Expanded version of a paper from the Sixth Working Conference on Dependable Computing for Critical Applications, Garmisch-Partenkirchen, Germany, March, 1997. IEEE Computer Society Press, pp. 203–222.

Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms*

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
Rushby@csl.sri.com

Abstract

Many critical real-time applications are implemented as time-triggered systems. We present a systematic way to derive a time-triggered implementation from a fault-tolerant algorithm specified as a functional program. It is relatively easy to formally and mechanically verify correctness and fault-tolerance properties of algorithms expressed in this latter form. The functional program is next transformed into an untimed synchronous system, and then to a time-triggered implementation. The second step is independent of the algorithm concerned and we prove its correctness; the proof has also been formalized and mechanically checked with the PVS verification system. This approach provides a methodology that can ease the formal specification and assurance of critical fault-tolerant systems.

*This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044, by the National Science Foundation under contract CCR-9509931, and by NASA Langley Research Center under contract NAS1-20334.

Contents

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 [6, 13].

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. For example, process scheduling algorithms that can miss deadlines, buffer overflows, and contention buses such as Ethernet can all lead to violations of the synchrony assumption, but may be considered "good enough" in less than truly critical applications. Those applications that are truly critical, however, 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 Honeywell SAFEbus™ [1, 17] that provides the safety-critical backplane for the Boeing 777 Airplane Information Management System (AIMS) [31, 39], the control system for the Shinkansen (Japanese Bullet Train) [16], and the Time-Triggered Protocol (TTP) proposed for safety-critical automobile functions [21] all use this latter approach.

A number of basic functions have been identified that provide important building blocks in the construction of fault-tolerant synchronous systems [8, 10]; these include consensus (also known as interactive consistency and Byzantine agreement) [33], reliable and atomic broadcast [9], and group membership [7]. 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 [15, 23, 43] and mechanically checked [2, 26–28, 34] verifications, as have their combination into larger functions such as diagnosis [25], and their synthesis into a fault-tolerant architecture based on active (state-machine) replication [11, 35].

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 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 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 [34] and Bevier and Young [2] describe mechanically checked formal verifications of the same "Oral Messages" algorithm [24] for the consensus problem that were performed using different verification systems. Young [42] argues that differences in the difficulty of these treatments (that of [34] is generally considered simpler and clearer than that of [2]) are due to different choices in the way things are formalized. 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 [28] in a couple of lines and its verification is almost completely automatic. In contrast, the treatment of OM(1) in [23] 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. If things work out well, the refinement argument should be a routine calculation of timeouts and other concrete details. The purpose of this paper is to present a framework for such a hierarchical treatment and to show that, for the important case of time-triggered implementations of round-based algorithms, most of the details of the refinement to a concrete formulation can be worked out once and for all.

2 Round-Based Algorithms

In her textbook [29], Nancy 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 will 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

[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.

SAFEbus) and all run a common, deterministic 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 the 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).

Some algorithms were first introduced using an event-triggered formulation (for example, Cristian's atomic broadcast and group membership algorithms [7, 9]), but it is possible to reconstruct explicitly round-based equivalents for them, and then transform them to time-triggered implementations (Kopetz' time-triggered algorithms [19] for the same problems do this to some extent). 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 [20,21] argues persuasively that time-triggered systems are more predictable (and hence easier to verify), easier to test, make better use of broadcast communications bandwidth (since no addresses need be communicated—these are implicit in the time at which a message is sent), 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 Round-Based Algorithms Implemented 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. This treatment has been formalized and mechanically checked using the PVS verification system—see Section 3.4.

3.1 Synchronous Systems

For the untimed case, we use Nancy 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 $\mathit{out\text{-}nbrs}_p$ to which p is connected by outgoing channels.
- A set of processors $\mathit{in\text{-}nbrs}_p$ to which p is connected by incoming channels; the function $\mathit{inputs}_p : \mathit{in\text{-}nbrs}_p \rightarrow \mathit{mess}$ 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 $\mathit{msg}_p : \mathit{states}_p \times \mathit{out\text{-}nbrs}_p \rightarrow \mathit{mess}$ that determines the message to be placed in each outgoing channel in a way that depends on the current state.
- A function $\mathit{trans}_p : \mathit{states}_p \times \mathit{inputs}_p \rightarrow \mathit{states}_p$ 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.

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, page 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 communications faults by perturbations to the $trans_p$ and msg_p functions, respectively (rather than allowing messages received to differ from those sent). 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; 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. When speaking of clocks, it is usual to 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. It is also usual to 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 $C_p : \mathcal{R} \rightarrow \mathcal{C}$. 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 of satisfying the following assumptions.

Assumption 1 Monotonicity. *Nonfaulty clocks are monotonic increasing functions:*

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

Satisfying this assumption requires some care in implementation, because clock synchronization algorithms can make adjustments to clocks that cause them to

[2] In the terminology of [22], these are actually "inverse" clocks.

jump backwards. Lamport and Melliar-Smith describe some solutions [22], and a particularly clever and economical technique for one particular algorithm is introduced by Torres-Pomales [40] and formally verified by Miner and Johnson [30]. Schmuck and Cristian [38] 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 (typically $\rho < 10^{-6}$):

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

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

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

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 action is scheduled. In our formal model, the schedule is a function $sched : \mathbb{N} \to \mathcal{C}$, 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 $\quad 0 < D < P < dur(r).$

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.[3]

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.

[3] In fact, 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.

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 channel in a given communication phase.

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-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-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.

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 $sent(p, q, m, t'))$.

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 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 realtime $t + d$ where $d \le \delta$. We need to be sure that

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

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

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

By the monotonic clocks assumption, this gives

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

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

Constraint 2 $\qquad D \ge \Sigma.$

The clock synchronization calculation (2) above also gives

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

and the clock drift rate assumption gives

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

from which it follows that

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

Thus, the second inequality in (1) can be ensured by

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

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. An implementation must ensure that such faults are either masked, or are transformed in such a way that their manifestations are accurately modeled by perturbations in the $trans_p$ and msg_p functions.

In general, it is desirable to transform low-level faults (i.e., those outside the model considered here) into the *simplest* (most easily tolerated) fault class for the algorithm concerned. If no low-level mechanism for dealing with loss of clock synchronization is present, then synchronization faults may manifest themselves as arbitrary, Byzantine faults to the abstract algorithm. 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 easily appear as Byzantine behavior at the level of the untimed synchronous algorithm. For this reason, it is desirable to include the round number in messages, so that those from the wrong round can be rejected (thereby reducing the fault manifestation 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 [21].

Less drastic clock skews may leave a processor in the right round, but sending messages at the wrong time, so that they arrive during the computation phases of the other (correct) processors. It is partly to counter this fault mode that the time-triggered model used here explicitly moves messages from their input channels to an input buffer during the communication phase: this shields the receiving processor from any changes in channel contents during the computation phase.

If the physical implementation of the time triggered system multiplexes its communications channels onto shared buses, then it is necessary to control the "babbling idiot" fault mode where a faulty processor disrupts the communications of other processors by speaking out of turn. In practice, this is controlled by a Bus Interface Unit (BIU) that only grants 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 independent knowledge of the schedule. In both cases, babbling can occur only if there are undetected double failures.

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.[4] 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 r'th round. We want this instant to be one in which all nonfaulty processors have started the r'th 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 constraints:

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

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

[4] 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., [12, Figure 3.2–2]).

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 r'th round. We denote the global state of the untimed system at the start of the r'th 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 r'th round is $msg_p(gu(r)(p))(q)$.[5]

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.

□

[5] 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 $msg_p(gu(r)(p))(q)$.

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 [32], 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. The formal specification and verification are described in the Appendix; the specification files themselves are available at URL `http://www.csl.sri.com/dcca97.html`.

While it is reassuring to know that the semiformal development withstands mechanical scrutiny, we have argued previously (for example, [32, 36]) 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. In this case, 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 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 [2]) 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 6).

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.[6] 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 [2]. In a higher-order logic such as PVS [32], 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

[6] 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.

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` [4].) 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:

$$OM1(T, v)(p) = maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p)). \qquad (7)$$

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

Agreement: $nonfaulty(p) \wedge nonfaulty(q) \supset OM1(T, v)(p) = OM1(T, v)(q)$,

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 we 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. A similar method has been advocated by Florin, Gómez, and Lavallée [14].

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 proved 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 provides a more general and abstract treatment of the analysis performed for a particular system by Di Vito and Butler [5]. 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 [20].

I have also shown, by example, 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.

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 current work, we are applying the methodology to some of the algorithms of TTP [21].

Acknowledgments

Discussions with N. Shankar and advice from Joseph Sifakis were instrumental in the development of this work. Comments by the anonymous referees improved the presentation.

References

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

[2] W. R. Bevier and W. D. Young. The design and proof of correctness of a fault-tolerant circuit. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications—2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*, pages 243–260. Springer-Verlag, Vienna, Austria, February 1991.

[3] Robert S. Boyer and J Strother Moore. MJRTY—a fast majority vote algorithm. Technical Report 32, Institute for Computing Science, University of Texas, Austin TX, February 1981.

[4] Robert S. Boyer and J Strother Moore. MJRTY—a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning Series*, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[5] Ricky W. Butler and Ben L. Di Vito. Formal design and verification of a reliable computing platform for real-time control: Phase 2 results. NASA Technical Memorandum 104196, NASA Langley Research Center, Hampton, VA, January 1992.

[6] Tushar D. Chandra, Vassos Hadzilicos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, PA, May 1996. Association for Computing Machinery.

[7] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Systems*, 4:175–187, 1991.

[8] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[9] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Fault Tolerant Computing Symposium 15*, pages 200–206, Ann Arbor, MI, June 1985. IEEE Computer Society. Reprinted in [18, pp. 431–437].

[10] Flaviu Cristian, Bob Dancey, and Jon Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computer Systems*, 14(3):265–286, August 1996.

[11] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 163–188. Springer-Verlag, Vienna, Austria, September 1992.

[12] Carl S. Droste and James E. Walker. *The General Dynamics Case Study on the F16 Fly-by-Wire Flight Control System*. AIAA Professional Study Series. American Institute of Aeronautics and Astronautics. Undated.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[14] Gérard Florin, Roberto Gómez, and Ivan Lavallée. Systematic building of a distributed recursive algorithm. Rapport de recherche 1902, INRIA, Rocquencourt, France, May 1993.

[15] Yuri Gurevich and Raghu Mani. Group membership protocol: Specification and verification. In Egon Börger, editor, *Specification and Validation Methods*, International Schools for Computer Scientists, pages 295–328. Oxford University Press, Oxford, UK, 1995.

[16] Akira Hachiga. The concepts and technologies of dependable and real-time computer systems for Shinkansen train control. In H. Kopetz and Y. Kakuda, editors, *Responsive Computer Systems*, volume 7 of *Dependable Computing and Fault-Tolerant Systems*, pages 225–252. Springer-Verlag, Vienna, Austria, 1993.

[17] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. *IEEE Aerospace and Electronic Systems Magazine*, 8(3):34–39, March 1993.

[18] *Fault Tolerant Computing Symposium 25: Highlights from 25 Years*, Pasadena, CA, June 1995. IEEE Computer Society.

[19] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avižienis and J. C. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable*

Computing and Fault-Tolerant Systems, pages 411–429, Santa Barbara, CA, August 1989. Springer-Verlag, Vienna, Austria.

[20] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered? *IEICE Transactions on Information and Systems*, E76-D(11):1325–1332, November 1993. Institute of Electronics, Information, and Communications Engineers, Japan.

[21] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

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

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

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[25] Patrick Lincoln. Formally verified algorithms for diagnosis of manifest, symmetric, link, and Byzantine faults. Technical Report SRI-CSL-95-14, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1995.

[26] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.

[27] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society. Reprinted in [18, pp. 438–447].

[28] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[29] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1996.

[30] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, September 1996. Springer-Verlag.

[31] Michael J. Morgan. Integrated modular avionics for next-generation commercial airplanes. *IEEE Aerospace and Electronic Systems Magazine*, 6(8):9–12, August 1991.

[32] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[33] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[34] John Rushby. Formal verification of an Oral Messages algorithm for interactive consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992. Also available as NASA Contractor Report 189704, October 1992.

[35] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [41, pp. 237–257].

[36] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[37] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Revised, July 1996. Available, with specification files, at `http://www.csl.sri.com/csl-95-10.html`.

[38] Frank Schmuck and Flaviu Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. In *Ninth ACM Symposium on Principles of Distributed Computing*, pages 133–143, Québec City, Québec, Canada, August 1990.

[39] William Sweet and Dave Dooling. Boeing's seventh wonder. *IEEE Spectrum*, 32(10):20–23, October 1995.

[40] Wilfredo Torres-Pomales. An optimized implementation of a fault-tolerant clock synchronization circuit. NASA Technical Memorandum 109176, NASA Langley Research Center, Hampton, VA, February 1995.

[41] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

[42] William D. Young. Comparing verification systems: Interactive Consistency in ACL2. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 35–45, Gaithersburg, MD, June 1996. IEEE Washington Section.

[43] Ping Zhou and Jozef Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, 9(2):119–145, 1995.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Appendices

In these appendices I present mechanically-checked formal specifications and verifications of the developments in Sections 3 and 4 using the PVS verification system [32]. The corresponding PVS specification and proof files are available from URL `http://www.csl.sri.com/dcca97.html`.

A Untimed and Timed Synchronous Systems in PVS

The PVS treatment is presented in three subsections corresponding to those of Section 3.

A.1 Untimed Synchronous Systems

I begin by specifying the synchronous system model of Section 3.1 in the language of PVS.

The starting point is Definition 1 on Page 7. In PVS, the sets of processors, messages, and states are represented by the nonempty types, `proc`, `mess`, and `state` respectively. For simplicity, we assume that the state type is the same for all processors (in Section 3.1, each processor p had its own individual set of states, $states_p$; it is easy to add this embellishment if desired). The initial states are represented by the nonempty subtype `init_state` of `state`, and the distinguished null message by the uninterpreted constant `null` of type `mess`. Next, some variables are declared and `in_nbrs` and `out_nbrs` are specified as functions from processors to sets of processors, so that, for example, `out_nbrs(p)` represents $out\text{-}nbrs_p$, the set of processors connected to processor p by outgoing channels. The axiom `channels_match` states a necessary constraint, missing from the treatment of Section 3.1, that q is among the *in-nbrs* of p if and only p is among the *out-nbrs* of q.

```
untimed_system: THEORY
BEGIN

  proc, mess, state: TYPE+
  init_state: TYPE+ FROM state
  null: mess

  in_nbrs, out_nbrs: [proc -> setof[proc]]

  p, q: VAR proc
  s, t: VAR state

  channels_match: AXIOM in_nbrs(p)(q) IFF out_nbrs(q)(p)
```

In PVS, a set (or equivalently, a predicate) enclosed in parentheses denotes the corresponding subtype; thus `(out_nbrs(p))` is the subtype of `proc` comprising those processors connected to `p` by outgoing channels. We can therefore specify the message function `msg(p)` for processor `p` to have the following dependently-typed signature.

```
msg(p): [state, (out_nbrs(p)) -> mess]
```

That is, `msg(p)` takes a state and processor from among the *out-nbrs* of `p` and returns the message to be sent to that processor. The messages received by processor `p` form an array indexed by `(in_nbrs(p))`; arrays are equivalent to functions in PVS, so this array has type `[(in_nbrs(p)) -> M]`, which we denote by the PVS *type application* (essentially, a parameterized type) `inputs(p)` as follows.

```
inputs(p): TYPE = [(in_nbrs(p)) -> M]
```

The state transition function for processor `p` can then be given the following signature.

```
trans(p): [state, inputs(p) -> state]
```

The full ("global") synchronous system is composed of the states of each of its constituent processors; thus the global states and initial states have the following types.

```
global_state: TYPE = [proc -> state]
global_init: TYPE = [proc -> init_state]
```

If the system is in global state `s`, then processor `q`'s component of that state is `s(q)`, and so the message that it sends to processor `p` during the communication phase is `msg(q)(s(q), p)`. Thus, the array of inputs received by `p` in that phase is given by abstracting this expression over all `p`'s *in-nbrs* `q`:

```
(LAMBDA (q:(in_nbrs(p))): msg(q)(s(q), p)).
```

Therefore, `p`'s computation phase will result in the state

```
trans(p)(s(p), (LAMBDA (q:(in_nbrs(p))): msg(q)(s(q), p))).
```

This expression provides the heart of the definition for the recursive function `run(ginit)(r)`, which specifies the global state of the system following `r` rounds, starting from the global initial state `ginit`, as follows.

```
round: TYPE = nat

run(ginit: global_init)(r: round):  RECURSIVE global_state =
   IF r = 0 THEN ginit
   ELSE LAMBDA p: trans(p)(s(p),
                        LAMBDA (q:(in_nbrs(p))): msg(q)(s(q), p))
     WHERE s = run(ginit)(r - 1)
   ENDIF
MEASURE r
```

This definition causes PVS to generate three TCCs (proof obligations): one to establish termination, one to ensure that the argument `r-1` in the recursive call satisfies the constraints on `rounds` (i.e., is nonnegative), and one to ensure that the second argument to `msg(q)` (i.e., `p`) is in the *out-nbrs* of `q`. The first two TCCs are discharged by PVS's default strategies, and the third by appeal to the `channels_match` axiom.

A.1.1 LaTeX-printed Specification

untimed_system : THEORY
BEGIN

proc, mess, state : TYPE+
init_state : TYPE+ FROM state
null : mess
in_nbrs, out_nbrs : [proc → setof[proc]]
p, q : VAR proc
s, t : VAR state

channels_match : AXIOM in_nbrs(p)(q) ⇔ out_nbrs(q)(p)

msg(p) : [state, (out_nbrs(p)) → mess]
inputs(p) : TYPE = [(in_nbrs(p)) → mess]
trans(p) : [state, inputs(p) → state]

global_state : TYPE = [proc → state]
global_init : TYPE = [proc → init_state]

round : TYPE = nat

run(ginit : global_init)(r : round) : RECURSIVE global_state =
IF $r = 0$ THEN ginit
ELSE λ p : trans(p)(s(p), (λ (q : (in_nbrs(p))) : msg(q)(s(q), p)))
WHERE s = run(ginit)(r − 1)
ENDIF
MEASURE r

END untimed_system

A.1.2 Testing the Specification. It is generally a good idea to validate top-level specifications by checking that some expected properties do indeed hold. For the `untimed_system` specification, we could do this by checking that it "runs" the algorithm OM(1) correctly. However, this will be a rather complicated demonstration (after all, the whole point of Section 4 was to demonstrate an easier approach using a functional specification), so we will be satisfied by the simpler algorithm OM(0). This one-round algorithm proceeds as follows.

Algorithm OM(0)

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.

In the absence of faults, OM(0) ensures that all receivers store the value sent by the transmitter.

To specify and prove this in PVS, we import the `untimed_system` specification, and introduce the nonempty type `value` of values. We then posit the function `val` that extracts values from a processor's state, and functions `encode`, and `decode` that convert values to and from messages; we require that `decode` is a left inverse of `encode`.

```
OM_zero: THEORY
BEGIN

  IMPORTING untimed_system
  value: TYPE+
  v: VAR value
  s: VAR state

  val: [state -> value]
  encode: [value -> mess]
  decode: [mess -> value]
  code_ax: AXIOM decode(encode(v)) = v
```

We can then identify the `transmitter` as a distinguished processor, the `receivers` are all the other processors, and we assert that the transmitter has outgoing connections to all receivers.

```
transmitter: proc
receivers: setof[proc] = remove(transmitter, fullset[proc])
rcvr: VAR (receivers)
connectivity: AXIOM out_nbrs(transmitter) = receivers
```

In all states, the transmitter takes the value stored in its state, and encodes it in a message that is sent to all receivers.

```
alg_send_ax: AXIOM msg(transmitter)(s, rcvr) = encode(val(s))
```

This axiom generates a TCC requiring us to demonstrate that all receivers are among the *out-nbrs* of the transmitter. This obligation is discharged by the proof command `(STEW :LEMMAS ("connectivity"))`.

In their computation phase, all receivers take the message received from the transmitter and store it in their state in such a way that the value represented equals the decoded message.

```
alg_trans_ax: AXIOM FORALL rcvr, s, (i: inputs(rcvr)):
   val(trans(rcvr)(s, i)) = decode(i(transmitter))
```

This axiom generates a TCC requiring us to demonstrate that all receivers have the transmitter among their *in-nbrs*. This obligation is discharged by the proof command `(STEW :LEMMAS ("connectivity" "channels_match"))`.

The property required of this algorithm, after one round, is that all processors have the same value in their state as the transmitter had initially.

```
works: THEOREM FORALL (ginit:global_init):
   val(run(ginit)(1)(rcvr)) = val(ginit(transmitter))
```

This challenge theorem is proved automatically in PVS using the single command `(grind :theories "OM_zero")`.

A.1.3 LaTeX-printed Specification

OM_zero : THEORY
BEGIN

IMPORTING untimed_system

value : TYPE+
v : VAR value
s : VAR state

val : [state $\rightarrow$ value]

encode : [value $\rightarrow$ mess]
decode : [mess $\rightarrow$ value]

code_ax : AXIOM decode(encode(v)) = v

transmitter : proc
receivers : setof[proc] = (fullset[proc] \ {transmitter})
rcvr : VAR (receivers)

connectivity : AXIOM out_nbrs(transmitter) = receivers

alg_send_ax : AXIOM msg(transmitter)(s, rcvr) = encode(val(s))

alg_trans_ax : AXIOM
$\forall$ rcvr, s, (i : inputs(rcvr)) : val(trans(rcvr)(s, i)) = decode(i(transmitter))

works : THEOREM
$\forall$ (ginit : global_init) : val(run(ginit)(1)(rcvr)) = val(ginit(transmitter))

END OM_zero

Encouraged that our specification of untimed systems meets this simple challenge, we proceed to the specification of time-triggered systems.

A.2 Time-Triggered Synchronous Systems

Development of the time-triggered system model in Section 3.2 builds on that of the untimed system model. The formal specification of time-triggered systems similarly begins by importing the theory `untimed_systems`, then introduces `clocktime` and `realtime` (modeled by the natural and real numbers, respectively), declares some variables, and then introduces the clock function `C(p, t)`, which corresponds to $C_p(t)$ of Section 3.2 (Page 9) and represents the value of `p`'s clock at realtime `t`.

```
time_triggered_system: THEORY
BEGIN
  IMPORTING untimed_system

  p, q: VAR proc
  s: VAR state

  clocktime: TYPE = nat
  realtime: TYPE = real

  T: VAR clocktime
  t, t1, t2: VAR realtime

  C(p, t): clocktime
```

Next, we introduce uninterpreted constants `rho` and `Sigma` corresponding to ρ and Σ of Section 3.2, and three axioms corresponding to the Assumptions 1 to 3 that were presented on Page 9.

```
  monotone_clock: AXIOM t1 < t2 => C(p, t1) < C(p, t2)

  rho: {x: real | 0 < x AND x < 1}

  drift_rate: AXIOM (1 - rho) * (t1 - t2) <= C(p, t1) - C(p, t2)
              AND C(p, t1) - C(p, t2) <= (1 + rho) * (t1 - t2)

  Sigma: clocktime

  clock_sync: AXIOM abs(C(p, t) - C(q, t)) < Sigma
```

The declaration of `rho` generates a TCC to ensure that its type is not empty. This is discharged by exhibiting the value `1/2`.

The uninterpreted function `sched(r)` gives the clocktime at which execution of round `r` should begin. The axiom `monotone_sched` ensures that this function is monotone increasing. The function `dur(r)` is defined to be the duration of the `r`'th round, while the uninterpreted functions `D(r)` and `P(r)` respectively give the clocktime offsets into round `r` at which each processor sends its messages and starts its computation phase. Notice that these offsets are functions of the round, thereby generalizing the treatment in Section 3.2 (Page 10), where they were fixed constants. The axioms `constraint1` and `constraint2` specify the corresponding constraints from Section 3.2 (Pages 10 and 12, respectively).

```
i, j, r: VAR round
sched(r): clocktime

monotone_sched: AXIOM i < j IMPLIES sched(i) < sched(j)

dur(r): clocktime = sched(r + 1) - sched(r)

D(r): clocktime
P(r): clocktime

constraint1: AXIOM 0 < D(r) AND D(r) < P(r) AND P(r) < dur(r)
constraint2: AXIOM D(r) >= Sigma
```

The definition of `dur(r)` generates a TCC to ensure that its value is nonnegative; this obligation is discharged by appeal to `monotone_sched`.

From `constraint1` and `monotone_sched`, the following lemma can be established by induction on the difference between `j` and `i`.

```
comp_phase: LEMMA i < j => sched(i) + P(i) < sched(j)
```

This result states that the computation phase for round `i` starts strictly earlier than the start of any round greater than `i`.

The processors of the time-triggered system schedule their actions according to the passage of clocktime. Thus, given a global initial state `ginit` and clocktime `T`, we need to model the state of processor `p` at that time: the uninterpreted function `ttss(ginit)(p)(T)` is used for this purpose (`ttss` stands for "time-triggered system state"). The uninterpreted predicates `sent` and `recv` are used to model the sending and receipt of messages just as in Section 3.2 (Page 11).

```
ginit: VAR global_init
ttss(ginit)(p)(T): state

sent(p, (q:(out_nbrs(p))), m, t): bool
recv(q, (p:(in_nbrs(q))), m, t): bool
```

The properties of these three uninterpreted functions are specified axiomatically, as follows. The state of processor `p` at the start of round `r` is `ttss(ginit)(p)(sched(r))`, so the message `m` that it should send to processor `q` in that round will be `msg(p)(ttss(ginit)(p)(sched(r)), q)`. This message should be sent at clocktime `sched(r)+D(r)`; we denote the corresponding realtime by `sendtime(p, r)`.

```
sendtime(p, r): realtime

sendtime_ax: AXIOM C(p, sendtime(p, r)) = sched(r) + D(r)
```

The following axiom then specifies that the predicate `sent(p, q, m, t)` should be true when `m` and `t` have the values just identified.

```
comm_phase1: AXIOM
     FORALL (q: (out_nbrs(p))): sent(p, q, m, sendtime(p, r))
         WHERE m = msg(p)(ttss(ginit)(p)(sched(r)), q)
```

It is also necessary to specify that the predicate is true *only* in this circumstance—that is, only correct messages are sent, and only at the correct time.

```
comm_phase2: AXIOM
    FORALL (q: (out_nbrs(p))): sent(p, q, m, t)
      => EXISTS r: t = sendtime(p, r)
                       AND m = msg(p)(ttss(ginit)(p)(sched(r)), q)
```

Sending and reception of messages are related by the `max_delay` axiom, which corresponds to Assumption 4 on Page 11 in Section 3.2. In the following specification of this axiom, `delta`, the maximum delay, is some nonnegative realtime constant and `delay` is the subtype of realtime comprising those times between `0` and `delta`, inclusive. The declaration of `delta` generates an existence TCC to ensure that its type is not empty; this is discharged by exhibiting the value `0`.

```
delta: {x: realtime | 0 <= x}
delay: TYPE = {x: realtime | 0 <= x AND x <= delta}
d: VAR delay

max_delay: AXIOM
  sent(p, q, m, t) IFF EXISTS (d: delay): recv(q, p, m, t + d)
```

It is convenient to separate the equivalence in this axiom into two implications, specified by the following lemmas, whose proofs are trivial.

```
max_delay1: LEMMA
   sent(p, q, m, t) => EXISTS (d: delay): recv(q, p, m, t + d)

max_delay2: LEMMA
   recv(q, p, m, t) => EXISTS (d: delay): sent(p, q, m, t - d)
```

The `max_delay` axiom and its lemmas generate TCCs to ensure that `p` is among the *in-nbrs* of `q` or, dually, that `q` is among the *out-nbrs* of `p`; these are discharged by appeal to the `channels_match` axiom.

Next, we specify the remaining constraint from Section 3.2 (Page 12).

```
constraint3: AXIOM P(i) > D(i) + Sigma + (1 + rho) * delta
```

The nonlinear product in `constraint3` complicates theorem proving (PVS has decision procedures for *linear* arithmetic only), so it is convenient to establish the following lemmas.

```
rho_prop1: LEMMA (1 - rho) * d >= 0
rho_prop2: LEMMA (1 + rho) * d >= 0

constraint3_lemma: LEMMA P(i) > D(i) + Sigma + (1 + rho) * d
```

The first two follow from the type constraints on `rho` by the PVS prelude formula `pos_times_ge`; the third follows from `constraint3` using the prelude formula `both_sides_times_pos_le2`.

Next, we introduce a predicate `in_comm_phase(t, p, r)` that is true when the realtime `t` is in the communication phase for round `r` on processor `p`.

```
in_comm_phase(t, p, r): bool =
   sched(r) <= C(p, t) AND C(p, t) < sched(r) + P(r)
```

A straightforward proof appealing to `drift_rate`, `clock_sync`, `constraints 1` and `2`, `constraint3_lemma`, and `rho_props` `1` and `2` then establishes the following lemma, which ensures that a message sent by processor `p` in round `r` and subject to a delay `d` arrives at processor `p` while that processor is in its communication phase for the same round.

```
arrival_prop: LEMMA in_comm_phase(sendtime(q, r)+d, p, r)
```

(This is a formal restatement of the formula (1) on Page 12 of Section 3.2.)

Using `arrival_prop`, `comm_phase1`, and `max_delay1`, we can then prove the following lemma, which establishes that processor `p` will receive some message `m` from processor `q` during the communication phase of the `r`'th round.

```
recv_prop: LEMMA FORALL r, p, (q: (in_nbrs(p))):
   EXISTS (m: mess), (t: realtime):
            in_comm_phase(t, p, r) AND recv(p, q, m, t)
```

The function `ttin` specifies the message available to processor `p` at clocktime `T` from processor `q`. The signature of this function is specified as follows

```
ttin(p)(T)((q: (in_nbrs(p)))): M
```

and its interpretation is supplied by the following axiom.

```
ttin_ax: AXIOM sched(r) + P(r) <= T AND T < sched(r + 1)
   => FORALL (q: (in_nbrs(p))):
          ttin(p)(T)(q) = epsilon! (m):
               EXISTS t: in_comm_phase(t, p, r) AND recv(p, q, m, t)
```

This axiom uses Hilbert's ε operator (represented in PVS by `epsilon`) to say that if `T` is in the computation phase for round `r`, then the message `ttin(p)(T)(q)` is the one received from `q` during the communication phase for that round—or one of them if more than one was received, or an arbitrary message if none was received. To rule out these latter possibilities, we will need to demonstrate that the time-triggered system is sufficiently well-behaved that exactly one message is received by `p` from `q`.

We can predicate this demonstration on the assumption that the state of the time-triggered system at the start of the `r`'th round (i.e., `ttss(ginit)(q)(sched(r))`) is the same as that of the untimed system at that point (i.e., `run(ginit)(r)(q)`—call this `s`), since this will be proved in the main theorem. What we then need to show is that at the start of the computation phase for round `r` (i.e., at time `sched(r)+P(r)`), `ttin(p)(sched(r)+P(r))(q)` has the same value as the message sent by `q` to `p` in the untimed system (i.e., `msg(q)(s, p)`). The necessary demonstration is therefore encoded in the following lemma.

```
ttin_prop: LEMMA
  FORALL (q: (in_nbrs(p))): LET s = run(ginit)(r)(q) IN
    ttss(ginit)(q)(sched(r)) = s
      IMPLIES ttin(p)(sched(r)+P(r))(q) = msg(q)(s ,p)
```

This is proved straightforwardly, using the formulas `ttin_ax`, `constraint1`, `max_delay2`, `comm_phase2`, `arrival_prop`, `recv_prop`, and `comp_phase`. The lemma generates a TCC to ensure that `p` is among the *out-nbrs* of `q`; as usual, this is discharged by appeal to the `channels_match` axiom.

The state of a processor `p` is unspecified during its computation phase, but at time `sched(r)` (i.e., the end of the `r-1`'st computation phase) it has the value that results from applying its transition function `trans(p)` to its state at the time when the computation phase began (i.e., in state `ttss(ginit)(p)(T)`, where `T` is `sched(r - 1) + P(r-1)`), and the array of incoming messages available at that time (i.e., `ttin(p)(T´)`). (In the special case `r = 0`, the state is `ginit(p)`.) This is specified by the following axiom. (This axiom generates a TCC to ensure that the occurence of `r-1` is nonnegative; this obligation is discharged automatically by the default proof strategy.)

```
ttss_start: AXIOM ttss(ginit)(p)(sched(r))
      = IF r = 0 THEN ginit(p)
        ELSE trans(p)(ttss(ginit)(p)(T), ttin(p)(T))
                WHERE T = sched(r - 1) + P(r-1)
        ENDIF
```

The state of each processor is held constant during the communication phase of a round, as specified in the following axiom.

```
ttss_comm: AXIOM sched(r) <= T AND T <= sched(r) + P(r)
    => ttss(ginit)(p)(T) = ttss(ginit)(p)(sched(r))
```

Using `ttss_start`, `ttss_comm`, `constraint1`, and `ttin_prop`, we can establish the following lemma (which is the antecedent to `ttin_prop`) by straightforward induction on `r`.

```
Main_Lemma: LEMMA ttss(ginit)(p)(sched(r)) = run(ginit)(r)(p)
```

We can now define `gs(r)`, the global start time for round `r` as in formulas (3) and (4) on Page 14 of Section 3.3.

```
gs(r): realtime

gs_ax: AXIOM (FORALL q: C(q, gs(r)) >= sched(r))
         AND (EXISTS p: C(p, gs(r)) = sched(r))
```

If `gi` is some initial global state then `gu(r)`, the global state of the untimed system at the start of round `r` is given by `run(gi)(r)`. Similarly, `gt(t)(p)`, processor `p`'s component of the global state of the time-triggered system at realtime `t` is given by `ttss(gi)(p)(C(p, t))`. These definitions are recorded as follows.

```
gi: global_init

gt(t)(p): state = ttss(gi)(p)(C(p, t))

gu(r): global_state = run(gi)(r)
```

The global state of the time-triggered system at time `gs(r)` is therefore `gt(gs(r))`, and our desired theorem can then be specified as follows.

```
Theorem_1: THEOREM gt(gs(r)) = gu(r)
```

This is proved straightforwardly using `gs_ax`, `Main_Lemma`, `ttss_comm`, `clock_sync`, `constraint3`, and `rho_prop2`.

A.2.1 LaTeX-printed Specification

time_triggered_system : THEORY
BEGIN

IMPORTING untimed_system

$p, q:$ VAR proc
$s:$ VAR state
$m:$ VAR mess

clocktime : TYPE = nat
realtime : TYPE = real

$T:$ VAR clocktime
$t, t_1, t_2:$ VAR realtime
$C(p, t):$ clocktime

monotone_clock : AXIOM $t_1 < t_2 \Rightarrow C(p, t_1) < C(p, t_2)$

$\rho : \{x : \text{real} \mid 0 < x \wedge x < 1\}$

drift_rate : AXIOM
$(1 - \rho) \times (t_1 - t_2) \leq C(p, t_1) - C(p, t_2) \wedge$
$C(p, t_1) - C(p, t_2) \leq (1 + \rho) \times (t_1 - t_2)$

$\Sigma :$ clocktime

clock_sync : AXIOM $\text{abs}(C(p, t) - C(q, t)) < \Sigma$

$i, j, r:$ VAR round

$\text{sched}(r):$ clocktime

monotone_sched : AXIOM $i < j \supset \text{sched}(i) < \text{sched}(j)$

$\text{dur}(r) : \text{clocktime} = \text{sched}(r + 1) - \text{sched}(r)$

$D(r):$ clocktime
$P(r):$ clocktime

constraint1 : AXIOM $0 < D(r) \wedge D(r) < P(r) \wedge P(r) < \text{dur}(r)$

constraint2 : AXIOM $D(r) \geq \Sigma$

$\text{sendtime}(p, r):$ realtime

sendtime_ax : AXIOM $C(p, \text{sendtime}(p, r)) = \text{sched}(r) + D(r)$

$\text{in\_comp\_phase}(t, p, r) : \text{bool} =$
LET $T = C(p, t)$ IN $\text{sched}(r) + P(r) \leq T \wedge T < \text{sched}(r + 1)$

comp_phase : LEMMA $i < j \Rightarrow \text{sched}(i) + P(i) < \text{sched}(j)$

ginit : VAR global_init

ttss(ginit)$(p)(T)$: state

sent$(p, (q : (\text{out_nbrs}(p))), m, t)$: bool
recv$(q, (p : (\text{in_nbrs}(q))), m, t)$: bool

comm_phase1 : AXIOM
$\forall\ (q : (\text{out\_nbrs}(p))) : \text{sent}(p, q, m, \text{sendtime}(p, r))$
WHERE $m = \text{msg}(p)(\text{ttss}(\text{ginit})(p)(\text{sched}(r)), q)$

comm_phase2 : AXIOM
$\forall\ (q : (\text{out\_nbrs}(p))) :$
$\text{sent}(p, q, m, t) \Rightarrow$
$\exists r : t = \text{sendtime}(p, r) \wedge m = \text{msg}(p)(\text{ttss}(\text{ginit})(p)(\text{sched}(r)), q)$

$\delta : \{x : \text{realtime} \mid 0 \le x\}$
delay : TYPE $= \{x : \text{realtime} \mid 0 \le x \wedge x \le \delta\}$
d : VAR delay

max_delay : AXIOM
$\forall\ p, (q : (\text{out\_nbrs}(p))), m, t :$
$\text{sent}(p, q, m, t) \Leftrightarrow \exists\ (d : \text{delay}) : \text{recv}(q, p, m, t + d)$

max_delay1 : LEMMA
$\forall\ p, (q : (\text{out\_nbrs}(p))), m, t :$
$\text{sent}(p, q, m, t) \Rightarrow \exists\ (d : \text{delay}) : \text{recv}(q, p, m, t + d)$

max_delay2 : LEMMA
$\forall\ q, (p : (\text{in\_nbrs}(q))), m, t :$
$\text{recv}(q, p, m, t) \Rightarrow \exists\ (d : \text{delay}) : \text{sent}(p, q, m, t - d)$

constraint3 : AXIOM $P(i) > D(i) + \Sigma + (1 + \rho) \times \delta$

rho_prop1 : LEMMA $(1 - \rho) \times d \ge 0$
rho_prop2 : LEMMA $(1 + \rho) \times d \ge 0$

constraint3_lemma : LEMMA $P(i) > D(i) + \Sigma + (1 + \rho) \times d$

in_comm_phase(t, p, r) : bool = LET $T = C(p, t)$ IN $\text{sched}(r) \le T \wedge T < \text{sched}(r) + P(r)$

arrival_prop : LEMMA in_comm_phase$(\text{sendtime}(q, r) + d, p, r)$

recv_prop : LEMMA
$\forall\ r, p, (q : (\text{in\_nbrs}(p))) :$
$\exists\ (m : \text{mess}), (t : \text{realtime}) : \text{in\_comm\_phase}(t, p, r) \wedge \text{recv}(p, q, m, t)$

ttin$(p)(T)((q : (\text{in_nbrs}(p))))$: mess

ttin_ax : AXIOM

$$\text{sched}(r) + P(r) \le T \wedge T < \text{sched}(r+1) \Rightarrow$$
$$\forall (q : (\text{in\_nbrs}(p))) :$$
$$\text{ttin}(p)(T)(q) =$$
$$\varepsilon!\ (m) : \exists\ t : \text{in\_comm\_phase}(t, p, r) \wedge \text{recv}(p, q, m, t)$$

$\text{ttin\_prop} :$ LEMMA
$$\forall\ (q : (\text{in\_nbrs}(p))) :$$
$$\text{LET } s = \text{run}(\text{ginit})(r)(q)$$
$$\text{IN ttss}(\text{ginit})(q)(\text{sched}(r)) = s \Rightarrow$$
$$\text{ttin}(p)(\text{sched}(r) + P(r))(q) = \text{msg}(q)(s, p)$$

$\text{ttss\_start} :$ AXIOM
$$\text{ttss}(\text{ginit})(p)(\text{sched}(r)) =$$
$$\text{IF } r = 0 \text{ THEN ginit}(p) \text{ ELSE trans}(p)(\text{ttss}(\text{ginit})(p)(T), \text{ttin}(p)(T))$$
$$\text{WHERE } T = \text{sched}(r-1) + P(r-1)$$
$$\text{ENDIF}$$

$\text{ttss\_comm} :$ AXIOM
$$\text{sched}(r) \le T \wedge T \le \text{sched}(r) + P(r) \Rightarrow$$
$$\text{ttss}(\text{ginit})(p)(T) = \text{ttss}(\text{ginit})(p)(\text{sched}(r))$$

$\text{Main\_Lemma} :$ LEMMA $\text{ttss}(\text{ginit})(p)(\text{sched}(r)) = \text{run}(\text{ginit})(r)(p)$

$\text{gs}(r) :$ realtime

$\text{gs\_ax} :$ AXIOM
$$(\forall\ q : C(q, \text{gs}(r)) \ge \text{sched}(r)) \wedge (\exists\ p : C(p, \text{gs}(r)) = \text{sched}(r))$$

$\text{gi} :$ global_init

$\text{gt}(t)(p) : \text{state} = \text{ttss}(\text{gi})(p)(C(p, t))$

$\text{gu}(r) : \text{global\_state} = \text{run}(\text{gi})(r)$

$\text{Theorem\_1} :$ THEOREM $\text{gt}(\text{gs}(r)) = \text{gu}(r)$

END time_triggered_system

B OM(1) As a Functional Program in PVS

The main additional detail needed to turn the treatment of Section 4 into valid PVS is to provide definitions or axioms for the *maj* (majority) function. In previous treatments of Oral Messages algorithms [26–28,34], we have specified this function axiomatically; here, however, a constructive definition is preferred because it allows greater automation in the proofs. The definition is based on the MJRTY algorithm of Boyer and Moore [4]; its formalization in PVS is due to Shankar.[7]

[7] See Shankar's Lecture 5 for the 1996 Marktoberdorf Summer School at URL `http://www.csl.sri.com/~shankar/marktoberdorf/marktoberdorf.html`.

B.1 MJRTY in PVS

MJRTY, discovered by Boyer and Moore in 1981 [3] but not published until ten years later [4], is a linear-time algorithm for finding the majority among values in an array, if such a majority (i.e., a value held by more than half the locations in the array) exists. The idea is to pair off and eliminate dissimilar values; the surviving value (if any) is the only candidate for the majority, and a second scan of the array is needed to check whether it does indeed have a majority.

This idea can be converted to the following pseudocode for calculating the majority candidate `cand` and its `lead` over all other values in the array `poll` (with indices `0...n-1`).

```
cand, lead := poll[0], 1;
FOR i = 1 TO n - 1 DO
  IF poll[i] = cand
    THEN lead := lead + 1
    ELSIF lead > 0
      THEN lead := lead - 1
    ELSE cand, lead := poll[i], 1
  ENDIF
```

We can convert this imperative program into a recursive function `mjrty` that returns the pair `(cand, lead)` as its value. The recursive function examines the array entries in the opposite order to the imperative program, but is otherwise a fairly direct transliteration.

```
mjrty(poll, i): RECURSIVE [T, nat] =
      IF i > 0
      THEN LET (cand, lead) = mjrty(poll, i - 1) IN
        IF poll(i) = cand THEN (cand, lead + 1)
        ELSIF lead > 0 THEN (cand, lead - 1)
        ELSE (poll(i), 1)
        ENDIF
      ELSE (poll(0), 1) ENDIF
MEASURE i
```

The candidate majority value is obtained by projecting the first element from the pair returned by `mjrty(poll, n-1)`.

One complication, however, is that the `maj` function we actually require must take an additional `participants` argument to indicate those members of the array that are to be considered in the majority vote. We can easily modify the core of the function (in the scope the `LET`) to be conditioned on `participants(i)`, but must be careful with the base case `i = 0`, since this might not be a member of `participants`. A more uniform treatment is possible if we allow the index `i` to go from `0` to `n` (rather than `n-1`) and then access the array at location `i-1` as follows.

```
preamble...

  mjrty(poll, participants, i): RECURSIVE [T, nat] =
        IF i > 0
        THEN (LET (cand, lead) = mjrty(poll, participants, i - 1) IN
             IF member(i - 1, participants)
             THEN
                  IF poll(i - 1) = cand THEN (cand, lead + 1)
                  ELSIF lead > 0 THEN (cand, lead -1)
                  ELSE (poll(i - 1), 1) ENDIF
             ELSE (cand, lead) ENDIF)
        ELSE (noname, 0) ENDIF
       MEASURE i

  maj(participants, poll): T = PROJ_1(mjrty(poll, participants, n))
```

The preamble declarations are shown below.

```
mjrty[T: NONEMPTY_TYPE, n: posnat]: THEORY
BEGIN

  cand, X: VAR T
  index: TYPE = below[n]
  p: VAR index
  vector: TYPE = [index -> T]
  poll, v, v1, v2: VAR vector

  participants: VAR setof[index]
  i: VAR upto[n]
  lead: VAR nat

  noname: T
```

The declaration `mjrty` generates four TCCs: three are subtype TCCs to ensure that the instances of `i - 1` are nonnegative, and the other is a termination TCC. All are proved automatically by the default strategies.

In order to tell whether the value returned by `maj` really does have a majority, we need to count the number of times that value occurs in the array among the `participants`. Because `mjrty` is defined by recursion, it is likely that verification of its properties will require proof by induction (on `i`), so the counting function should also be parameterized by `i`. This leads to the following specification. Here, `cardinality` is the name of the standard PVS library containing cardinality functions, and `fincardi` is the cardinality function for the first `i` members of an array on a finite type.

```
IMPORTING cardinality@finite_cardinality[below[n], n, id[below[n]]],
          cardinality@card_set[below[n], n, id[below[n]]]

count_votes(poll, participants, X, i): nat =
  fincardi((LAMBDA p: member(p, participants) AND poll(p) = X), i)
```

The key property of `mjrty` is captured in the following `invariant`. The lemma is proved automatically by the command `(induct-and-simplify "i")`.

```
invariant(poll, participants, i): bool =
  LET (cand, lead) = mjrty(poll, participants, i)
    IN FORALL X:
       2 * count_votes(poll, participants, X, i)
            <= fincardi(participants, i)
                 + IF X = cand THEN lead ELSE -lead ENDIF

invariant_holds: LEMMA invariant(poll, participants, i)
```

It is then easy to establish that no value other than the candidate returned by `mjrty` can be a majority.

```
correctness_step2: LEMMA
      LET (cand, lead) = mjrty(poll, participants, i)
        IN X /= cand
              IMPLIES 2 * count_votes(poll, participants, X, i)
                <= fincardi(participants, i)
```

This is proved automatically from the previous lemma by the command `(stew :lemmas "invariant_holds")`. See [37] for a description of the `STEW` strategy (which simply introduces the given lemmas and calls `grind`).

It then follows (by `(stew :lemmas "correctness_step2")`) that if there is a majority value, then it is the one produced by `maj`.

```
maj_correctness: LEMMA
   2 * count_votes(poll, participants, X, n) > fincard(participants)
        IMPLIES maj(participants, poll) = X
```

Finally, we can establish that the value of `maj` depends only on values of elements among the `participants`.

```
maj_ext: LEMMA
      (FORALL p: member(p, participants) IMPLIES v1(p) = v2(p))
          IMPLIES maj(participants, v1) = maj(participants, v2)
```

This is proved by expanding the definition of `maj` and applying induction to establish equality of the values of `mjrty` thereby revealed.

B.1.1 LaTeX-printed Specification

mjrty[T : NONEMPTY_TYPE, n : posnat] : THEORY
BEGIN

cand, X : VAR T

index : TYPE = below[n]
p : VAR index

participants : VAR setof[index]

vector : TYPE = [index $\rightarrow$ T]
poll, v, v_1, v_2 : VAR vector

i : VAR upto[n]
lead : VAR nat
noname : T

mjrty(poll, participants, i) : RECURSIVE [T, nat] =
  IF i > 0 THEN
   LET (cand, lead) = mjrty(poll, participants, $i - 1$) IN
     IF ($i - 1 \in$ participants)
      THEN IF poll($i - 1$) = cand
       THEN (cand, lead + 1)
      ELSIF lead > 0
       THEN (cand, lead − 1)
      ELSE (poll($i - 1$), 1)
      ENDIF
     ELSE (cand, lead)
     ENDIF
  ELSE (noname, 0)
  ENDIF
 MEASURE i

maj(participants, v) : T = PROJ_1(mjrty(v, participants, n))

IMPORTING cardinality@finite_cardinality[below[n], n, id[below[n]]],
     cardinality@card_set[below[n], n, id[below[n]]]

count_votes(poll, participants, X, i) : nat =
 fincardi((λ p : participants(p) $\wedge$ poll(p) = X), i)

invariant(poll, participants, i) : bool =
 LET (cand, lead) = mjrty(poll, participants, i)
  IN $\forall$ X :
    $2 \times$ count_votes(poll, participants, X, i) $\leq$
     fincardi(participants, i) + IF X = cand THEN lead ELSE − lead ENDIF

invariant_holds : LEMMA invariant(poll, participants, i)

correctness_step2 : LEMMA

$$\begin{array}{l} \text{LET } (\text{cand}, \text{lead}) \ = \ \text{mjrty}(\text{poll}, \text{participants}, i) \\ \quad \text{IN } X \neq \text{cand} \supset \\ \qquad 2 \times \text{count\_votes}(\text{poll}, \text{participants}, X, i) \leq \\ \qquad\quad \text{fincardi}(\text{participants}, i) \end{array}$$

$$\begin{array}{l} \text{maj\_correctness}: \ \text{LEMMA} \\ \quad 2 \times \text{count\_votes}(\text{poll}, \text{participants}, X, n) \ > \ \text{fincard}(\text{participants}) \supset \\ \qquad \text{maj}(\text{participants}, \text{poll}) = X \end{array}$$

$$\begin{array}{l} \text{maj\_ext}: \ \text{LEMMA} \\ \quad (\forall\ p: \ (p \in \text{participants}) \supset v_1(p) = v_2(p)) \supset \\ \qquad \text{maj}(\text{participants}, v_1) = \text{maj}(\text{participants}, v_2) \end{array}$$

END mjrty

B.2 Specifying and Verifying OM(1)

The properties required of OM(1) hold only if `n`, the number of processors, is greater than 3. We therefore undertake specification of OM(1) in PVS in the context of a theory in which `n` is constrained to have the appropriate type and `V`, the type of values, is uninterpreted. We can then import the corresponding instance of the `mjrty` theory, and introduce the types `processors`, and `pset` (for processor set). The uninterpreted constant `ok` identifies the set of nonfaulty processors. We need to ensure that this set includes at least `n-1` processors (i.e., at most one processor may be faulty); this constraint is encoded in the axiom `max_faults`.

```
OM_one[V: TYPE+, n: above(3)]: THEORY
BEGIN

  IMPORTING mjrty[V, n]
  v: VAR V

  processors: type = below(n)
  T, p, q: VAR processors

  pset: TYPE = setof[processors]
  fullpset: pset = fullset[processors]
  ok: pset
  max_faults: AXIOM fincard(ok) >= n-1
```

Next, we import the appropriate instances of the library theories dealing with cardinality reasoning for finite types, axiomatize the `send` function, and then specify `OM1` in essentially the same manner as (7) on Page 18.

```
IMPORTING cardinality@card_set[processors, n, id[processors]],
   cardinality@finite_cardinality[processors, n, id[processors]],
   card_set_more[processors, n, id[processors]]

rounds: TYPE = upto(1)
r: VAR rounds
send: [rounds, V, processors, processors -> V]

send_ax: AXIOM ok(p) AND ok(q) IMPLIES send(r, v, q, p) = v

OM1(T, v)(p): V =
  maj(remove(T,fullpset), LAMBDA q: send(1, send(0, v, T, q), q, p))
```

Specific instances of the Agreement property, such as the following, can be checked automatically by the proof command `(grind :rewrites ("send_ax"))`.

```
Agreement_instance: LEMMA
   n = 4 AND ok(2) AND ok(3) AND (ok(0) OR ok(1))
       IMPLIES OM1(1, t)(2) = OM1(1, t)(3)
```

Such specific instances as this could be examined by running `OM1` as a functional program—in fact, theorem proving and, in particular, rewriting is essentially being used as an execution mechanism in this case. A less specific instance of the Validity property can be constructed by fixing the number of processors at 4, but (implicitly) universally quantifying over the identity of the processors concerned, as follows.

```
Validity_instance:
  LEMMA n = 4 AND ok(p) AND ok(T) IMPLIES OM1(T, v)(p) = v
```

Execution would test only specific instances of a conjecture such as this, but the general case is proved automatically by the proof command

```
(STEW :REWRITES ("send_ax") :LEMMAS ("max_faults")).
```

The fully general cases of Agreement and Validity each require a lemma.

```
a: VAR [processors -> V]

ok_maj: LEMMA
  (FORALL p: ok(p) AND member(p, remove(q, fullpset)) => a(p) = v)
     IMPLIES maj(remove(q, fullpset), a) = v

Validity: THEOREM ok(p) AND ok(T) IMPLIES OM1(T, v)(p) = v
```

Here, `ok_maj` says that if all the nonfaulty (i.e., `ok`) processors among the receivers (i.e., in `remove(q, fullpset)`) agree on a value (i.e., `v`), than that value has a majority. Given this lemma (whose proof is a cardinality argument based on `max_faults`), the general `Validity` theorem is proved automatically by the command

```
(STEW :THEORIES ("OM_one") :EXCLUDE ("maj")
      :LEMMAS ("ok_maj") :LAZY-MATCH T)
```

The lemma `one_fault` needed for Agreement states that if processor `p` is faulty, then any other processor `q` must be nonfaulty. This follows by a cardinality argument from the `max_faults` axiom.

```
one_fault: LEMMA NOT ok(p) AND q /= p => ok(q)

Agreement: THEOREM ok(p) AND ok(q)
       IMPLIES FORALL (T: processors): OM1(T, v)(p) = OM1(T, v)(q)
```

The theorem `Agreement` is proved by cases according to whether or not `T` is faulty. If it is not (i.e., if it is `ok`), then the result follows from `Validity`; otherwise, it follows from `maj_ext` and `one_fault`.

```
(SKOSIMP*)
(CASE "ok(T!1)")
(("1" (GRIND :DEFS NIL :REWRITES (("Validity"))))
 ("2"
  (GRIND :THEORIES ("OM_one") :EXCLUDE "maj")
  (USE "maj_ext")
  (REDUCE)
  (USE "one_fault" :IF-MATCH ALL)
  (REDUCE)))
```

The theory `OM_one` generates nine TCCs to check that the various literal constants appearing in its formulas satisfy their subtype constraints. All are proved automatically by PVS's default strategy.

B.2.1 LaTeX-printed Specification

OM_one[V : TYPE+, n : above(3)] : THEORY
BEGIN

IMPORTING mjrty[V, n]

v : VAR V
processors : TYPE = below(n)
T, p, q : VAR processors
pset : TYPE = setof[processors]
fullpset : pset = fullset[processors]
ok : pset
max_faults : AXIOM fincard(ok) $\geq n - 1$

IMPORTING cardinality@finite_cardinality[processors, n, id[processors]],
cardinality@card_set[processors, n, id[processors]],
card_set_more[processors, n, id[processors]]

rounds : TYPE = upto(1)
r : VAR rounds

send : [rounds, V, processors, processors $\rightarrow$ V]

send_ax : AXIOM $\mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \mathrm{send}(r, v, q, p) = v$

$\mathrm{OM1}(T, v)(p) : V = \mathrm{maj}((\mathrm{fullpset} \setminus \{T\}),\ \lambda\, q : \mathrm{send}(1, \mathrm{send}(0, v, T, q), q, p))$

Agreement_instance : LEMMA
$n = 4 \wedge \mathrm{ok}(2) \wedge \mathrm{ok}(3) \wedge (\mathrm{ok}(0) \vee \mathrm{ok}(1)) \supset$
$\forall v : \mathrm{OM1}(1, v)(2) = \mathrm{OM1}(1, v)(3)$

Validity_instance : LEMMA $n = 4 \wedge \mathrm{ok}(p) \wedge \mathrm{ok}(T) \supset \mathrm{OM1}(T, v)(p) = v$

a : VAR [processors $\rightarrow$ V]

ok_maj : LEMMA
$(\forall\, p : \mathrm{ok}(p) \wedge (p \in (\mathrm{fullpset} \setminus \{q\})) \Rightarrow a(p) = v) \supset$
$\mathrm{maj}((\mathrm{fullpset} \setminus \{q\}), a) = v$

Validity : THEOREM $\mathrm{ok}(p) \wedge \mathrm{ok}(T) \supset \mathrm{OM1}(T, v)(p) = v$

one_fault : LEMMA $\neg \mathrm{ok}(p) \wedge q \neq p \Rightarrow \mathrm{ok}(q)$

Agreement : THEOREM $\mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \forall\, T,\ v : \mathrm{OM1}(T, v)(p) = \mathrm{OM1}(T, v)(q)$

END OM_one