechanically verify, down to time-triggered implementations with appropriate timing constraints and consideration for realistic fault modes. The transformation is accomplished in two steps: first, from a functional program to an (untimed) synchronous system, then to a time-triggered implementation. The first step is systematic but must be undertaken separately for each algorithm (see Section 4); the other is generic and deals with a large class of algorithms and fault assumptions in a single verification. This generic treatment of the second step is described in the following section.

3 Implementation of Round-Based Algorithms as Time-Triggered Systems

The issues in transforming an untimed round-based algorithm to a time-triggered implementation are basically to ensure that the timing and duration of events in the communication phase are such that messages between nonfaulty processors always arrive in the communication phase of the same round and fault modes are interpreted appropriately. To verify the transformation, we introduce formal models for untimed synchronous systems and for time-triggered systems and, then, establish a simulation relation between them. We verify the simulation by means of a traditional mathematical proof and, then, describe a mechanized verification performed using the PVS verification system [31].

3.1 Synchronous Systems

For the untimed case, we use Lynch's formal model for synchronous systems [29, chapter 2], with some slight adjustments to the notation that make it easier to match up with the mechanically verified treatment.

Definition 1 (Untimed Synchronous Systems). *We assume a set $mess$ of messages that includes a distinguished value $null$ and a set* proc *of processors. Processors are partially connected by directed* channels*; each channel can be thought of a buffer that can hold a single message. Associated with each processor p are the following sets and functions.*

- *A set of processors $out\text{-}nbrs_p$ to which p is connected by outgoing channels.*
- *A set of processors $in\text{-}nbrs_p$ to which p is connected by incoming channels; the function $inputs_p : in\text{-}nbrs_p \rightarrow mess$ gives the message contained in each of those channels.*
- *A set $states_p$ of states with a nonempty subset $init_p$ of initial states. It is convenient to assume that there is a component in the state that counts rounds; this counter is zero in initial states.*
- *A function $msg_p : states_p \times out\text{-}nbrs_p \rightarrow mess$ that determines the message to be placed in each outgoing channel in a way that depends on the current state.*
- *A function $trans_p : states_p \times inputs_p \rightarrow states_p$ that determines the next state in a way that depends on the current state and the messages received in the incoming channels.*

The system starts with each processor in an initial state. All processors p then repeatedly perform the following two actions in lockstep.

Communication Phase: *Apply the message generation function msg_p to the current state to determine the messages to be placed in each outgoing channel. (The message value* null *is used to indicate "no message.")*

Computation Phase: *Apply the state transition function $trans_p$ to the current state and the message held in each incoming channel to yield the next state (with the round counter incremented).*

A particular algorithm is specified by supplying interpretations to the various sets and functions identified above.

3.1.1 Faults

Distributed algorithms are usually required to operate in the presence of faults: The specific kinds and numbers of faults that may arise constitute the *fault hypothesis*. Usually, processor faults are distinguished from communication faults; the former can be modeled by perturbations to the transition functions $trans_p$, and the latter by allowing the messages received along a channel to be changed from those sent. Following [29, p. 20], an *execution* of the system is then an infinite sequence of triples

$$(S_0, M_0, N_0),\ (S_1, M_1, N_1),\ (S_2, M_2, N_2),\ \ldots$$

where S_r is the global state at the start of round r, M_r is the collection of messages placed in the communication channels, and N_r is the (possibly different) collection of messages received.

Because our goal is to show that a time-triggered implementation achieves the same behavior as the untimed synchronous system that serves as its specification, we will need some way to ensure that faults match up across the two systems. For this reason, I prefer to model processor and communication faults by perturbations to the $trans_p$ and msg_p functions, respectively (rather than allowing messages received to differ from those sent); no faulty behaviors are lost by this change. In particular, I assume that the current round number is recorded as part of the state and that if processor p is faulty in round r, with current state s and the values of its input channels represented by the array i, then $trans_p(s, i)$ may yield a value other than that intended; similarly, if the channel from p to q is faulty, then the value $msg_p(s)(q)$ may be different than intended (and may be *null*). Exactly how these values may differ from those intended depends on the fault assumption. For example, a crash fault in round r results in $trans_p(s, i) = s$ and $msg(s)(q) = null$ for all i, q, and states s whose round component is r or greater. Notice that although $trans_p$ and msg_p may no longer be the intended functions, they are still functions; in fact, there is no need to suppose that the $trans_p$ and msg_p were *changed* when the fault arrived in round r: Since the round counter is part of the state, we can just assume these functions behave differently than intended when applied to states having round counters equal or greater than r.

The benefit of this treatment is that, since $trans_p$ and msg_p are uninterpreted, they can represent any algorithm and any fault behavior whatsoever; if we can show that a time-triggered system supplied with arbitrary $trans_p$ and msg_p functions has the same behavior as the untimed synchronous system supplied with the same functions, then this demonstration encompasses behavior in the presence of faults as well as the fault-free case. Furthermore, since we no longer need to hypothesize that faults can cause differences between those messages sent and those received (we instead assume the fault is in msg_p and the "different" messages were actually sent), executions can be simplified from sequences of triples to simple sequences of states

$$S_0,\ S_1,\ S_2,\ \ldots$$

where S_r is the global state at the start of round r. Consequently, to demonstrate that a time-triggered system implements the behavior specified by an untimed synchronous system, we simply need to establish that both systems have the same execution sequences; by mathematical induction, this will reduce to showing that the global states of the two systems are the same at the start of each round r.

3.2 Time-Triggered Systems

For the time-triggered system, we elaborate the model of the previous section as follows.

Each processor is supplied with a clock that provides a reasonably accurate approximation to "real" time. Following [32], we distinguish two notions of time: *clocktime*, denoted $\mathcal{C}$, is the local notion of time supplied by each processor's clock, while *realtime*, denoted $\mathcal{R}$, is an abstract global quantity. We follow the usual convention and denote clocktime quantities by upper case Roman or Greek letters and realtime quantities by lower case letters.

Formally, processor p's clock is a function $C_p : \mathcal{R} \to \mathcal{C}$. The intended interpretation is that $C_p(t)$ is the value of p's clock at realtime t.[2] The clocks of nonfaulty processors are assumed to be well-behaved in the sense that they satisfy the following assumptions.

Assumption 1 (Monotonicity). *Nonfaulty clocks are monotonic increasing functions:*[3]

$$t_1 < t_2 \supset C_p(t_1) < C_p(t_2).$$

Satisfying this assumption requires some care in implementation, because clock synchronization algorithms can make adjustments to clocks that cause them to jump backwards. Lamport and Melliar-Smith describe some solutions [32], and a particularly clever and economical technique for one particular algorithm is introduced by Torres-Pomales [33] and formally verified by Miner and Johnson [34]. Schmuck and Cristian [35] examine the general case and show that monotonicity can be achieved with no loss of precision.

Assumption 2 (Clock Drift Rate). *Nonfaulty clocks drift from realtime at a rate bounded by a small positive quantity ρ:*

$$(1 - \rho)(t_1 - t_2) \leq C_p(t_1) - C_p(t_2) \leq (1 + \rho)(t_1 - t_2).$$

This assumption concerns the hardware clocks employed Inexpensive devices can achieve $\rho < 10^{-6}$.

Assumption 3 (Clock Synchronization). *The clocks of nonfaulty processors are synchronized within some small clocktime bound Σ:*

$$\mid C_p(t) - C_q(t) \mid \leq \Sigma.$$

This assumption can be discharged by a suitable clock synchronization algorithm. There are many such algorithms, several of which have been formally verified [36], [37], [38], [39], [40], [41].

Definition 2 (Time-Triggered Systems). *The feature that characterizes a time-triggered system is that all activity is driven by a global schedule: A processor performs an action when the time on its local clock matches that for which the*

2. In the terminology of [32], these are actually "inverse" clocks.
3. The symbol $\supset$ indicates logical implication.

action is scheduled. In our formal model, the schedule is a function $sched : \mathbb{N} \rightarrow \mathcal{C}$, *where* $sched(r)$ *is the clocktime at which round* r *should begin. The duration of the* r*th round is given by*

$$dur(r) = sched(r+1) - sched(r).$$

In addition, there are fixed global clocktime constants D *and* P *that give the offsets into each round when messages are sent and when the computation phase begins, respectively. Obviously, we need the following constraint.*

$$Constraint\ 1: \qquad 0 < D < P < dur(r)$$

Notice that the duration of the communication phase is fixed (by P*); it is only the duration of the computation phase that can differ from one round to another.*[4]

The states, messages, and channels of a time-triggered system are the same as those for the corresponding untimed synchronous system, as are the transition and message functions. In addition, processors have a one-place buffer for each incoming message channel.

The time-triggered system operates as follows. Initially each processor is in an initial state, with its round counter zero and its clock synchronized with the others and initialized so that $C_p(t_0) \leq sched(0)$, *where* t_0 *is the current realtime. All processors* p *then repeatedly perform the following two actions.*

Communication Phase: *This begins when the local clock reads* $sched(r)$, *where* r *is the current value of the round counter. Apply the message generation function* msg_p *to the current state to determine the messages to be sent on each outgoing channel. The messages are placed in the channels at local clock time* $sched(r) + D$. *Incoming messages that arrive during the communication phase (i.e., no later than* $sched(r) + P$*) are moved to the corresponding input buffer, where they remain stable through the computation phase. These buffers are initialized to null at the beginning of each communication phase and their value is unspecified if more than one message arrives on their associated communications.*

Computation Phase: *This begins at local clock time* $sched(r) + P$. *Apply the state transition function* $trans(p)$ *to the current state and the messages held in the input buffers to yield the next state. The computation will be complete at some local clock time earlier than* $sched(r+1)$. *Increment the round counter, and wait for the start of the next round.*

Message transmission in the communication phase is explained as follows. We use $sent(p, q, m, t)$ to indicate that processor p sent message m to processor q (a member of $out\text{-}nbrs(p)$) at real time t (which must satisfy $C_p(t) = sched(r) + D$ for some round r). We use $recv(q, p, m, t)$ to indicate that processor q received message m from processor p (a member of $in\text{-}nbrs(q)$) at real time t (which must satisfy the constraint $sched(r) \leq C_q(t) < sched(r) + P$ for some round r). These two events are related as follows.

4. There is no difficulty in generalizing the treatment to allow the time at which messages are sent, and the duration of the communication phase, to vary from round to round. That is, the fixed clocktime constants D and P can be systematically replaced by functions $D(r)$ and $P(r)$, respectively. This generalization was developed during the mechanized verification; see Section 3.4.

Assumption 4 (Maximum Delay). *When* p *and* q *are nonfaulty processors,*

$$sent(p, q, m, t) \supset recv(q, p, m, t + d)$$

for some $0 \leq d \leq \delta$.

In addition, we require no spontaneous generation of messages (i.e., $recv(q, p, m, t)$ only if there is a corresponding $sent(p, q, m, t')$ with $t' < t$).

Provided there is exactly one $recv(q, p, m, t)$ event for each p in the communication phase for round r on processor q (as there will be if p is nonfaulty), that unique message m is moved into the input buffer associated with p on processor q before the start of the computation phase for that round and remains there throughout the phase.

Because the clocks are not perfectly synchronized, it is possible for a message sent by a processor with a fast clock to arrive while its recipient is still on the previous round. It is for this reason that we do not send messages until D clocktime units into the start of the round. In general, we need to ensure that a message from a processor in round r cannot arrive at its destination before that processor has started round r nor after it has finished the communication phase for round r. We must establish constraints on parameters to ensure these conditions are satisfied.

Now, processor p sends its message to processor q, say, at realtime t, where $C_p(t) = sched(r) + D$ and, by the maximum delay assumption, the message will arrive at real time $t + d$, where $d \leq \delta$. We need to be sure that

$$sched(r) \leq C_q(t + d) < sched(r) + P. \tag{1}$$

By clock synchronization, we have $| C_q(t) - C_p(t) | \leq \Sigma$; substituting $C_p(t) = sched(r) + D$, we obtain

$$-\Sigma \leq C_q(t) - sched(r) - D \leq \Sigma. \tag{2}$$

By the monotonic clocks assumption, this gives

$$sched(r) + D - \Sigma \leq C_q(t) \leq C_q(t + d)$$

and, so, the first inequality in (1) can be ensured by

$$Constraint\ 2: \qquad D \geq \Sigma.$$

The clock synchronization calculation (2) above also gives

$$C_q(t) \leq sched(r) + D + \Sigma$$

and the clock drift rate assumption gives

$$(1 - \rho)d \leq C_q(t + d) - C_q(t) \leq (1 + \rho)d$$

from which it follows that

$$C_q(t + d) \leq C_q(t) + (1 + \rho)d.$$

Combining these and recalling that $d \leq \delta$, the second inequality in (1) can be ensured by

$$Constraint\ 3: \qquad P > D + \Sigma + (1 + \rho)\delta.$$

3.2.1 Faults

We will prove that a time-triggered system satisfying the various assumptions and constraints identified above

achieves the same behavior as an untimed synchronous system supplied with the same $trans_p$ and msg_p functions. I explained earlier that faults are assumed to be modeled in the $trans_p$ and msg_p functions; by using the same functions in both the untimed and time-triggered systems, we ensure that the latter inherits the same fault behavior and any fault-tolerance properties of the former. Thus, if we have an algorithm that has been shown, in its untimed formulation, to achieve some fault-tolerance properties (e.g., "this algorithm resists a single Byzantine fault or two crash faults"), then we may conclude that the implementation has the same properties.

This simple view is somewhat compromised, however, because the time-triggered system contains a mechanism—time triggering—that is not present in the untimed system. This mechanism admits faults (notably, loss of clock synchronization) that do not arise in the untimed system.

The implementation of a time-triggered system is required to satisfy the synchrony hypothesis and the four assumptions about nonfaulty clocks listed previously. These can be achieved using a suitable fault-tolerant clock synchronization algorithm. The algorithm and its various parameters must be chosen to tolerate the number and kinds of faults specified for the system concerned. For example, the clock synchronization algorithm of TTP (which is based on that of [42]) has recently been formally verified using PVS and shown to satisfy the assumptions we require [41]. However, a clock synchronization algorithm only constrains the behavior of nonfaulty clocks: A processor with a faulty clock may behave in a way that violates the fault model of our time-triggered construction. For example, if one processor's clock drifts to such an extent that it is in the wrong round, then it will execute the transition and message functions appropriate to that round and will supply systematically incorrect messages to the other processors. This could appear as Byzantine behavior at the level of the untimed synchronous algorithm. Less drastic synchronization faults may leave a processor in the right round, but sending messages at the wrong time so that they arrive during the computation phases of other (correct) processors, possibly disrupting their activity.

The implementation of the time-triggered system must include mechanisms that transform faults (such as those due to loss of clock synchronization) that are outside the model considered here into those that are adequately modeled as perturbations to the $trans_p$ and msg_p functions. For example, the round number should be included in messages so that those from the wrong round can be rejected at the message communication layer (thereby reducing the manifestation of such a synchronization fault to fail-silence). TTP goes further and includes all critical state information (operating mode, time, and group membership) in its messages as part of the CRC calculation [10]; messages from a processor that is out of step with respect to any of these items will be rejected by the TTP controllers of other processors.

The impact of messages that arrive in the right round but at the wrong time can be partly countered by moving messages from their input channels to an input buffer at the start of the communication phase: This shields the receiving processor from any changes in channel contents during the computation phase. However, the performance of the computation phase may be degraded by the need to handle interrupts from messages arriving unexpectedly, thereby challenging the synchrony hypothesis. Strong elimination of such timing faults is achieved in practice by techniques to control the "babbling idiot" fault mode. This fault mode occurs when a faulty or unsynchronized processor transmits at arbitrarily wrong times. As well as undesirable manifestations at the synchronous system level, this fault is potentially devastating to the underlying implementation if that implementation multiplexes its communication channels onto shared buses—because the faulty processor can then disrupt the communications of nonfaulty processors. Babbling is eliminated by use of a Bus Interface Unit (BIU) that only grants its processor access to the bus at appropriate times. For example, in SAFEbus, processors are paired, with each member of a pair controlling the other's BIU; in TTP, the BIU has an independent clock and independent knowledge of the schedule [43]. In both cases, babbling can occur only if there are undetected double failures. These mechanisms prevent messages being sent at inappropriate times and ensure that the fault modes of the time-triggered implementation correspond those assumed for the untimed synchronous system.

3.3 Verification

We now need to show that a time-triggered system achieves the same behavior as its corresponding untimed synchronous system. We do this in the traditional way by establishing a simulation relationship between the states of an execution of the time-triggered system and those of the corresponding untimed execution. It is usually necessary to invent an "abstraction function" to relate the states of an implementation to those of its specification; here, however, the states of the two systems are the same and the only difficult point is to select the moments in time at which states of the time-triggered system should correspond to those of the untimed system.

The untimed system makes progress in discrete global steps: All component processors perform their communication and computation phases in lockstep, so it is possible to speak of the complete system being in a round r. The processors of the time-triggered system, however, progress separately at a rate governed by their internal clocks, which are imperfectly synchronized, so that one processor may still be on round r while another has moved on to round $r+1$. We need to establish some consistent "cut" through the time-triggered system that provides a global state in which all processors are at the same point in the same round. In some treatments of distributed systems, it is not necessary for the global cut to correspond to a snapshot of the system at a particular realtime instant: The cut may be an abstract construction that has no direct realization. In our case, however, it is natural to assume that the time-triggered system is used in some control application and that outputs of the individual processors (i.e., some functions of their states) are used to provide redundant control signals in real time—for example, a typical application will be one in which the outputs of the processors are subjected to majority voting or separately drive some

actuator in a "force-summing" configuration.[5] Consequently, we do want to identify the cut through the system with its global state at a specific real time instant.

In particular, we need some realtime instant $gs(r)$ that corresponds to the "global start" of the rth round. We want this instant to be one in which all nonfaulty processors have started the rth round, but have not yet started its computation phase (when they will change their states).

We can achieve this by defining the global start time $gs(r)$ for round r to be the realtime when the processor with the slowest clock begins round r. That is, $gs(r)$ satisfies the following conditions:

$$\forall q : C_q(gs(r)) \geq sched(r) \tag{3}$$

and

$$\exists p : C_p(gs(r)) = sched(r) \tag{4}$$

(intuitively, p is the processor with the slowest clock).

Since the processors are not perfectly synchronized, we need to be sure that they cannot drift so far apart that some processor q has already reached its computation phase—or is even on the next round—at $gs(r)$. Thus, we need

$$\forall q : C_q(gs(r)) < sched(r) + P. \tag{5}$$

By (3), we have $C_q(gs(r)) = sched(r) + X$ for some $X \geq 0$, and (4) plus the clock synchronization assumption then gives $X \leq \Sigma$. Now, processor q will still be on round r and in its communication phase provided $X < P$ and this is ensured by the inequality just derived when taken together with Constraint 3.

We now wish to establish that the global state of a time-triggered system at time $gs(r)$ will be the same as that of the corresponding untimed synchronous system at the start of its rth round. We denote the global state of the untimed system at the start of the rth round by $gu(r)$ (for *global untimed*). Global states are simply arrays of the states of the individual processors so that the state of processor p at this point is $gu(r)(p)$. Similarly, the global state of the time-triggered system at time $gs(r)$ is denoted $gt(r)$ (for *global timed*) and the state of its processor p is $gt(r)(p)$. We can now state and prove the desired result.

Theorem 1. *Given the same initial states, the global states of the untimed and time-triggered systems are the same at the beginning of each round:*

$$\forall r : gt(r) = gu(r).$$

Proof. The proof is by induction.

Base case. This is the case $r = 0$. Both systems are then in their initial states which, by hypothesis, are the same.

Inductive step. We assume the result for r and prove it for $r + 1$. For the untimed case, the message $inputs_q(p)$ from processor p received by q in the rth round is $msg_p(gu(r)(p))(q)$.[6]

By the inductive hypothesis, the global state of processor p in the time-triggered system at time $gs(r)$ is $gu(r)(p)$ also. Furthermore, processor p is in its communication phase (ensured by (5)) and has not changed its state since starting the round. Thus, at local clocktime $sched(r) + D$, it sends $msg_p(gu(r)(p))(q)$ to q. By (1), this is received by q while in the communication phase of round r and transferred to its input buffer $inputs_q(p)$. Thus, the corresponding processors of the untimed and time-triggered systems have the same state and input components when they begin the computation phase of round r. The same state transition functions $trans_p$ are then applied by the corresponding processors of the two systems to yield the same values for the corresponding elements of $gu(r + 1)$ and $gt(r + 1)$, thereby completing the inductive proof. □

3.4 Mechanized Verification

The treatment of synchronous and time-triggered systems in Sections 3.1 and 3.2 has been formally specified in the language of the PVS verification system [31] and the verification of Section 3.3 has been mechanically checked using PVS's theorem prover. The PVS language is a higher-order logic with subtyping and formalization of the semiformal treatment in Sections 3.1 and 3.2 was quite straightforward. The PVS theorem prover includes decision procedures for integer and real linear arithmetic and mechanized checking of the calculations in Section 3.3, and the proof of the theorem, were also quite straightforward. The complete formalization and mechanical verification took less than a day and no errors were discovered. A description, and the formal specifications and proofs themselves, are available at URL http://www.csl.sri.com/dcca97.html.

While it is reassuring to know that the semiformal development of the previous sections withstands mechanical scrutiny, we have argued before (for example, [31], [39]) that mechanized formal verification provides several benefits in addition to the "certification" of proofs. In particular, mechanization supports reliable and inexpensive exploration of alternative designs, assumptions, and constraints. After completing the first version of the work reported here, I wondered whether the requirement that messages be sent at the fixed offset D clocktime units into each round, and that the computation phase begin at the fixed offset P, might not be unduly restrictive. It was the work of a few minutes to generalize the formal specification to allow these offsets to become functions of the round, and to adjust the mechanized proofs. I contend that corresponding revisions to the semiformal development in Sections 3.2 and 3.3 would take longer than this and that it would be difficult to summon the fortitude to scrutinize the revised proofs with the same care as the originals.

4 Round-Based Algorithms as Functional Programs

The theorem of Section 3.3 ensures that synchronous algorithms are correctly implemented by time-triggered implementations that satisfy the various assumptions, constraints, and constructions introduced in the previous

5. For example, the outputs of different processors may energize separate coils of a single solenoid, or multiple hydraulic pistons may be linked to a single shaft (see, e.g., [44, Fig. 3.2–2]).

6. For the benefit of those not used to reading Curried higher-order function applications, this is decoded as follows: $gu(r)(p)$ is p's state in round r; p's message function msg_p applied to that state gives $msg_p(gu(r)(p))$, which is an array of the messages sent to its outgoing channels; q's component of that array is $msg_p(gu(r)(p))(q)$.

section. The next (though logically preceding) step is to ask how one might verify properties of a particular algorithm expressed as an untimed synchronous system.

Although simpler than its time-triggered implementation, the specification of an algorithm as a synchronous system is not especially convenient for formal (and, particularly, mechanized) verification because it requires reasoning about attributes of imperative programs: explicit state and control. It is generally easier to verify functional, rather than imperative, programs because these represent state and control in an applicative manner that can be expressed directly in conventional logic.

There is a fairly systematic transformation between synchronous systems and functional programs that can ease the verification task by allowing it to be performed on a functional program. I illustrate the idea (which comes from Bevier and Young [23]) using the OM(1) algorithm from Section 2. Because that algorithm has already been introduced as a synchronous system, I will illustrate its transformation to a functional program; once the technique becomes familiar, it is easy to perform the transformation in the other direction.

We begin by introducing a function $send(r, v, p, q)$ to represent the sending of a message with value v from processor p to processor q in round r. The value of the function is the message received by q. If p and q are nonfaulty, then this value is v:

$$nonfaulty(p) \wedge nonfaulty(q) \supset send(r, v, p, q) = v,$$

otherwise, it depends on the fault modes considered (in the Byzantine case, it is left entirely unconstrained, as here).

If T represents the transmitter, v its value, and q an arbitrary receiver, then the communication phase of the first round of OM(1) is represented by

$$send(0, v, T, q).$$

The computation phase of this round simply moves the messages received into the states of the processors concerned and can be ignored in the functional treatment (though see footnote 7).

In the communication phase of the second round, each processor q sends the value received in the first round (i.e., $send(0, v, T, q)$) on to the other receivers. If p is one such receiver, then this is described by the functional composition

$$send(1, send(0, v, T, q), q, p). \tag{6}$$

In the computation phase for the second round, processor p gathers all the messages received in the communication phase and subjects them to majority voting.[7] Now, (6) represents the value p receives from q so we need to gather together in some way the values in the messages p receives from all the other receivers q, and use that combination as an argument to the majority vote function. How this "gathering together" is represented will depend on the resources of the specification language and logic concerned: In the treatment using the Boyer-Moore logic, for example, it is represented by a `list` of values [23]. In a higher-order logic, such as PVS [31], however, it can be represented by a function, specified as a λ-abstraction:

$$\lambda q : send(1, send(0, v, T, q), q, p)$$

(i.e., a function that, when applied to q, returns the value that p received from q).

Majority voting is represented by a function *maj* that takes two arguments: the "participants" in the vote, and a function over those participants that returns the value associated with each of them. The function *maj* returns the majority value if one exists; otherwise some functionally determined value. (This behavior can either be specified axiomatically, or defined constructively using an algorithm such as Boyer and Moore's linear time `MJRTY` [45].) Thus, p's decision in the computation phase of the second round is represented by

$$maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p))$$

where *rcvrs* is the set of all receiver processors. We can use this formula as the definition for a higher-order function $OM1(T, v)$ whose value is a function that gives the decision reached by each receiver p when the (possibly faulty) transmitter T sends the value v:

$$\begin{aligned} &OM1(T, v)(p) \\ &\quad = maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p)). \end{aligned} \tag{7}$$

The properties required of this algorithm are the following, whenever the number of receivers is three or more, and at most one processor is faulty.

Requirement 1. Agreement

$$\begin{aligned} &nonfaulty(p) \wedge nonfaulty(q) \\ &\quad \supset \quad OM1(T, v)(p) = OM1(T, v)(q), \end{aligned}$$

Requirement 2. Validity

$$nonfaulty(T) \wedge nonfaulty(p) \supset OM1(T, v)(p) = v.$$

Definition 7 and the requirements for Agreement and Validity stated above are acceptable as specifications to PVS almost as given (PVS requires us to be a little more explicit about the types and quantification involved). Using a constructive definition for *maj*, PVS can prove Agreement and Validity for a specific number of processors (e.g., 4) completely automatically. For the general case of $n \geq 4$ processors, PVS is able to prove Agreement with only a single user-supplied proof directive, while Validity requires half a dozen (the only one requiring "insight" is a case-split on whether the transmitter is faulty).

Not all synchronous systems can be so easily transformed into a recursive function nor can their properties always be formally verified so easily. Nonetheless, I believe the approach has promise for many algorithms of practical interest.

7. In the formulation of the algorithm as a synchronous system, p votes on the messages from the *other* receivers, and the message that it received directly from the transmitter, which it has saved in its state. In the functional treatment, q includes itself among the recipients of the message that it sends in the communication phase of the second round and, so, the vote is simply over messages received in that round.

5 Conclusion

Many round-based fault-tolerant algorithms can be formulated as synchronous systems. I have shown that synchronous systems can be implemented as time-triggered systems and have proven that, provided care is taken with fault modes, the correctness and fault tolerance properties of an algorithm expressed as a synchronous system are inherited by its time-triggered implementation. The proof identifies necessary timing constraints and is independent of the particular algorithm concerned; it can be considered a more general and abstract treatment of the analysis performed for a particular system by Butler and Di Vito [46]. The relative simplicity of the proof supports the argument that time-triggered systems allow for straightforward analysis and should be preferred in critical applications for that reason [30].

In recent work, Pfeifer et al. have formally verified the clock synchronization algorithm used in TTP [41]. Their verification was conducted in PVS and explicitly incorporates the PVS specification, described in Section 3.4, that establishes conditions under which synchronous systems can be implemented as time-triggered systems. Thus, in particular, their work provides a mechanically checked formal verification that the TTP clock synchronization algorithm satisfies the four assumptions of Section 3.2.

I also showed, by example in Section 4, how a round-based algorithm formulated as a synchronous system can be transformed into a functional "program" in a specification logic, where its properties can be verified more easily and more mechanically. I have used the same technique to mechanically verify the three-phase commit algorithm (with its termination protocol) [29, Section 7.3.3]. This is a more difficult algorithm than OM(1) and its verification requires proof by induction (in this respect, it is comparable to the r-round algorithm OM(r)), but its representation as a functional program made the mechanized verification quite straightforward and allowed it to be accomplished in a couple of days. Recently, I have verified a group membership algorithm based on [47] (which is related to the group membership algorithm of TTP) using a similar representation. This is a much more challenging exercise and required further methodological development to make it tractable.

I hope this paper has demonstrated that systematic transformations of fault-tolerant algorithms from functional programs to synchronous systems to time-triggered implementations provides a methodology that can significantly ease the specification and assurance of critical fault-tolerant systems. In collaboration with colleagues from Ülm, I am currently applying the methodology to some of the algorithms of TTP [10].

Acknowledgments

Discussions with N. Shankar and advice from Joseph Sifakis were instrumental in the development of this work. Comments by the anonymous referees of both DCCA and TSE improved the presentation considerably. This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, under Contract F49620-95-C0044; by the National Science Foundation under Contract CCR-9509931; and by NASA Langley Research Center under Contract NAS1-20334.

References

[1] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," *Proc. 15th ACM Symp. Principles of Distributed Computing,* pp. 322–330, Philadelphia, May 1996.

[2] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374–382, Apr. 1985.

[3] R. Guerraoui and A. Schiper, "Consensus: The Big Misunderstanding," *Proc. Sixth IEEE Workshop Future Trends in Distributed Computing,* pp. 183–188, Tunis, Tunisia, Oct. 1997.

[4] H, Kopetz, *Real-Time Systems: Design Princples for Distributed Embedded Applications.* Dordrecht, The Netherlands: Kluwer, 1997.

[5] K. Hoyme and K. Driscoll, "SAFEbus™," *IEEE Aerospace and Electronic Systems Magazine,* vol. 8, no. 3, pp. 34–39, Mar. 1993.

[6] Aeronautical Radio, Inc, Annapolis, Md. *ARINC Specification 659: Backplane Data Bus,* Dec. 1993. Prepared by the Airlines Electronic Engineering Committee.

[7] W. Sweet and D. Dooling, "Boeing's Seventh Wonder," *IEEE Spectrum,* vol. 32, no. 10, pp. 20–23, Oct. 1995.

[8] M.J. Morgan, "Integrated Modular Avionics for Next-Generation Commercial Airplanes," *IEEE Aerospace and Electronic Systems Magazine,* vol. 6, no. 8, pp. 9–12, Aug. 1991.

[9] A. Hachiga, "The Concepts and Technologies of Dependable and Real-Time Computer Systems for Shinkansen Train Control," *Responsive Computer Systems,* H. Kopetz and Y. Kakuda, eds., *Dependable Computing and Fault-Tolerant Systems,* vol. 7, pp. 225–252, Vienna, Austria: Springer-Verlag, 1993.

[10] H. Kopetz and G. Grünsteidl, "TTP—A Protocol for Fault-Tolerant Real-Time Systems," *Computer,* vol. 27, no. 1, pp. 14–23, Jan. 1994.

[11] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Comm. ACM,* vol. 34, no. 2, pp. 56–78, Feb. 1991.

[12] F. Cristian, B. Dancey, and J. Dehn, "Fault-Tolerance in Air Traffic Control Systems," *ACM Trans. Computer Systems,* vol. 14, no. 3, pp. 265–286, Aug. 1996.

[13] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM,* vol. 27, no. 2, pp. 228–234, Apr. 1980.

[14] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Proc. Fault Tolerant Computing Symp. 15,* pp. 200–206, Ann Arbor, Mich., June 1985.

[15] F. Cristian, "Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems," *Distributed Systems,* vol. 4, pp. 175–187, 1991.

[16] P. Zhou and J. Hooman, "Formal Specification and Compositional Verification of an Atomic Broadcast Protocol," *Real-Time Systems,* vol. 9, no. 2, pp. 119–145, 1995.

[17] Y. Gurevich and R. Mani, "Group Membership Protocol: Specification and Verification," *Specification and Validation Methods,* E. Börger, ed., pp. 295–328, Oxford, U.K.: Oxford Univ. Press, 1995.

[18] L. Lamport and S. Merz, "Specifying and Verifying Fault-Tolerant Systems," *Formal Techniques in Real-Time and Fault-Tolerant Systems.* H. Langmaack, W.-P. de Roever, and J. Vytopil, eds., Lübeck, Germany, Lecture Notes in Computer Science 863, pp. 41–76, Springer-Verlag, Sept. 1994.

[19] J. Rushby, "Formal Verification of an Oral Messages Algorithm for Interactive Consistency," Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI Int'l, Menlo Park, Calif., July 1992. also available as NASA Contractor Report 189704, Oct. 1992.

[20] P. Lincoln and J. Rushby, "Formal Verification of an Agorithm for Interactive Consistency Under a Hybrid Fault Model," *Proc. Computer-Aided Verification, CAV '93,* pp. 292–304, C. Courcoubetis, ed., Elounda, Greece, Lecture Notes in Computer Science 697, Springer-Verlag, June/July 1993.

[21] P. Lincoln and J. Rushby, "A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model," *Proc. Fault Tolerant Computing Symp. 23,* pp. 402–411, Toulouse, France, June 1993.

[22] P. Lincoln and J. Rushby, "Formal Verification of an Interactive Consistency Algorithm for the Draper FTP Architecture Under a Hybrid Fault Model," *Proc. Ninth Ann. Conf. Computer Assurance,* pp. 107–120, Gaithersburg, Md., June 1994.

[23] W.R. Bevier and W.D. Young, "The Design and Proof of Correctness of a Fault-Tolerant Circuit," *Dependable Computing for Critical Applications—2,* J.F. Meyer and R.D. Schlichting, eds., *Dependable Computing and Fault-Tolerant Systems,* vol. 6, pp. 243–260, Vienna, Austria: Springer-Verlag, Feb. 1991.

[24] C.J. Walter, P. Lincoln , and N. Suri, "Formally Verified On-Line Diagnosis," *IEEE Trans. Software Eng.,* vol. 23, no. 11, pp. 684–721, Nov. 1997.

[25] J. Rushby, "A Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems," *Formal Techniques in Real-Time and Fault-Tolerant Systems,* J. Vytopil, ed., ch. 5, pp. 109–136, Boston, Dordecht, London: Kluwer, 1993.

[26] B.L. Di Vito and R.W. Butler, "Formal Techniques for Synchronized Fault-Tolerant Systems," *Dependable Computing for Critical Applications—3,* C.E. Landwehr, B. Randell, and L. Simoncini, eds., *Dependable Computing and Fault-Tolerant Systems,* vol. 8, pp. 163–188, Vienna, Austria: Springer-Verlag, Sept. 1992.

[27] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 382–401, July 1982.

[28] W.D. Young, "Comparing Verification Systems: Interactive Consistency in ACL2," *IEEE Trans. Software Eng.,* vol. 23, no. 4, pp. 214–223 Apr. 1997

[29] N.A. Lynch, *Distributed Algorithms.* San Francisco: Morgan Kaufmann, 1996.

[30] H. Kopetz, "Should Responsive Systems be Event-Triggered or Time-Triggered?," *IEICE Trans. Information and Systems,* vol. D, no. 11, pp. 1,325–1,332, Nov. 1993.

[31] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 107–125, Feb. 1995.

[32] L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. ACM,* vol. 32, no. 1, pp. 52–78, Jan. 1985.

[33] W. Torres-Pomales, "An Optimized Implementation of a Fault-Tolerant Clock Synchronization Circuit," NASA Technical Memo. 109176, NASA Langley Research Center, Hampton, Va., Feb. 1995.

[34] P.S. Miner and S.D. Johnson, "Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit: A Case Study Exploring the Boundary between Formal Reasoning Systems," *Designing Correct Circuits,* M. Sheeran and S. Singh, eds., Bastad, Sweden, Sept. 1996.

[35] F. Schmuck and F. Cristian, "Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronization Algorithm," *Proc. Ninth ACM Symp. Principles of Distributed Computing,* pp. 133–143, Québec City, Québec, Canada, Aug. 1990.

[36] J. Rushby and F. von Henke, "Formal Verification of Algorithms for Critical Systems," *IEEE Trans. Software Eng.,* vol. 19, no. 1, pp. 13–23, Jan. 1993.

[37] N. Shankar, "Mechanical Verification of a Generalized Protocol for Byzantine Fault-Tolerant Clock Synchronization," pp. 217-236, Jan. 1992.

[38] P.S. Miner, "Verification of Fault-Tolerant Clock Synchronization Systems," NASA Technical Paper 3349, NASA Langley Research Center, Hampton, Va., Nov. 1993.

[39] J. Rushby, "A Formally Verified Algorithm for Clock Synchronization Under a Hybrid Fault Model," *Proc. 13th ACM Symp. Principles of Distributed Computing,* pp. 304–313, Los Angeles, Calif., Aug. 1994. Also available as NASA Contractor Report 198289

[40] D. Schwier and F. von Henke,, "Mechanical Verification of Clock Synchronization Algorithms," *Formal Techniques in Real-Time and Fault-Tolerant Systems,* Lyngby, Denmark, Lecture Notes in Computer Science 1486, pp. 262–271, Springer-Verlag, Sept. 1998.

[41] H. Pfeifer, D. Schwier, and F.W. von Henke, "Formal Verification for Time-Triggered Clock Synchronization," *Dependable Computing for Critical Applications—7,* J. Rushby, ed., San Jose, Calif., Jan. 1999. *Dependable Computing and Fault Tolerant Systems,* pp. 207-286, C.B. Weinstock and J. Rushby, eds., IEEE Computer Society.

[42] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. Computers,* vol. 36, no. 8, pp. 933–940, Aug. 1987

[43] C. Temple, "Avoiding the Babbling-Idiot Failure in a Time-Triggered Communication System," *Proc. Fault Tolerant Computing Symp. 28,* pp. 218–227, Munich, Germany, June 1998.

[44] C.S. Droste and J.E. Walker, *The General Dynamics Case Study on the F16 Fly-by-Wire Flight Control System,* AIAA Professional Study Series. American Inst. of Aeronautics and Astronautics, Undated.

[45] R.S. Boyer and J.S. Moore, "MJRTY—A Fast Majority Vote Algorithm," *Automated Reasoning: Essays in Honor of Woody Bledsoe,* R.S. Boyer, ed., vol. 1, pp. 105–117, Dordrecht, The Netherlands: Kluwer Academic, 1991.

[46] R.W. Butler and B.L. Di Vito, "Formal Design and Verification of a Reliable Computing Platform for Real-Time Control: Phase 2 Results," NASA Technical Memo. 104196, NASA Langley Research Center, Hampton, Va., Jan. 1992.

[47] S. Katz, P. Lincoln, and J. Rushby, "Low-Overhead Time-Triggered Group Membership," *Proc. 11th Int'l Workshop Distributed Algorithms (WDAG '97),* pp. 155–169, M. Mavronicolas and P. Tsigas, eds., Saarbrüken, Germany, Lecture Notes in Computer Science 1320, Springer-Verlag, Sept. 1997.

[48] IEEE Computer Society, *Proc. Fault Tolerant Computing Symp. 25: Highlights from 25 Years,* Pasadena, Calif., June 1995.

[49] *Formal Techniques in Real-Time and Fault-Tolerant Systems,* J. Vytopil, ed., Nijmegen, The Netherlands, Lecture Notes in Computer Science 571, Springer-Verlag, Jan. 1992.

John Rushby received the BSc and the PhD degrees, both in computing science, from the University of Newcastle upon Tyne in 1971 and 1977, respectively. He joined the Computer Science Laboratory of SRI International in 1983 and served as its director from 1986 to 1990; he currently manages its research program in formal methods and dependable systems. Prior to joining SRI, he held academic positions at the Universities of Manchester and Newcastle upon Tyne in England. His research interests center on the use of formal methods for problems in the design and assurance of dependable systems. Dr. Rushby is a member of the IEEE, the ACM, the American Institute of Aeronautics and Astronautics, and the American Mathematical Society. He is an associate editor for the *IEEE Transactions on Software Engineering* and a member of the editorial board for the *Formal Aspects of Computing* journal.