SRI International

CSL Technical Report • February 2003, updated February 2004

Preliminary Formal Analysis of TTA Startup

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

This research was supported by NASA Langley Research Center under contract NAS1-00079 and by the Next TTA project of the European Union.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

We formally specify the TTA startup algorithm of Paulitsch and Steiner in the SAL language. Using the SALenv model checker, we confirm that the algorithm succeeds in starting the cluster after at most one collision. We explore alternative algorithms and system parameters, and investigate behavior in the presence of faults. The limitations of finite-state modeling are discussed, together with plans for more comprehensive analysis and verification.

Contents

Chapter 1

Introduction

When first powered up, TTA controllers operate as autonomous, self-timed entities and their initial task is to synchronize themselves so that they function as a coordinated whole. Each TTA controller has a mapping from its local time to the "slots" in the cyclic "Time Division Multiple Access" (TDMA) schedule of TTA; in particular, each controller i will know the local time at which the slot for controller p begins—we can denote this time by $s_i(p)$. Synchronization simply means that for any two nonfaulty controllers i and j, the instants when i's local clock reads $s_i(p)$ and when j's clock reads $s_j(p)$ should occur very close together in real time (within the acceptable clock skew parameter Π). The *startup problem* is to establish values for the functions s_i (or, equivalently, for their local clocks) as the controllers first power up so that they quickly become synchronized. The *synchronization problem* is to adjust the values of s_i (or, equivalently, their local clocks) so that the controllers remain synchronized despite the drift of their hardware clocks (due to their oscillators operating at slightly different rates). The *restart problem* is to reestablish synchronization after transient faults have afflicted one or more (or all) controllers.

The synchronization problem is well understood and many algorithms to solve it have been developed, analyzed, and formally verified [LMS85, Min93, Sha92, WL88]. The algorithm employed in TTA belongs to the general class of averaging synchronization algorithms [Sch87]: each controller i estimates its skew relative to each controller p by comparing the reading of its local clock at the instant when the message in slot p arrives against $s_i(p)$ (adjusted for known transmission delays). It then adjusts s_i (or, equivalently, its local clock) by some fault-tolerant average of its estimates of its skews to all (or, in the case of TTA, some) other controllers. This algorithm has been formally verified [PSvH99].

One way to construct a startup algorithm is as a variation on a synchronization algorithm: even if the local clocks of various controllers are initially far apart, successive rounds of averaging should bring them into convergence. This is plausible, but Miner [Min93] presents scenarios in which the controllers do not converge to a single coordinated entity, but instead divide into two "cliques," each synchronized within itself but unaware of the

existence of the other. It is possible that some startup algorithms of this kind are correct, and can be formally verified to be so, but the analysis seems difficult.

An alternative approach uses a startup algorithm that is separate from the synchronization algorithm. The basic idea is that each newly started controller p listens for at least one round to determine whether there is activity on the communications bus (a TTA cluster consists of controllers connected to a bus, or to a star network that functions as a bus). If there is, it synchronizes to the ongoing traffic; if not, it sends out a "cold start" message bearing its own identity and adjusts s_p (or equivalently, its local clock) so that its current local time equals $s_p(p)$. Other controllers that receive this message synchronize to it (i.e., each controller j arranges that $s_j(p)$ equals its local clock at the time the cold start message from p was received). There is, of course, the possibility that two controllers p and q send out simultaneous or overlapping cold start messages. The other controllers will see this as noise and will ignore it, but p and q (assuming they cannot listen while transmitting) will not know whether their messages were received or not. They each therefore wait for a round or so to see if messages from other controllers indicate that their startup was successful (i.e., p checks for messages from controllers i that arrive when p's clock reads close to $s_p(i)$). If not, each sends another cold start message, but to avoid a second collision, it delays this for a time equal to its own slot position.

The startup algorithm sketched above is essentially that which Paulitsch and Steiner [PS02] have proposed for Next TTA. Paulitsch and Steiner suggest specific values for the various timeouts involved and argue that the system will synchronize after at most one collision. They also examine scenarios in the presence of various faults and show that most of these can be controlled satisfactorily only in TTA architectures that employ a central star or hub. Such a central hub not only must provide detection and masking of faults in its attached controllers, but must do so while itself attempting to synchronize with those controllers.

The criticality and subtlety of TTA startup algorithm, and the importance of accurately identifying the fault detection and masking requirements on the central hub suggest that formal analysis and verification could be useful. In the next chapter, we present a formal specification of the startup algorithm in the SAL language, and we describe use of the SALenv model checker to confirm some properties of the algorithm.

Chapter 2

Formal Analysis with SAL

The paper by Paulitsch and Steiner [PS02] identifies the startup problem as that of transitioning from asynchronous to synchronous system operation. If correct, this observation could influence formal modeling for this problem domain, since asynchronous and synchronous composition are generally modeled by quite different mathematical constructions (disjunction and conjunction, respectively).

In fact, the terms "asynchronous" and "synchronous" carry different connotations in different areas of computer science. Here, I think Paulitsch and Steiner intend the connotation used in distributed systems—and I think they are incorrect under this connotation to describe the initial operation of a TTA cluster as asynchronous. In distributed systems, a *synchronous system* is one in which there are known bounds on the time that it can take for messages to be delivered, or for computation steps to be performed; an *asynchronous system* is one in which there are no such bounds, and it is provably impossible to achieve many forms of coordinated behavior (e.g., consensus [FLP85, DDS87]) in a truly asynchronous system. Under these connotations of the terms, it is clear that a TTA cluster at startup has the character of a synchronous rather than an asynchronous system: it is synchronous, but *unsynchronized*. Given that there are known bounds on the time taken for computation and communication, the challenge to the startup algorithm in TTA is to ensure that it terminates successfully in some bounded time (i.e., it must not admit the possibility of endless collisions on the bus). This is what Paulitsch and Steiner's algorithm aims to achieve, and it is the property we will examine here.

The following description of the TTA startup algorithm is taken verbatim from [PS02, page 332]; to be consistent with their terminology, I henceforth refer to each TTA controller as a *node* (they also use "frame" where I use "message"). The algorithm is described as a state machine with four states; the states and their outgoing transitions are described in the following list. The various timeouts are defined later.

Init State: After power-up, a node starts in Init State. After its internal initialization it transits to Listen State.

Listen State: The node starts its Listen Timeout and begins listening on the bus. If an incorrect frame or any noise has been received while listening, the Listen Timeout is restarted. If a correct frame has been received, the receiving node accepts the global time and node ID of the incoming frame. Then, the node enters Active State. If no traffic has been detected on the communication medium while the node was listening and the Listen Timeout expires, the node enters Cold Start State.

Cold Start State: The node generates a frame using its local time as global time and its node ID (we call such a frame a Cold Start Frame). The node then starts its Cold Start Timeout, begins sending the Cold Start Frame on the communication medium, proceeds the transmission schedule for one TDMA round, and listens for a frame on the communication medium. If during this TDMA round no frames from other nodes have been received, the node expects that no other node was in Listen State. It waits for its Cold Start Timeout to expire (i.e., it waits for the duration of the remaining unique Startup Timeout) and sends another Cold Start Frame. If during this TDMA round a correct frame has been received and the global time in this frame equals the node's view of the global time, the node enters the Active State. The node enters Listen State if it received a faulty frame, other noise has been detected, or it received a correct frame that contains a global time different from the node's view of the global time.

Active State: In Active State the node is in synchronized operation and proceeds its transmission schedule cyclically, that is, it waits until the time for its slot is reached and starts sending.

There is no mention of a central star or hub in this description, but it is implicit in one of the assumptions made: namely, "simultaneous" messages will result in a collision that is perceived identically at all nodes. This assumption may not be valid for a true bus: consider a scenario where one node is finishing transmission of a cold start message just as another node, at the other end of the bus, is starting transmission of its own cold start message; due to propagation delays, other nodes near the former may not see a collision, whereas those nearer to the latter may do so. Without consistent collision detection, it does not seem possible to implement a correct startup algorithm.[1] The central hub is introduced, in part, to supply the consistent collision detection that is absent in a true bus. In startup mode, the hub monitors all its input ports for incoming messages. When one appears, the hub will perform some signal boosting and reshaping while directly transferring the bits constituting the message to all its output ports (i.e., it operates as a crossbar switch; it does not store and forward the messages). If other input messages appear while the hub is already

[1] The reader may ask how protocols such as Ethernet are feasible if collision detection is unreliable; the answer is that an Ethernet may experience some kind of glitch once every few million hours due to unreliable collision detection but this is acceptable to the kinds of applications that use Ethernet. But TTA is used for safety-critical functions where we require at most one failure in 10 billion or more hours of operation (i.e., 10^{-10} or better).

transferring one, it may either continue the current transfer (ignoring the new messages), or terminate it (and either ignore the new messages, or arbitrate among them and select one for transfer). A message that is terminated prematurely will appear to all receiving nodes as noise (because all TTA messages carry a checksum and an incomplete message will not satisfy the checksum test).

The timeout parameters in the algorithm are specified as follows [PS02, page 331] (actually Paulitsch and Steiner use a more detailed system model and define these timeouts by formulas, but the intuition is that given here).

Startup Timeout: This is unique to each controller; it is defined as the time at which the node's slot starts, measured from the beginning of the round. This timeout is not actually used in the algorithm.

Cold Start Timeout: This is equal to the Startup Timeout plus the duration of a complete round.

Listen Timeout: This is equal to the Startup Timeout plus the duration of two complete rounds.

Our goal is to formally specify the algorithm and its timeouts as described above, and to check that it satisfies certain desired properties. To begin, we wish to check that, in the nonfaulty case, the system synchronizes successfully after at most one collision.

2.1 Fault-Free Case

The first choice to be made in formal specification is the choice of system model. Here, the algorithm involves time in an essential way, and the most realistic system model will be one in which time is treated as a continuous variable. Timed automata [AD94] might therefore be a suitable model; these can be encoded in PVS [ORSvH95] and formally verified "by hand" or with the aid of specialized libraries and strategies such as those of TAME [Arc00], or we could use a model checker for timed automata such as Kronos [BDM+98] or UPPAAL [LPY97], or an experimental encoding in SAL/ICS [dMRS]. Lönn [Lön99a,Lön99b] considers startup algorithms for TDMA systems similar to TTA and verifies one of them using UPPAAL [LP97], [Lön99b, Chapter 9].

Although a timed automaton model would be more realistic, we have chosen, in the interest of simplicity, to use a purely finite state model for the preliminary analysis reported here. Nodes executing the startup algorithm measure time by counting off slots in the TDMA schedule. If we imagine slots shrunk to atomic instants, then a discrete model of time is adequate. We do not care by how much the slots at different nodes are offset, just whether they overlap at all (so that a collision can occur). Hence, we can model the collective behavior of a cluster of nodes as synchronous composition of discrete systems. Another way to justify this modeling approach is to think of it as describing the system

seen from the hub's point of view: each discrete instant corresponds to some real time interval at the hub and all messages that (start to) arrive in that interval are regarded as simultaneous; the behavior of the nodes is driven off (i.e., synchronously composed with) the discretization provided by the hub.

The system model will comprise n nodes, each synchronously composed with a central hub. At each time step, each node examines the input message received from the hub, consults its private state variables, and generates an output message that it sends to the hub. The rôle of the hub is to act like a broadcast bus (so that the description below often speaks of "the bus"), except that it must suppress certain faulty node behaviors, and it must ensure consistent message receptions across all nonfaulty nodes. In particular, at each time step, the hub examines the messages output from each node and constructs the single message that will comprise the consistent input to the nodes at the next time step.

We can specify this discrete, synchronous model in the language of SAL [BGL+00] as follows. We begin by defining the types over which the state variables will range.

```
startup: CONTEXT =
BEGIN
n: NATURAL = 3;
index: TYPE = [0..n-1];
maxcount: NATURAL = 3*n-1;
counts: TYPE = [0..maxcount];
states: TYPE = {init, listen, start, active};
msgs: TYPE = {quiet, normal, noise};
```

Here, `n` is the number of nodes, which are identified by elements of the type `index`. The largest timeout that needs to be considered is `maxcount`, and the values of a timeout counter are given by the type `counts`. The enumerated types `states` and `msgs` specify, respectively, the four states in the algorithm, and the kinds of messages that can be exchanged with the hub. An individual node will only output messages with values `quiet` (meaning no message), or `normal`; the hub can return values `quiet`, `normal`, or `noise`, indicating that it received simultaneous transmission by zero, one, or more nodes in the given slot.

It is useful to have a function `incslot` that calculates the index of the next slot in the TDMA round.

```
incslot(r: index): index = IF r=n-1 THEN 0 ELSE r+1 ENDIF;
```

We next specify the input and output variables of an individual node as follows.

```
node[i:index]: MODULE =
BEGIN
INPUT
  busmsg: msgs,
  bustime: index
OUTPUT
  msg: msgs,
  slot: index,
  state: states,
  counter: counts
```

The `busmsg` represents the kind of message that the node reads from the hub; if it is a normal message, then `bustime` indicates the sender's identity (i.e., its index), and hence the current time measured relative to the start of the TDMA round. We can think of this information as being included in the message, but it is easier to model it as a separate variable.

The output variables `msg`, `slot`, `state`, and `counter` represent, respectively, the message that this node will output to the hub, its estimate of the identity of the node associated with the current slot (i.e., its estimate of time relative to the start of the TDMA round), its state within the algorithm, and the value of its timeout counter.

The various timeouts are specified as local variables.

```
LOCAL
  ctimeout, ltimeout: counts
DEFINITION
  ctimeout = n+i;
  ltimeout = n+n+i
```

The output state variables are initialized as follows.

```
INITIALIZATION
  msg = quiet;
  slot = 0;
  state = init;
  counter = 0
```

The algorithm is specified by a series of guarded commands. We begin with those that apply to a node in the `init` state.

```
TRANSITION
[  % Case 1
   state = init AND counter < n -->
    counter' = counter+1
[] % Case 2
   state = init -->
    state' = listen;
    counter' = 0
```

Here, the [character introduces a set of guarded commands that are separated by the [] symbol, and the % character introduces a comment. A SAL guarded command is eligible for execution in the current state if its guard (i.e., the part before the --> arrow) is true. The SAL model checker nondeterministically selects one of the enabled commands for execution at each step; if no commands are eligible, the system is deadlocked. Primed state variables refer to their values in the new state that results from execution of the command, and unprimed to their old (pre-execution) values.

Here, so long as the counter is less than n, both the above commands are eligible for execution; thus, the node can nondeterministically choose to stay in the init state (incrementing its counter by 1), or to transition to the listen state. If the counter reaches n, the node must transition to the listen state. Hence, the two guarded commands above allow the node to "wake up" and transition to the listen state at any point during the first round; on entering the listen state, its counter is reset to zero.

Behavior in the listen state is specified in four cases as follows. If a collision was detected by the hub, the busmsg will be set to the value noise; in this case the node resets its timeout counter and sets its output msg to quiet, meaning that it will not broadcast in the next slot, and remains in the listen state.

```
[] % Case 3
   state = listen AND busmsg = noise -->
    counter' = 0;
    msg' = quiet
```

If a normal message is received in the current slot, then it indicates that another node is attempting to start the bus, or that normal traffic is already in progress. Hence, the receiving node transitions to the active state. The bustime value carried with the message indicates the identity of the sender, and thus the current value of time as measured from the start of the TDMA round; hence, the correct value of slot in the next step will be the successor to bustime, and the node uses the incslot function to compute this. If that value corresponds to this node's identity (i.e., if slot' = i), then it must output a normal message; otherwise it should stay quiet. The node also resets its counter to zero; this has no effect on the algorithm (since the counter is not used in the active state), but it reduces the number of trivially different states that will need to be considered during model checking.

```
[] % Case 4
   state = listen AND busmsg = normal -->
     state' = active;
     counter' = 0;
     slot' = incslot(bustime);
     msg' = IF slot' = i THEN normal ELSE quiet ENDIF
```

If a quiet message is received in the current slot and this node's listen timeout has not expired (i.e., counter < ltimeout), then the node simply increments its counter and stays silent.

```
[] % Case 5
   state = listen AND busmsg = quiet AND counter < ltimeout -->
     counter' = counter+1;
     msg' = quiet
```

If, however, the listen timeout expires with no message having been received, then this node will transition to its cold `start` state and output a `normal` message. Its `slot` variable (which will be sent with the message) is set to this node's identity, and its `counter` is reset to zero.

```
[] % Case 6
   state = listen AND busmsg = quiet AND counter = ltimeout -->
     state' = start;
     counter' = 0;
     slot' = i;
     msg' = normal
```

If the cold start timeout expires (i.e., `counter = ctimeout`) while the node is in the cold `start` state, then it sends another cold start message by repeating the actions of Case 6.

```
[] % Case 7
  state = start AND counter = ctimeout  -->
     counter' = 0;
     slot' = i;
     msg' = normal
```

If the node is in the cold `start` state, the cold start timeout has not expired, and the bus is `quiet`, it simply increments its `counter` and `slot` variables and remains `quiet`. In this state, the node is waiting for confirmation from another node that it has succeeded in starting the bus and so it is waiting for a `normal` message; however, it is important that it should not respond to its *own* cold start message; hence, the check `counter = 0` in the guard of this command.

```
[] % Case 8
  state = start AND counter < ctimeout
          AND (busmsg=quiet OR counter = 0) -->
     counter' = counter+1;
     slot' = incslot(slot);
     msg' = quiet
```

If a normal message is received from a node *different* than this node (i.e., `counter > 0`), and if the TDMA position indicated by that node is the same as that calculated by this one (i.e., `bustime = slot`), then the node takes this as confirmation that it was successful in starting the bus and it transitions to the `active` state. It sets the `counter`, `slot`, and `msg` variables as in Case 4.

```
[] % Case 9
  state = start AND counter < ctimeout AND counter > 0
         AND busmsg = normal AND bustime = slot -->
     counter' = 0;
     state' = active;
     slot' = incslot(slot);
     msg' = IF slot' = i THEN normal ELSE quiet ENDIF
```

If a `normal` message is received, but it indicates a different TDMA position than that calculated by this node (i.e., `bustime /= slot`), or if `noise` is received, the node resets its `counter` and returns to the `listen` state. The variable `slot` is set to zero to reduce the size of the statespace to be explored during model checking.

```
[] % Case 10
  state = start AND counter < ctimeout AND counter > 0
         AND (busmsg=noise OR
               (busmsg = normal AND bustime /= slot)) -->
     counter' = 0;
     slot' = 0;
     msg' = quiet;
     state' = listen
```

In the `active` state, a node simply increments its `slot` for every time interval that passes, and sends a `normal` message when its turn comes.

```
[] % Case 11
  state = active -->
     slot' = incslot(slot);
     msg' = IF slot' = i THEN normal ELSE quiet ENDIF
]
```

We now need to specify the hub; this receives a `msg` and an `index` (corresponding to time measured as a position in the TDMA round) from each node, and delivers a single value of each type. We specify the inputs `inmsgs` and `intimes` as arrays of the corresponding types; the outputs are the simple values `outmsg` and `outtime`. Because we want to count how many message collisions occur, we add `collisions` as an additional output variable.

```
hub: MODULE =
BEGIN
INPUT
  inmsgs: ARRAY index OF msgs,
  intimes: ARRAY index OF index
OUTPUT
  collisions: [0..2],
  outmsg: msgs,
  outtime: index
INITIALIZATION
  collisions = 0;
  outmsg = quiet;
  outtime = 0
```

The output values of the hub are most conveniently specified as guarded commands. We specify the primed (i.e., new) values of `outmsg` and `outtime` in terms of other primed variables. This means that the hub behaves as a combinational circuit (i.e., takes no time). This is safe because SAL checks that no combinational (zero time) loops are introduced thereby, and convenient because it means that each "tick" of the clock corresponds to a complete interaction between the nodes and the hub.

The first case is when two or more nodes send a normal message and a collision occurs.

```
TRANSITION
[  % Case A
   (EXISTS (i,j: index): i/=j AND
       inmsgs'[i] = normal AND inmsgs'[j] = normal) -->
    outmsg' = noise;
    outtime' = 0;
    collisions' = IF collisions=2 THEN 2 ELSE collisions+1 ENDIF
```

In this case, we set `outmsg' = noise` and we set `outtime` to zero (to reduce the statespace for model checking, it does not affect the algorithm), and increment `collisions` by one.

If a single normal message is received at the hub, then we want to find the node that sent that message and set `outtime` equal to the value of `intimes` at that node. We use the

```
([] (i:  index):  ...
```

construction to create a set of guarded commands, one for each value of the index `i`. Only the command corresponding to the node `i` that sent the message will satisfy the guard and be eligible for execution, and it is the value of `intimes` from that node that will be assigned to `outtime`.

```
[] % Case B
   ([] (i: index): inmsgs'[i] = normal -->
     outmsg' = normal;
     outtime' = intimes'[i])
```

Now it might seem that we should augment the guard to this command to be sure that a normal message was sent by exactly one node (i.e., we should conjoin the negation of the guard from the previous case), for without such a restriction, this command also applies when there is a collision. Further consideration, however, should convince us to leave the command as it stands. The reason is that our discrete model will otherwise exclude some behaviors that are possible in the more accurate continuous time model. Suppose two nodes both send cold start messages and that one finishes its transmission around the time that the other starts. The hub, depending on the details of its implementation and the exact timing, could treat this as a collision, or it could relay the first and suppress the second, or it could relay both messages. The first of these corresponds to a collision as modeled here; the last corresponds to the messages being sent in different slots as modeled here,[2] but the middle choice is not available in our model unless we allow the command above when there are normal messages from more than one node. Hence, our SAL model specifies Case B exactly as stated above.

If no node sends a message, then we will have `outmsg' = quiet`, and in this case we simply set `outtime` to zero (to reduce the statespace for model checking). This case concludes the transitions for the hub.

```
[] % Case C
   (FORALL (i: index): inmsgs'[i] = quiet) -->
     outmsg' = quiet;
     outtime' = 0
]
END;
```

Now we need to "wire up" the `n` nodes and the single hub. This is accomplished in the following specification.

[2] Actually, this is not quite correct: if the messages were sent in different slots, why did the second node not hear the transmission by the first and respond appropriately? In the timed model, this could be because the second had already taken the decision to send and was not even listening for incoming messages at that point. We cannot reproduce this behavior in the discrete model as it stands, but it becomes feasible when we extend the modeling to include faults—see the discussion on "deafness" in Section 2.3 on page 16.

```
system: MODULE =
 hub || (WITH OUTPUT inmsgs:  ARRAY index OF msgs;
              OUTPUT lstates: ARRAY index OF states;
              OUTPUT intimes: ARRAY index OF index;
              OUTPUT lcounts: ARRAY index OF counts
                (|| (i: index):  RENAME msg     TO inmsgs[i],
                                        slot    TO intimes[i],
                                        state   TO lstates[i],
                                        counter TO lcounts[i],
                                        bustime TO outtime,
                                        busmsg  TO outmsg
                         IN  node[i]));

END
```

Here, we specify that the hub is synchronously composed with an n-fold synchronous composition of nodes specified by the
`(|| (i: index): ...`
construction. The `WITH` construction introduces arrays for each of the four output state variables used in the nodes; the choice of names `inmsgs` and `intimes` automatically will cause these to be "wired up" to the corresponding inputs of the hub (otherwise we would have to `RENAME` them). The `RENAME` construction picks out the member of each state variable array to be "wired up" to this (`i`'th) node, and associates the `outtime` and `outmsg` outputs of the hub with the `bustime` and `busmsg` inputs of every node.

We have now completed specification of the startup algorithm. Before proceeding to state the properties expected of it, we should first make sure that the specification does not deadlock (i.e., have reachable states with no successor). We can do this using the SAL deadlock checker as follows.

```
sal-deadlock-checker startup system
./a.out

Deadlock traces weren't detected.
```

Here, `startup` is the name of the SAL context (in the file `startup.sal`), and `system` is the name of the module we wish to check for deadlocks. The `sal-deadlock-checker` command compiles the SAL specification into a program that checks it for deadlocks, and we run the compiled program as `./a.out`. We are relieved to see this produce the output indicating that no deadlocks were detected. Earlier versions of this specifications did deadlock, and the reader can easily reproduce deadlocked behavior by removing some of the guarded commands or modifying some of the guards.

Now we can specify properties of the algorithm that we wish to check. The first of these is the requirement that every node should eventually reach the `active` state. We specify this as follows.

```
ok: LEMMA system |- F( G(FORALL (i:index): lstates[i] = active));
```

Here `F` and `G` represent the *eventually* and *always* modalities of linear temporal logic (often written as a diamond and a box, respectively). This property states that eventually all nodes reach the `active` state and stay there.

We compile and run a model checker for this property as follows (the `--optimize` flag simply improves efficiency and allows larger models to be checked).

```
sal-model-checker --optimize startup ok
./a.out

verified
```

Here, `startup` is the SAL context concerned, and `ok` is the property to be checked. As with the deadlock checker, we run the compiled model checker as `./a.out`, and are gratified to see that this property is verified. Additional properties that also are verified are the following.

```
fast: LEMMA system |- G(collisions <= 1);

sync: LEMMA system |- G(FORALL (i, j: index):
   lstates[i] = active AND lstates[j] = active
      => intimes[i] = intimes[j]);
```

The `fast` property checks that there is never more than a single collision, while the `sync` property ensures that all active nodes agree on the time (i.e., where they are in the TDMA schedule).

To be sure that our checks are not vacuous for some reason, we should check some properties that ought to be false. A suitable one is the following, which asserts that collisions never occur.

```
optimism: LEMMA system |- G(collisions = 0);
```

Model checking this formula reveals a counterexample. To make the counterexample as short and compact as possible, we can run the compiled model checker with the following command.

```
./a.out --bfs --compact-trace
```

Here,`--bfs` causes the search to be performed in breadth-first order. This produces the counterexample shown below. We see that nodes 2 and 3 immediately transition to the `listen` state, and node 0 does so one slot later. They each then count to their listen timeouts, which nodes 0 and 1 reach at the same time (the timeout is one less for node 0, but it started one slot later) and thereby generate a collision when they enter their cold `start` states and send their messages.

```
Counter-example detected:
outtime = 0
outmsg = quiet
collisions = 0
flags = [0 := FALSE, 1 := FALSE, 2 := FALSE]
inmsgs = [0 := quiet, 1 := quiet, 2 := quiet]
intimes = [0 := 0, 1 := 0, 2 := 0]
lstates = [0 := init, 1 := init, 2 := init]
lcounts = [0 := 0, 1 := 0, 2 := 0]
----------------
Transition: <[startup:139], <[startup:43], [startup:43], [startup:47]>>
lstates = [0 := init, 1 := listen, 2 := listen]
lcounts = [0 := 1, 1 := 0, 2 := 0]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:43]>>
lstates = [0 := listen, 1 := listen, 2 := listen]
lcounts = [0 := 0, 1 := 1, 2 := 1]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:60]>>
lcounts = [0 := 1, 1 := 2, 2 := 2]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:60]>>
lcounts = [0 := 2, 1 := 3, 2 := 3]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:60]>>
lcounts = [0 := 3, 1 := 4, 2 := 4]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:60]>>
lcounts = [0 := 4, 1 := 5, 2 := 5]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:60]>>
lcounts = [0 := 5, 1 := 6, 2 := 6]
----------------
Transition: <[startup:139], <[startup:60], [startup:60], [startup:60]>>
lcounts = [0 := 6, 1 := 7, 2 := 7]
----------------
Transition: <[startup:145], <[startup:60], [startup:64], [startup:64]>>
outtime = 0
outmsg = noise
collisions = 1
flags = [0 := FALSE, 1 := FALSE, 2 := FALSE]
inmsgs = [0 := normal, 1 := normal, 2 := quiet]
intimes = [0 := 0, 1 := 1, 2 := 0]
lstates = [0 := start, 1 := start, 2 := listen]
lcounts = [0 := 0, 1 := 0, 2 := 8]
```

2.2 Design Exploration

One of the attractions of model checking is that it facilitates exploration of alternative designs and design parameters. In this regard it is like simulation but, because it explores *all* possible behaviors, it is much more reliable. Here, we can check that the choice of `n+i` as the cold start timeout parameter is crucial. If we set this timeout to `n+n` (long enough, but not personalized to each node), or to `i` (personalized to each node, but not long enough), we discover counterexamples, as expected.

However, the choice of `n+n+i` as the listen timeout parameter turns out not to be critical. If we remove the personalization from the listen timeout and set it to `n+n`, the algorithm continues to work; experimentation shows that it also works with this timeout set to `n` or `n+i`, but more than a single collision can occur in these cases. However, `n+1` seems a perfectly good choice for the listen timeout as all three properties are verified for this value of the parameter.[3]

Recall, from the discussion on page 12 concerning Cases A and B in the model of the hub, that both of these commands are eligible when there are collisions. Case B also handles other circumstances and is plainly necessary, but it is interesting to inquire what happens if Case A is removed. Perhaps to our surprise, we find that all three properties are verified—and remain so when the listen and cold start timeouts are both set to `n`.

On reflection, this is not surprising: with Case A removed, we are postulating a hub that never transmits `noise` in response to a collision, but rather simply arbitrates between any colliding messages and forwards just one of them. This arbitrated message will succeed in starting all other nodes that are in the `listen` state; those that are in the `init` state will be started by the ongoing traffic as soon as they enter the `listen` state; and those that are in the `start` state (which will be those whose messages collided with this and lost the arbitration) will drop back to `listen`, and then be started by the ongoing traffic. This behavior seems preferable to that of a hub that generates noise messages in response to collisions,[4] and the concrete design of such a hub seems perfectly feasible.

2.3 Behavior in Presence of Faults

Paulitsch and Steiner consider three classes of faulty nodes: those that violate their timeouts, "deaf" nodes, and those that "masquerade" as other nodes. We start by considering "deaf" nodes, which are those that fail to recognize incoming messages.

A limited form of this behavior would be nodes that can fail to respond to incoming messages in slots where they are preparing to send a cold start message. Notice that this is the behavior discussed on the footnote on page 12 that was previously "missing" from

[3] This is not true for $n > 3$; please see the update on page 23.

[4] Although the modified design generally starts the bus up faster than the original, it takes just as long in the worst case. This worst case is when all nodes send a cold start message at the same time; all then listen for a confirmation, which never comes, and one of them must send a second cold start message to start things up.

our discrete model. This behavior can be specified by adding a guarded command similar to Case 6, but which ignores the incoming `busmsg`. We add a local `BOOLEAN` variable `simpledeafness` and conjoin this to the guard so that we have a simple way to turn this behavior on and off (by initializing the variable to `TRUE` or `FALSE`, respectively).

```
[] % Case 6a: simple deafness
   simpledeafness AND
   state = listen AND counter = ltimeout -->
     state' = start;
     counter' = 0;
     slot' = i;
     msg' = normal
```

We find that this modified system continues to satisfy the `ok` and `sync` properties,[5] but it violates `fast` (i.e., all nodes eventually correctly synchronize, but there may be more than one collision). By checking the property `fairlyfast`, we are able to verify that at most two collisions will occur.

```
fairlyfast: LEMMA system |- G(collisions <= 2);
```

However, if we increase the number of nodes `n` to 4, then this property also is violated; in general, a system with `n` nodes can suffer `n-1` collisions.

Intuitively, the reason the system continues to work in the presence of this form of "deafness" is that all other nodes that are in their `listen` states will synchronize (and enter their `active`) states on receipt of the cold start message from the first node. The second (deaf) node will not receive any messages acknowledging its startup attempt (unless it fortuitously synchronized with the first), so it will drop back to the `listen` mode and then synchronize with the ongoing traffic. A second collision can occur because, in a run that has one ordinary collision, a deaf node may fail to hear the next cold start message and then sends one of its own, colliding with the message of a node that is acknowledging the other one.

We can also verify that the "arbitrated" version of the system (i.e., the system without Case A of the hub model) also works in the presence of this type of deafness but, like the standard version, it can require additional rounds to start the bus in the presence of deafness.

Although the behavior considered here has been described as a fault, it could arise, as mentioned in the footnote on page 12, in more innocent circumstances where two cold start messages follow each other very quickly and the sender of the second is already committed to the act when the first is received. One way to eliminate this "innocent" scenario is for the hub to block messages that arrive too close on the heels of their predecessor. The nodes should then be designed to a specification requiring that they respond correctly to messages that are spaced at least this far apart. Enforcing a minimum message separation will also

[5] This is not true for $n > 3$; please see the update on page 23.

eliminate the dual "innocent" scenario in which a sending node is unable to respond to a message that arrives too soon *after* its own transmission.

Of course, the kind of behavior considered here could be the result of a genuine "deafness" fault, in which case a hub-enforced minimum message spacing will not cure the problem. A second form of "deafness" also constitutes a definite fault. This form, also considered by Paulitsch and Steiner, arises when a node in the `start` state fails to hear an acknowledging message. This can be modeled by adding a guarded command similar to Case 8, but which ignores the incoming `busmsg`. As before, we use a `BOOLEAN` state variable `deafness` to control activation of this fault.

```
[] % Case 8a: deafness
  state = start AND counter < ctimeout AND deafness -->
     faulty' = TRUE;
     counter' = counter+1;
     slot' = incslot(slot);
     msg' = quiet
```

We also add a flag `faulty` to keep track of which nodes exhibit this faulty behavior, since we cannot expect such nodes to correctly synchronize: we are interested in whether they prevent *nonfaulty* nodes from synchronizing, so add two new properties as follows.

```
fok: LEMMA system |- F( G(FORALL (i:index):
       NOT faults[i]
        => lstates[i] = active))

fsync: LEMMA system |- G(FORALL (i, j: index):
       lstates[i] = active AND lstates[j] = active
          AND NOT faults[i] AND NOT faults[j]
        => intimes[i] = intimes[j]);
```

Here `faults` is the array in the hub to which the `faulty` flags in the nodes are wired.

As Paulitsch and Steiner observe, this kind of faulty behavior is disruptive, since the afflicted node repeatedly sends cold start messages that can collide with good messages, thereby delaying synchronization. Model checking reveals a more dangerous scenario, however. Although the nonfaulty nodes do eventually synchronize (i.e., reach the `active` state, as specified in the `fok` property), they can disagree on where they are in the schedule (i.e., fail the `fsync` property). The scenarios that manifest this require application of the Case B rule in the hub (i.e., arbitrated rather than noisy collisions). An example is where nodes 0 and 2 send cold start messages; the hub passes the one from node 0; node 1 acknowledges this and goes to its `active` state while node 2 takes this as a rejection and reverts to the `listen` state; node 0 is deaf and does not hear the acknowledgment and so it times out and sends a second cold start message at the same time that node 1 sends its regularly scheduled message; node 0 wins the arbitration and node 2 synchronizes to this and enters its `active` state thinking it is at position 0 in the schedule, while node 1 thinks it is at position 1.

I do not think it is acceptable to eliminate this failure by postulating that the hub always generates noise messages when collisions occur because, as mentioned earlier, messages that follow one another closely but do not overlap can (and should) cause the hub to block the second message, thereby creating the scenario above without requiring arbitration of actual collisions. A more robust method for masking this kind of fault is to require the hub to recognize when one cold start message has been successfully acknowledged and to block later attempts to send cold start messages. Formal modeling and analysis of this approach is not undertaken here because it depends on a more detailed hub design and is postponed until this is available.

A final "deafness" fault is one that afflicts nodes in the `listen` state; if combined with miscalculation of the timeout, this could result in "babbling" nodes that repeatedly send cold start messages that collide with good messages, thereby preventing startup. A solution to this is for the hub to disallow repeated cold start messages from any single node: a node may send additional cold start messages only if some other (or m other for some $m > 0$) nodes have sent messages in the meantime. Care is needed here, because it is perfectly legitimate for a node to send a second cold start message following a collision. Again, formal modeling and analysis of these choices is postponed until more detailed hub designs are available.

Synchronization requires that all nodes know where they are in the TDMA schedule, and the cold start message provides this knowledge by conveying the information "*now* we are at slot i"; plainly, a faulty node can upset things by sending the wrong value for i (which is intended to be its own identity). Paulitsch and Steiner call this a "masquerading" fault and discuss several scenarios that it can provoke. Some of these are fairly benign (merely postponing startup for a round or so), but they also identify one that leads to endless collisions. The proposed solution to this is for the hub to block messages that carry the wrong sender identity; this is possible because the hub knows the identity of the node connected to each of its interfaces; as soon as the hub detects that a cold start message is indicating the wrong sender, it stops copying the message to the other nodes, thereby rendering it as noise.

We can model this fault by adding the following guarded command (controlled by the variable `badtime`): it causes the node `i` to send the incorrect `slot j`.

```
[] % Case 6b: masquerading
  ([] (j:index): badtime AND j /= i AND
   state = listen AND busmsg = quiet AND counter = ltimeout -->
     faulty' = TRUE;
     state' = start;
     counter' = 0;
     msg' = normal;
     slot' = j)
```

The hub is modified to mask these faults by changing its Case B to check that only messages carrying correct times are treated normally, and adding a Case B1 that substitutes noise for messages carrying incorrect times.

```
[] % Case B
% strengthened to check correct time sent
([] (i: index): inmsgs'[i] = normal AND intimes'[i]=i -->
  outmsg' = normal;
  outtime' = intimes'[i])
[] % Case B1: send  noise if bad time sent
([] (i: index): inmsgs'[i] = normal AND intimes'[i]/=i -->
  outmsg' = noise;
  outtime' = 0)
```

Model checking shows that the modified hub is able to mask masquerading faults (i.e., the properties `fok`, `fsync`, and `fast` are verified), provided at least two nonfaulty nodes are present.

Chapter 3

Discussion

We have constructed a formal specification of the TTA startup algorithm in the SAL language and examined it by model checking with SALenv. The specification uses a discrete model of time and uses synchronous composition to connect the nodes with the hub. Since the goal of the startup algorithm is to achieve synchronization, and prior to synchronization the nodes are skewed from one another by arbitrary real-time intervals, it may seem that our modeling approach trivializes the problem by assuming synchrony and eliminating the continuous nature of time. However, I hope that the arguments presented as the model was constructed allay these doubts: the model can be seen as a hub-centric view in which activity over arbitrary real-time intervals is lumped together into the behavior of a discrete "slot." The subtle scenarios revealed by model checking also provide confidence in the fidelity and utility of the modeling approach used. Nonetheless, it would be interesting to repeat the exercises undertaken here with a model checker for timed automata and to determine empirically any advantages and disadvantages of this alternative modeling approach for this type of problem.

Model checking the algorithm confirmed the claims of its authors: in the fault-free case, the nodes correctly synchronize after at most one collision. However, model checking allowed us to simplify one of the parameters to the algorithm: the listen timeout can be reduced from $2n + i$ to simply $n + 1$. Model checking also revealed that a design in which the hub arbitrates among simultaneous messages, rather than registering the collision as noise, has attractive properties and allows both timeouts to be simplified to n.

We extended our SAL model to include some of the faults considered by Paulitsch and Steiner. We showed that "deaf" nodes can delay the synchronization of nonfaulty nodes. We noted that a certain kind of behavior attributed to "deaf" nodes could actually arise with (what might reasonably be considered) nonfaulty nodes when messages arrive at the hub in very close succession (so that a node might be unable to respond appropriately to a message received immediately before or after its own cold start message). We suggested that the hub should enforce a minimum spacing on messages in the startup phase (by discarding those that arrive too close on the heels of another) to avoid this problem. With truly faulty

"deaf" nodes, we exhibited a scenario more dangerous than those exhibited by Paulitsch and Steiner in which nonfaulty nodes enter their active states while differing in their assessment of the current position within the TDMA round.

To overcome these and certain other faults, the hub must be augmented to perform additional error detection and masking. We did not model these additional hub mechanisms because we lack information on their designs. For the same reason, we did not model faults and masking mechanisms related to nodes that perform actions at inappropriate times. We did, however, model "masquerading" faults and showed that a hub that squashes cold start messages carrying incorrect identification of the sender is adequate to mask these faults.

In future work we hope to undertake (or hope that collaborators will undertake) formal modeling and analysis of the neglected fault classes, and of more detailed hub designs. There are actually two designs for a TTA hub: that being developed in Next TTA and implemented by TTTech, and that being designed and implemented by Honeywell as part of its TTA-based MAC (Modular Aerospace Controls) architecture. The Honeywell hub is a "minimalist" design that focuses on ensuring consistent message reception among its attached nodes, while the Next TTA hub performs more elaborate diagnostic functions. The Honeywell hub is to be sufficiently simple that its fault coverage can be assured by exhaustive testing.

A hub not only must detect and mask certain faults during startup, as the nodes of the cluster are synchronizing, but it must also synchronize *itself* to the cluster (so it can perform its "guardian" function that stops faulty nodes broadcasting outside their assigned slots). Future work must consider the synchronization algorithm operating in the hub itself.

A full TTA cluster has two hubs. Since a message sent from one node can arrive at the two hubs at slightly different points in real time, it is not feasible to run the startup algorithm with both hubs operating in parallel (the same messages can collide or be arbitrated differently at the two hubs). The algorithms for starting both hubs of the cluster need to be formalized and examined. Restart algorithms (for both nodes and hubs) also need examination.

We have argued that the discrete modeling employed here is adequate and sound. However, when fault-tolerant systems are concerned, verification by model checking suffers from the disadvantage that it is necessary explicitly to model each faulty behavior, and it is possible to overlook some of them. Formal verification by theorem proving is attractive in this context because the assumptions on faulty components can be stated axiomatically (and may be empty): we do not have to enumerate all possible faults. Thus, once the algorithms of a hub design have been examined and explored by model checking in the presence of a wide class of fault models, we recommend that full verification is undertaken by theorem proving.

3.1 Update

The experiments described in this report were performed in early 2003 using the explicit-state model checker of what is now known as SAL-1. Explicit state model checkers have their virtues: they are very effective when the system specification is fairly complicated but the number of reachable states is small (less than a few million), or when there is a bug that can be found at a fairly shallow depth (assuming breadth-first exploration, which is the default in SAL). However, the SAL-1 model checker is not highly optimized and the experiments described in this report rapidly became tedious or infeasible as n, the number of nodes, was increased beyond a modest number (the number of reachable states is approximately exponential in n). Consequently, most of the experiments were performed with $n = 3$.

Later in 2003, the symbolic and bounded model checkers of SAL-2 became available. These are highly optimized and their underlying methods are effective with large state spaces (though they can be defeated by huge state spaces, or by very complicated system descriptions). The SAL symbolic model checker is highly effective and scales well on the specifications used in these experiments. For example, on a modest laptop it requires 11 seconds to check the `sync` property for the fault-free version of the specification with 3 nodes, 14 seconds for 4 nodes, and 19 seconds for 5. In contrast, the explicit-state model checker on the same laptop requires only 15 seconds for 3 nodes, but 715 seconds (nearly 12 minutes) for 4 nodes (and the time for 5 nodes exceeded the experimenter's patience).

Using the SAL symbolic model checker, we repeated some of the experiments described in Section 2.2 but using larger numbers of nodes. These larger models refuted our claim that the value of the listen timeout could be reduced to `n+1`: the `sync` property fails using this timeout when $n > 3$. The timeout value $2n + i$ recommended by Paulitsch and Steiner produces no violations in these larger models.

We also reexamined some of the behaviors with faulty components described in Section 2.3; in particular, we studied the case when `simpledeafness` is `TRUE` in a system with 4 nodes. Unlike the 3-node case considered earlier, this configuration does *not* satisfy the `sync` or `fsync` properties. The counterexample found by the SAL symbolic model checker is similar to that described on page 18, which was caused by a node that fails to hear an acknowledgment to its cold start message; here, the counterexample is due to a receiver that fails to hear a cold start message and instead sends one of its own. In narrative form, this counterexample proceeds as follows.

1. Nodes 0 and 1 send cold start messages simultaneously; the hub transmits only the message from Node 0. Both Nodes 0 and 1 are waiting for confirmation of their messages. Node 3 receives the cold start message and enters the `active` state.

2. Next, Node 2, which is exhibiting `simpledeafness` and failed to receive the cold start message from Node 0 sends a cold start message of its own. Node 1 takes this as confirmation of its own cold start and enters the `active` state.

3. Nonfaulty Nodes 1 and 3 are now both in the `active` state, but the clock of Node 1 is set to the assumption that it was the initiator while that of Node 3 is set to the assumption that Node 0 was the initiator, and so the `fsynch` property is violated.

This failure is not easily countered by mechanisms in the nodes alone: we really need the hub to engage in synchronization as well, so that it can block messages (such as that from Node 2 in the scenario above) that can disrupt an ongoing startup effort.

The conclusions drawn from these examinations of larger systems than were considered in the initial study are first, that it is very dangerous to base conclusion on evidence from small models and, second, that the performance of the SAL symbolic model checker is such that it does allow efficient examination of relatively large models.

The second of these points is particularly salient because the failure scenarios discovered by model checking "deaf" nodes indicate that a hub design that is capable of masking these and other plausible node faults will be rather complex, and its verification will prove correspondingly challenging.

During the first half of 2003, researchers at the Technical University of Vienna developed suitable node and hub algorithms for TTA startup in the presence of faults, and in the summer of 2003 they joined forces with researchers at SRI and the University of Ulm to examine these algorithms using model checking. The examination built directly on the techniques developed and reported here, and was successful in using the SAL symbolic and bounded model checkers to explore the behavior of a two-hub system with up to 6 nodes operating in the presence of an "exhaustive" collection of modeled faults. A paper describing this activity has been selected for presentation at a major conference [SRSP04].

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994. 5

[Arc00] Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):139–181, 2000. 5

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer-Verlag, 1998. 5

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`. 6

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for consensus. *Journal of the ACM*, 34(1):77–97, January 1987. 3

[dMRS] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. Submitted for publication. 5

[FLP85] 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. 3

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985. 1

[Lön99a] Henrik Lönn. Initial synchronization of TDMA communication in distributed real-time systems. In *The 19th International Conference on Distributed Computing Systems (ICDCS '99)*, pages 370–379, Austin, TX, May 1999. IEEE Computer Society. 5

[Lön99b] Henrik Lönn. *Synchronization and Communication Results in Safety-Critical Real-Time Systems*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1999. Available at http://www.ce.chalmers.se/staff/hlonn/papers/thesis_hlonn.pdf. 5

[LP97] Henrik Lönn and Paul Pettersson. Formal verification of a TDMA protocol start-up mechanism. In *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*, pages 235–242, Taipei, Taiwan, December 1997. IEEE Computer Society. 5

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. 5

[Min93] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349, NASA Langley Research Center, Hampton, VA, November 1993. 1

[ORSvH95] 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. 5

[PS02] Michael Paulitsch and Wilfried Steiner. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. In *The 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 329–336, Vienna, Austria, July 2002. IEEE Computer Society. 2, 3, 5

[PSvH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Charles B. Weinstock and John Rushby, editors, *Dependable Computing for Critical Applications—7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, pages 207–226, San Jose, CA, January 1999. IEEE Computer Society. 1

[Sch87] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987. 1

[Sha92] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236, Nijmegen, The Netherlands, January 1992. Springer-Verlag. 1

[SRSP04] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. To be presented at DSN '04, July 2004. Available at http://www.csl.sri.com/~rushby/abstracts/startup-verification. 24

[WL88] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988. 1