# Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation[*]

Wilfried Steiner
Real-Time Systems Group,
Technische Universität Wien, Austria
steiner@vmars.tuwien.ac.at

John Rushby
Computer Science Laboratory,
SRI International, USA
rushby@csl.sri.com

Maria Sorea, Holger Pfeifer
Abteilung Künstliche Intelligenz,
Universiät Ulm, Germany
sorea|pfeifer@informatik.uni-ulm.de

**Abstract**

The increasing performance of modern model-checking tools offers high potential for the computer-aided design of fault-tolerant algorithms. Instead of relying on human imagination to generate taxing failure scenarios to probe a fault-tolerant algorithm during development, we define the fault behavior of a faulty process at its interfaces to the remaining system and use model checking automatically to examine all possible failure scenarios. We call this approach "exhaustive fault simulation". In this paper we illustrate exhaustive fault simulation using a new startup algorithm for the Time-Triggered Architecture (TTA) and show that this approach is fast enough to be deployed in the design loop. We use the SAL toolset from SRI for our experiments and describe an approach to modeling and analyzing fault-tolerant algorithms that exploits the capabilties of tools such as this.

## 1 Introduction

Design of fault-tolerant distributed real-time algorithms is notoriously difficult and error-prone: the combinations of fault arrivals, interleaving of concurrent events, and variations in real-time durations lead to a *case explosion* that taxes the intellectual capacity of human designers. These difficulties are compounded when optimizing numerical parameters—e.g., seeking to determine a minimum safe timeout, or the least time required to stabilize after an upset.

In an idealized world, algorithms are derived by a systematic process guided by formal correctness arguments but, in contemporary reality, designers generally have an informal argument in mind and develop the final algorithm and its parameters by mentally exploring local variations against that argument and against scenarios that highlight tricky cases. Exploration against scenarios can be partially automated using a

[*]This work was supported by the European project NEXT TTA (project IST-2001-32111) and by NASA Contract NAS1-00079 and Cooperative Agreement NCC-1-377 with Honeywell Incorporated.

simulator or rapid prototype and such automation may increase the number of scenarios that can be examined and the reliability of the examination.

Automated examination of scenarios can be taken still further using model checking. In model checking, the case explosion problem is transformed into one of *state explosion*—meaning that the time and space required to run the model checker grows rapidly and eventually becomes infeasible as the size of the model grows, so that abstraction, or consideration of only limited numbers of fault cases and real-time delays, must be employed.

When using model checking in the design loop, the challenge is to cover a usefully large number of scenarios in a very short time (say a few minutes), so that the designers can perform an interactive exploration of the design space without losing concentration or patience. As a design becomes consolidated, attention shifts from exploration to verification and the challenge for model checking becomes one of covering a truly exhaustive set of scenarios for a realistically accurate model in reasonable time (say, overnight).

Whether model checking is performed for exploration or verification, a matter of concern is the ease or otherwise of encoding the algorithm, its fault model, and assumed environment in the language of the model checker: most of these languages were originally developed for specifying hardware circuits or programs and are less than ideal for describing fault tolerant algorithms and their fault models at appropriate levels of abstraction.

In this paper, we describe an approach that provides a "dial" so that a single model can be used in model checking for both rapid exploration and exhaustive verification, and we illustrate how the model can be used both to check correctness and to help estimate worst-case performance parameters. We also demonstrate how the latest generation of model-checking tools (we use SAL from SRI) meets the challenges of providing both a convenient modeling language and the performance to examine trillions of states in a few hours. We illustrate our approach using a new startup algorithm for the Time-Triggered Architecture (TTA).

This paper is structured as follows. In Section 2, we present an overview of the Time-Triggered Architecture and discuss its startup in an informal manner. In Section 3, we discuss generic modeling issues and introduce a formal model of the startup algorithm in the SAL language. The desired correctness properties are specified in Section 4 and Section 5 presents and discusses the results of our model-checking experiments. Finally, we conclude in Section 6.

## 2 The Time-Triggered Architecture

The Time-Triggered Architecture (TTA) supplies a foundation for fault-tolerant safety-critical applications, such as control functions in cars and aircraft. It provides an ultra-reliable logical bus connecting the "host" computers that implement the chosen application, and a set of services that make it relatively simple to organize the application in a fault-tolerant manner. Each host computer attaches to the system through a TTA *controller*; the combination of a host and its controller is called a *node*. Nodes communicate over replicated shared media, called *channels*. While initially the channels were physical buses, current realizations favor a "star" topology with a *central guardian* at the hub of each star. The central guardians provide more robust defense against error propagation than the previous approach that located simpler guardians at each node [7].

The node controllers and the central guardians collectively implement the Time-Triggered Protocol TTP/C that guarantees correct operation of the system despite faults in some of the hosts, controllers, or central guardians.

## 2.1 Synchronization and Startup

A TTA system or "cluster" with $4$ nodes is depicted on the left of Figure 1. During steady-state operation, the nodes execute a time-division multiple-access (TMDA) strategy to access the medium; that is, the access pattern (called the TDMA schedule) is defined a priori, as depicted on the right of Figure 1.

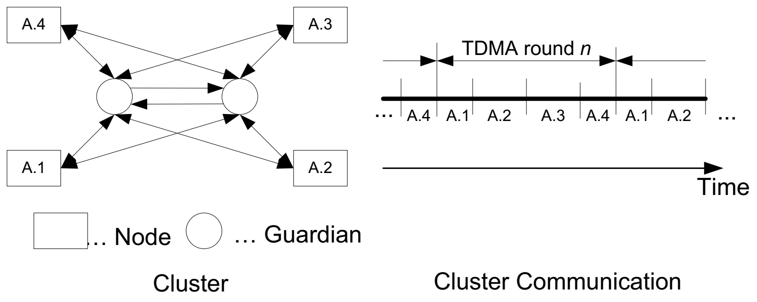


Figure 1: TTA cluster and TDMA schedule

Each TTA node has a mapping from its local time to the *slots* in the cyclic TDMA schedule; in particular, each node $i$ knows the local time at which the slot for node $k$ begins—we can denote this time by $s_i(k)$. Operation of TTA depends on synchronization of the local clocks, which simply means that for any two nonfaulty nodes $i$ and $j$, the instants when $i$'s local clock reads $s_i(k)$ and when $j$'s clock reads $s_j(k)$ must occur very close together in real time.

The *synchronization problem* is to adjust the values of $s_i$ (or, equivalently, the local clocks) so that nodes remain synchronized despite the drift of their hardware clocks (due to their oscillators operating at slightly different rates). The synchronization problem is well understood and many algorithms to solve it have been developed, analyzed, and formally verified, including the algorithm employed in TTA [9].

The *startup problem* is to establish values for the functions $s_i$ (or, equivalently, for the local clocks) as the nodes first power up so that they quickly become synchronized; the *restart problem* is to reestablish synchronization after transient faults have afflicted the values of $s_i$ or the local clocks at one or more (or all) nodes. In this paper, we are concerned with algorithms for the startup and restart problems.

## 2.2 Fault Hypothesis

Since TTA systems are designed for safety-critical applications, a sufficient degree of fault tolerance must be provided. The fault hypothesis (i.e., the number, arrival rate, and kind of faults to be tolerated) of the basic bus-based TTP/C protocol is discussed in [2]. Fault injection studies [1] showed that additional mechanisms, such as the central guardians of the star topology [3], are necessary to achieve the demanding requirements for fault tolerance in the aerospace and automotive industries.

With respect to the protocol execution, each central guardian has full knowledge of the parameters of its attached nodes, and can therefore judge whether a message (which is sometimes called a “frame” in TTA) sent by a node is valid or not (i.e., is sent within its assigned slot and satisfies certain consistency checks). Guardians relay valid messages to all the other nodes on their channel, so that from the nodes’ point of view, the channel looks like a broadcast bus. A basic TTA system uses two channels, whose central guardians are connected by a pair of *interlinks* that allow each guardian to receive data broadcast on the other channel. The interlinks are needed in the algorithm developed here to avoid scenarios in which one clique of nodes is synchronized to one guardian and another set to the other, with each clique unaware of the existence of the other. Each interlink is unidirectional: that is, the central guardian of channel $X$ receives data from channel $\bar{X}$ on one interlink but cannot transmit on this interlink, and vice versa.

The implementation of a central guardian makes it physically impossible for it to create a correct frame by itself (it lacks the hardware to construct the CRC that is part of a valid frame), or to store a previously sent frame and delay it for an arbitrarily long duration. Thus, if a central guardian receives a correct frame over the interlink connection from the other channel it can be assured that the frame was sent by a correct sender and the data can be used for further protocol execution.

Using central guardians, TTA is claimed to tolerate one faulty component (the “single failure hypothesis”); this can be either a faulty node, which is allowed to send arbitrary signals for arbitrary durations with arbitrary frequency, or a faulty channel (including its guardian), which is allowed to show the same behavior as a faulty node with the restriction that it cannot create correct frames, nor delay frames for an arbitrary duration.

## 2.3 Fault-Tolerant Startup

A basic solution to the startup problem is for nodes that see no traffic for some time to send a “wakeup” message that carries their own identity. This message provides a common event that all nodes can use as a baseline for their local clocks, and the identity of the sender indicates the position in the TDMA schedule to which this time corresponds.

Of course, two nodes may decide to send wakeup messages at approximately the same time, and these messages will “collide” on the channel. In a bus-based TTA, the signals from colliding messages physically overlay on the medium, but propagation delays cause different nodes to see the signals at different times so that collision detection can be unreliable. In a star topology, the central guardians arbitrate collisions and select just one message from any that arrive at approximately the same time to forward to the other nodes. However, each central guardian arbitrates independently, so nodes can receive different messages on the two channels at approximately the same time; resolving these “logical collisions” is a responsibility of the startup algorithm.

In addition to collisions, the startup algorithm must deal with faulty nodes that may send “wakeup” messages at inappropriate times, masquerade as other nodes, and generally fail to follow the algorithm. Many of these faults can be detected and masked only with sophisticated guardians; the central guardians of the star topology are a cost-effective way to provide this protection. However, this additional fault tolerance exacts a price: the central guardians must synchronize with the nodes during startup. Because the communication system is replicated and there are two central guardians, it is particularly crucial that a faulty node must not be able to initiate or infiltrate a startup sequence to cause the central guardians to start at different positions in the TDMA schedule. And, of course, one of the guardians could itself be faulty.

Fault-tolerant startup of a TTA system clearly requires rather intricate algorithms in the nodes and guardians. A suitable "node only" startup algorithm for the bus topology is implemented in TTP/C [12]. A startup algorithm for central guardians was designed as part of the star topology developed in the NEXT TTA project. Model checking assisted in the design loop of this algorithm and led to a more resource-efficient solution: whereas the initial guardian startup algorithm required 1 timer per node, the final version uses only a single timer. Model checking also was used in assurance of the overall algorithm and confirmed the need to modify the algorithm used in the nodes to overcome certain partitioning scenarios (see Section 5.2). The finished algorithms are outlined below and the model-checking activity that assisted in their development and assurance is the focus of the rest of the paper.

### 2.3.1 Node Startup

The state-machine of the startup algorithm executed in the nodes is depicted in Figure 2(a). It consists of $4$ states: **INIT, LISTEN, COLDSTART,** and **ACTIVE**. Each node $i$ has two unique timeout parameters, $\tau_i^{listen}$ and $\tau_i^{coldstart}$ that are defined in the following recursive way (based on the unique value $\tau_i^{startup}$).

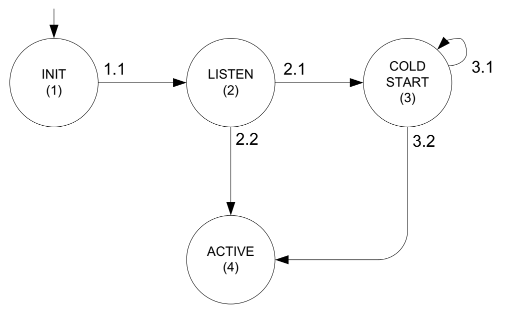
**Startup Delay:** $\tau_i^{startup}$ is unique to each node. It is given by the duration of all TDMA slots from the beginning of the TDMA round up to the beginning of the slot for node $i$ (whose duration is $\tau_i^{slot}$):

$$\tau_i^{startup} = \begin{cases} 0 & i = 0 \\ \sum_{j=1}^{i} \tau_{j-1}^{slot} & i > 0 \end{cases}$$

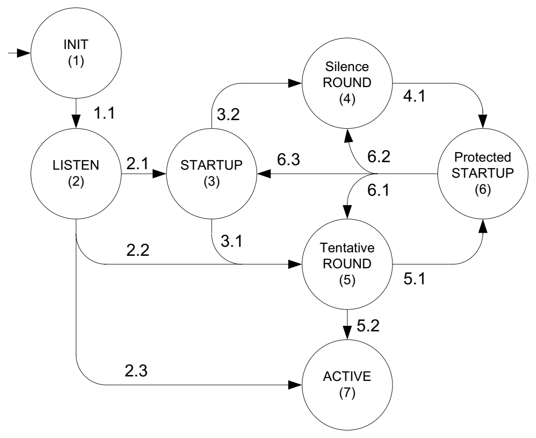
**Listen Timeout:** $\tau_i^{listen}$ is given by the sum of the node's startup delay $\tau_i^{startup}$ and 2 TDMA rounds (each of duration $\tau^{round}$): $\tau_i^{listen} = 2 * \tau^{round} + \tau_i^{startup}$.

**Cold-Start Timeout:** $\tau_i^{coldstart}$ is given by the sum of the node's startup delay $\tau_i^{startup}$ and 1 TDMA round: $\tau_i^{coldstart} = \tau^{round} + \tau_i^{startup}$.

When a node is powered-on it either has to integrate to an already synchronous set, or it must initiate or wait for a cold-start to be executed. Each newly started (or restarted) node $i$, after performing some internal initialization in the **INIT** state, transits to **LISTEN** (Transition 1.1) and listens for the unique duration $\tau_i^{listen}$ to determine whether there is a synchronous set of nodes communicating on the medium. During synchronous operation *i-frames* (this is the name of a kind of message, it has nothing to do with node $i$) are transmitted periodically that carry the current protocol state, including position in the TDMA round. If the node receives such an i-frame, it adjusts its state to the frame contents and is thus synchronized to the synchronous set (Transition 2.2); if not, the cold-start mechanism is executed. Cold-start is done in two phases. During the first phase (while in the **LISTEN** state), each node listens for a "cold-start" message (*cs-frame*) from another node indicating the beginning of the cold-start sequence; cs-frames are similar to



(a) Node Startup



(b) Guardian Startup

Figure 2: State-machine of the TTA startup algorithm

i-frames but carry a protocol state suggested by the sending node. When a node completes reception of a cs-frame, it enters the second phase **COLDSTART** (Transition 2.1) and resets its local clock to $\delta_{cs}$ (that is the transmission duration of the cs-frame). Thus, all nodes that received the cs-frame have synchronized local clocks (within system tolerances, including propagation delay). Each node that receives neither an i-frame nor a cs-frame during the **LISTEN** phase will enter **COLDSTART** (Transition 2.1), resets its local clock to 0 and sends out a cs-frame by itself. Thus, after the transmission of the cs-frame ($\delta_{cs}$ later), the local clock of the sending node will also be synchronized to the local clocks of the set of receiving nodes. This algorithmic choice, not to directly synchronize the receiving nodes on the contents of the first cs-frame while in the **LISTEN** state, is called the *big-bang* mechanism. There is, of course, the possibility that two nodes $p$ and $q$ send out simultaneous or overlapping cs-frames. The receiving nodes will see this as a logical collision but take the same actions as if a single cs-frame was received. Each node $p$ in **COLDSTART** state waits for reception of another cs-frame or i-frame until its local clock reaches the value of its individual cold-start timeout $\tau_p^{coldstart}$. If it receives such a frame it synchronizes on its contents and enters the **ACTIVE** state (Transition 3.2); if not, it resets its local clock and again broadcasts a cs-frame (Transition 3.1). No further collision can occur at this point, for the following reasons.

1. Based on the strict order of the unique cold-start timeouts $\tau_i^{coldstart}$ no two nodes that caused a collision can collide again.

2. Since $\tau_i^{listen} > \tau_j^{coldstart}$, for every two nodes $i, j$, no newly awoken node $i$ may cause a collision.

The big-bang mechanism ensures better precision, since the synchronization quality in the second phase is independent of the propagation delay: a receiving node knows the identity of the *unique* sender of the cs-frame and can compensate for its known propagation delay. More importantly, the big-bang mechanism is necessary to mask certain faults—see Section 5.2.

### 2.3.2 Guardian Startup

A faulty node could masquerade as another during startup, send cs-frames at inappropriate times (or continuously), and generally fail to follow the algorithm. A central guardian can mask these faults, but to do so (and to perform its prime function of enforcing the TDMA schedule during steady-state operation), it must itself synchronize with its nodes. The startup algorithm of the guardians is depicted in the state-machine in Figure 2(b).

A central guardian starts in **INIT** state where all communication on its channel is blocked. When its initialization is finished it transits to **LISTEN** state (1.1) and listens to the interlink for $2 * \tau^{round}$, that is, it tries to integrate to an already running system. If an i-frame is received, the central guardian transits to **ACTIVE** state (2.3); if a cs-frame is received, it transits to **Tentative ROUND** state (2.2). If an integration was not possible during **LISTEN**, the central guardian transits to **STARTUP** state (2.1). All ports are now opened and the central guardian waits until it receives a valid frame either on one of its ports or on the interlink. If more than one port becomes active at the same time, the central guardian selects one port non-deterministically. If a cs-frame is received and no logical collision occurred (that is the guardian received either two identical cs-frames or only one frame), the central guardian transits to **Tentative ROUND** state (3.1). If a collision occurred the central guardian transits to **SILENCE** state (3.2). In **Tentative ROUND** state the central guardian operates the remaining TDMA round (the received frame during **STARTUP** state is considered the first frame of a TDMA round); if during this round a valid i-frame is received, the startup initiated by the cs-frame sender is confirmed and the central guardian proceeds to **ACTIVE** state (5.2). If during this TDMA round no valid i-frame was received the central guardian transits to **Protected STARTUP** (5.1). If a central guardian transits to **SILENCE** state (because a collision was received) it blocks all communication for the remaining round and transits to **Protected STARTUP** as well (4.1). **Protected STARTUP** state differs from **STARTUP** state in that here the ports are enabled for one TDMA round according to the cold-start timeouts of the nodes. Thus, in contrast to **STARTUP** state every node is forced to stay to its timeout pattern. The transitions from **Protected STARTUP** state to **Tentative ROUND** state (6.1) and **SILENCE** state (6.2) underly the same rules as in **STARTUP** state. If no transition is done for a period of one TDMA round the central guardian transits back to **STARTUP** state (6.3) and the startup sequence is repeated. Since the central guardian has full knowledge of the attached nodes, it can detect faulty transmissions. If a central guardian detects a faulty node it will block all further attempts of this node to access the communication channel during the startup sequence. Thus, a faulty node cannot influence the startup sequence forever.

## 3 Verification Model

The startup algorithm described in the previous section is fairly subtle and must cope with many kinds of fault and timing behaviors. Model checking provides a way to explore these behaviors in an automatic way, but faces certain difficulties. First, the algorithm involves time in an essential way and the most realistic formal model for the algorithm will be one in which time is treated as a continuous variable. Timed automata provide a suitable formalism of this kind, and are mechanized in model checkers such as Kronos and UPPAAL . L ̈onn [8] considers startup algorithms for TDMA systems similar to TTA and verifies one of them using UPPAAL. However, model checking for timed automata is computationally complex, so that when we add the case/state explosion caused by considering a large number of fault scenarios, the model rapidly becomes computationally infeasible. Our initial experiments did use timed automata and we were unable to consider more than a very few simple kinds of faults.

It is essential to the utility of model checking for exploration and verification of fault-tolerant algorithms that we are able to consider a large number of different kinds of faults—ideally, we would like the fault model to be *exhaustive*, meaning that we describe every kind of fault within the fault hypothesis, and let the model checker inject these in all possible ways. Since this is impracticable in a model that uses continuous time, we looked for an abstraction employing discrete time.

Nodes executing the startup algorithm measure time by counting off slots in the TDMA schedule. Although slots have duration and may be offset at different nodes, we can think of them as indivisible units: 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). Thus, we can use a discrete notion of time and can model the collective behavior of a cluster of nodes as the synchronous composition of discrete systems. Another way to justify this modeling approach is to think of it as describing the system from the point of view of a central guardian: each discrete instant corresponds to some real time interval at the guardian 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 central guardian.

We explored this approach in an analysis of the original startup algorithm from [12] and found it to work very well. We used the SAL (Symbolic Analysis Laboratory) language and tools from SRI (see `sal.csl.sri.com`); SAL complements the widely-used PVS verification system by providing more automated forms of analysis for systems that can be specified as transition relations (see [5] for a description of SAL and a discussion of its relation to other SRI tools). Unlike most other model-checking languages, SAL supports a relatively abstract specification language that includes many of the high-level types and constructs found in PVS, and this allowed the algorithm and its properties to be specified in a convenient and succinct manner.

However, this experiment, which used the explicit-state model checker of SAL, exposed a second difficulty: comprehensive—let alone exhaustive—fault modeling provokes a state explosion problem even in a discrete-time model. Although only a single channel and just a few kinds of faults were considered, model checking required 30 seconds for a 4-node cluster, and over 13 minutes for a five-node cluster. A more efficient explicit-state model checker such as Spin could possibly have improved these figures, but even the largest of the models considered in these preliminary experiments has only 41,322 reachable states, whereas exhaustive fault modeling for the new algorithm with two channels could generate many *billions* of reachable states, which far exceeds the reach of any explicit-state model checker.

In the months since those initial experiments were performed, the SAL 2.0 toolset has become available, and this provides several state of the art model checkers, including symbolic (using BDDs), bounded (using a SAT solver), and infinite-bounded (using decision procedures). The SAL symbolic model checker is able to verify the 4- and 5-node examples mentioned above in 0.38 and 0.62 seconds, respectively, on the same machine used for the explicit-state experiments. These two or three orders of magnitude improvement in performance encouraged us to try using model checking during development of the new startup algorithm.

However, fully exhaustive fault models still posed a challenging prospect, so we developed a modeling "dial" that could inject varying *degrees* of faults: our idea was to use as high a degree (i.e., as many kinds) of faults as proved feasible in practice. In the remainder of this section we first present our basic model of the startup algorithm and then describe the modeling concepts for faulty components of varying degrees. Due to space limitations we only give representative parts of the model and refer the interested reader to [11] where the complete source code of the SAL model can be found, together with instructions that will help to recreate the experiments.

## 3.1 Basic Model

The system model comprises $n$ *nodes*, each synchronously composed with two central *hubs* that each contain a central guardian that blocks certain faulty messages. At each time step, each node examines the input messages received from the hubs, consults its private state variables, and possibly generates an output message that it sends to the hubs. Each hub examines the messages received from the nodes and the other hub and constructs the single message that will comprise the consistent input presented to the nodes at the next time step.

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

```
startup: CONTEXT =
BEGIN
n: NATURAL = 4;
index: TYPE = [0..n-1];
maxchannels: NATURAL = 2;
channels: TYPE = [0..maxchannels-1];
maxcount: NATURAL = 20*n;
counts: TYPE = [0..maxcount];
```

Here, `n` is the number of nodes (here assigned the value 4, but we also examine models with 3, 5, and 6 nodes), which are identified by elements of the type `index`. Analogously, `maxchannels` is the number of channels, which are identified by elements of the type `channels`. The largest timeout considered is `maxcount` and the values of a timeout counter are given by the type `counts`.

```
states: TYPE = {init, listen, start, active, faulty, faulty_lock0, faulty_lock1, faulty_lock01};
hub_states: TYPE = {hub_init,  hub_listen, hub_startup, hub_tentative, hub_silence,
                    hub_protected, hub_active};
msgs: TYPE = {quiet,noise,cs_frame,i_frame};
```

The enumerated types `states`, `hub_states`, and `msgs` specify, respectively, the states of the algorithm at a node, the states of the algorithm at a hub, and the kind of messages that can be exchanged with a hub. The states correspond to those in the state-machines of Section 2, plus additional `faulty` states that are used in the simulation of faulty components. Each node may output messages with values `quiet` (meaning no message), `noise` (meaning a syntactically invalid signal), `cs_frame` (a cs-frame), or `i_frame` (an i-frame); the hub will return a message type based on the inputs of the attached nodes.

```
LT_TO:ARRAY index OF NATURAL = [[j:index] 2*n+j];
CS_TO:ARRAY index OF NATURAL = [[j:index] n+j];
```

The unique timeouts for each node are specified as `LT_TO` (listen timeout) and `CS_TO` (cold-start timeout), as defined in Section 2.

### 3.1.1 Node Model

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

```
node[id:index]: MODULE = BEGIN INPUT
  msg_in:  ARRAY channels OF msgs,
  time_in: ARRAY channels OF index,
  lock_in: ARRAY channels OF BOOLEAN
OUTPUT
  msg_out:  ARRAY channels OF msgs,
  time_out: ARRAY channels OF index,
  state: states,
  counter: counts,
  errorflag: BOOLEAN
```

The `msg_in` represents the kind of message that the node receives from the hubs; if it is a normal message, then `time_in` indicates the slot position transmitted in the sender's frame, which equals the current time measured relative to the start of the TDMA round if the sender sends a correct value. We can think of this information as being included in the message, but it is easier to model it as a separate variable. The input variable `lock_in` is used to make the model more compact and is discussed in Section 3.2.

The output variables `msg_out`, `time_out`, `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 output variable `errorflag` is used for diagnosis of the model and has no influence on the protocol execution.

```
LOCAL
  startupdelay: counts,
  big_bang: BOOLEAN
```

Each node has a local variable `startupdelay` that indicates the maximum duration a node is allowed to stay in `init` state (simulating the different power-on times of the different nodes). The local variable `big_bang` is set to `TRUE` if no big-bang was received yet, and to `FALSE` otherwise.

The algorithm is specified by a series of guarded commands. We describe in detail those that apply to a node in the `init` state, and one transition of a node in `listen` state, as representative illustrations.

```
[ % Transition: 1.1
     state = init
 --> state' = IF NOT faulty_node[id] THEN listen ELSE faulty ENDIF;
     counter' = 1;
     msg_out' = msg_out;
     time_out' = time_out;
```

```
[] % Let time advance
    state = init AND counter < startupdelay
--> state' = state;
    counter' = counter+1;
    msg_out' = msg_out;
    time_out' = time_out;
```

Here, the [ character introduces a set of guarded commands, which are separated by the [] symbol; 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.

Provide the counter is less than startupdelay, 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 transit to the listen state. If the counter reaches startupdelay, the node must transit either to listen or faulty state, depending whether the node simulates a correct node or a faulty one. Hence, the two guarded commands above allow the node to "wake up" and transit to the listen state at any point during the specified period of startupdelay; on entering the listen (or faulty_node) state, its counter is reset to 1.

We next describe a class of transitions for a node from listen to (cold) start state.

```
[] % Transition 2.1
([] (k: channels):
   state = listen AND big_bang AND msg_in[k] = cs_frame
    AND (NOT (EXISTS (j:channels): j/=k AND (msg_in[j] = cs_frame OR msg_in[j] = i_frame)
               AND (time_in[k]/=time_in[j] OR msg_in[k]/=msg_in[j])))
--> state' = start; counter' = 2;
    msg_out'=[[j:channels] quiet];
    time_out'=[[j:channels] 0];
    big_bang' = FALSE;)
```

This guarded command is a short hand for a set of transitions. It represents one transition for each k, with $k = 0, 1$. The precondition is satisfied, if the node is in listen state, a big_bang was not received yet by this node, the incoming message on channel k is a cs-frame, and there does not exist a channel different from k (in a dual-channel system, there is only one other channel) where a cs-frame or i-frame is received that has another time_in value than that on channel k. The output and local variables will be set to the appropriate values. The subtly differentiated cases in the precondition were helpful in testing different algorithm designs.

### 3.1.2 Hub Model

The input/output behavior of a hub is specified as follows.

```
hub[c:channels]:MODULE = BEGIN
INPUT
  msg_in:  ARRAY index OF msgs,
  time_in: ARRAY index OF index,
  interlink_msg_in: msgs,
  interlink_time_in: index
```

A hub receives `msg_in` and `time_in` as input values from each node, and `interlink_msg_in` and `interlink_time_in` from the other channel (a hub also listens to the other channel during startup).

```
OUTPUT
  msg_out: ARRAY index OF msgs,
  time_out: ARRAY index OF index,
  interlink_msg_out: msgs,
  interlink_time_out: index,
  state: hub_states,
  collisions: [0..10],
  lock: ARRAY index OF BOOLEAN
```

A hub has the following outputs: `msg_out`, the kind of message the hub sends to the nodes; `time_out`, this variable represents the slot position when a frame is relayed to the other hub; `interlink_msg_out` and `interlink_time_out` are the kind of message and slot position a hub sends to the other channel (in a correct hub, these values will be equal to `msg_out` and `time_out`). The internal state of a hub is represented by `state`. We use the variable `collisions` to count the number of collisions during startup (this variable has no influence on the protocol execution but is used in analysis). `lock` is an array of boolean variables corresponding to “ports” (the connections from a hub to its nodes). Initially these are set to `FALSE`; if a hub discovers, by its sending behavior, that a node is faulty, it will set the corresponding boolean to `TRUE` and will disallow further transmissions from this node.

```
LOCAL
  round_counter1: [0..n+1],
  round_counter2: [0..n+1],
  round_counter_delay: [0..10*n],
  slot_position: index,
  hub_error: BOOLEAN,
  partitioning: ARRAY index OF BOOLEAN,
  send_noise: ARRAY index OF BOOLEAN
```

A hub uses two `round_counter` variables to count the slots per round, while `round_counter_delay` is used to count the initial delay in `hub_init` state (analogous to `startup-delay`). During the tentative and the synchronized states, `slot_position` is used to keep track of the current slot position. The variable `hub_error` is used to model certain errors, while `partitioning` and `send_noise` are used to simulate a faulty hub that selects only a subset to relay a message or broadcasts noise.

We discuss representative transitions of the hub next.

```
[] ([] (i: index):
   state =  hub_startup AND msg_in'[i] /= quiet AND NOT lock[i]
 -->
```

This guarded command again represents a set of transitions. The precondition is satisfied if the hub is in `hub_startup` state and some node `i` sends a message with a type other than `quiet` and the port of the respective node is not locked.

```
msg_out' = [[j:index]
  IF msg_in'[i]=cs_frame AND time_in'[i]=i THEN cs_frame ELSE noise ENDIF];
time_out' = [[j:index] time_in'[i]];
interlink_msg_out'= msg_out'[0];
interlink_time_out' = time_out'[i];
```

```
state' = IF (msg_out'[i] = cs_frame
    AND ((interlink_msg_in' = cs_frame AND interlink_time_in' = time_in'[i])
      OR (interlink_msg_in' /= cs_frame)))
  OR (msg_out'[i] /= cs_frame AND interlink_msg_in' = cs_frame)
THEN  hub_tentative
ELSIF msg_out'[i] = cs_frame AND interlink_msg_in' = cs_frame
    AND interlink_time_in' /= time_in'[i]
THEN hub_silence
ELSE hub_startup ENDIF;
```

Here we present parts of the postcondition of this transition. The hub performs semantic analysis in that it checks whether the type of `msg_in'` is a cs-frame with the correct time value. Depending on the semantic analysis it relays either the frame or noise to the nodes and the other channel. The hub's next state is calculated by comparing the incoming message on its own channel and the incoming message of the other channel, as depicted in second box of specification.

The node and hub modules are connected using a `switch` module, that simply connects input variables of the nodes to the respective output variables of the hubs and vice versa. The hubs are interconnected in the same way by an `interlink` module.

## 3.2 Failure Modeling

Faults vastly increase the statespace that must be explored in model checking, and they do so in two different ways. The first way is by introducing genuinely different behaviors; we provide a *fault degree* "dial" to parameterize this as described in the following section. The second way is to introduce "clutter" in the form of states that differ in irrelevant ways: for example, a faulty node can end up in one of many different states, but once the correct components have excluded this node from further consideration, its state has no effect on system behavior. However, a model checker distinguishes all the different states of the faulty component and this needlessly complicates its task. An essential "trick" in modeling fault-tolerant algorithms is to set the states of faulty components to fixed values once they can no longer affect the behavior of the system. We implement this by a mechanism we call *feedback*, also described in the next section.

### 3.2.1 Node Failures

The model simulates time in discrete slot granularity and a faulty node is simulated as one that can send arbitrary messages in each slot. We classify the possible outputs of such a faulty node into the six *fault degrees* depicted by the $(6 \times 6)$ matrix in Figure 3. For example, a fault degree of 2 allows a faulty node

| chB / chA | quiet | cs_frame (good) | i_frame (good) | noise | cs_frame (bad) | i_frame (bad) |
|---|---|---|---|---|---|---|
| quiet | 1 | 2 | 3 | 4 | 5 | 6 |
| cs_frame (good) | 2 | 2 | 3 | 4 | 5 | 6 |
| i_frame (good) | 3 | 3 | 3 | 4 | 5 | 6 |
| noise | 4 | 4 | 4 | 4 | 5 | 6 |
| cs_frame (bad) | 5 | 5 | 5 | 5 | 5 | 6 |
| i_frame (bad) | 6 | 6 | 6 | 6 | 6 | 6 |

Figure 3: Fault degree

to broadcast only cs-frames, with the correct semantics, on zero, one, or two channels, while fault degree $6$ allows a node to send an arbitrary combination of cs-frames and i-frames with correct or incorrect semantics, noise, or nothing on each channel.

Each of this $36$ combinations was explicitly described by guarded commands in the SAL model.

```
[] state = faulty AND degree >= 2
-->
msg_out’=[[j:channels] IF j = 0 THEN cs_frame ELSE quiet ENDIF];
time_out’ = [[j:channels] IF j = 0 THEN faulty_ID ELSE 0     ENDIF];
state’ = IF lock_in[0] AND lock_in[1] AND feedback THEN faulty_lock01
  ELSIF lock_in[0] AND feedback THEN faulty_lock0
  ELSIF lock_in[1] AND feedback THEN faulty_lock1
  ELSE state ENDIF;
```

Here, one guarded command of a faulty node with fault degree $2$ or greater is depicted: such a faulty node is allowed to broadcast a cs-frame on channel $0$ and does not send on the second channel. Furthermore, to reduce the statespace, we use “feedback”: the `lock_in[i]` input variables are set by the hub $i$ (corresponding to its `lock` output variables) if it discovers that the node is faulty (by judging on the node’s output behavior). A faulty node will then transmit only `quiet` on channel $i$, since the hub will block all messages of the faulty node anyway. To judge its effect, this feedback routine can be turned on and off by setting the `feedback` parameter to `TRUE` or `FALSE` respectively.

### 3.2.2 Hub Failures

Analogous to a faulty node, a faulty hub is simulated by assigning its output variables to arbitrary values, within its fault hypothesis (a faulty hub cannot create correct messages) in each slot.

```
[] ([] (i: index):
state=hub_faulty AND msg_in’[i] /= quiet
-->
msg_out’ = [[j:index] IF partitioning[j] THEN msg_in’[i]
     ELSE IF send_noise[j] THEN noise ELSE quiet ENDIF   ENDIF];
  time_out’ =[[j:index] time_in’[i]];
  interlink_msg_out’ = msg_in’[i];
  interlink_time_out’ = time_in’[i]; )
```

This example of a transition by a faulty hub is activated if an attached node sends a message other than `quiet` to the hub. The faulty hub then is free to select a subset of nodes to which the message is forwarded. The local variable `partitioning`, an array of boolean variables, creates such a partitioning of the nodes. By specifying no initial value for this variable, the model checker is forced to test every assignment. The faulty hub is allowed to send either `noise` or `quiet` to the other nodes, using the similarly uninitialized boolean array `send_noise`. We call this method *implicit* failure modeling (in the sense, that it is not necessary to model transitions for each subset explicitly).

## 4 Correctness Properties

In the following we describe some correctness properties of the algorithms and their formulation as "lemmas" in SAL notation. Here, `G` denotes the *always* or $\Box$ modality of linear temporal logic (LTL), and `F` denotes the *eventually* or $\Diamond$ modality.

**Lemma 1** *Safety: Whenever any two nodes are in the* ***ACTIVE*** *state, these nodes will agree on the slot time.*

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

**Lemma 2** *Liveness: All correct nodes will eventually reach the* ***ACTIVE*** *state.*

```
liveness: LEMMA system |- F((FORALL (i:index): lstates[i] = active OR faulty_node[i]));
```

**Lemma 3** *Timeliness: All correct nodes will reach the* ***ACTIVE*** *state within a bounded time.*

```
timeliness: LEMMA system |- G(startup_time <= @par_startuptime);
```

**Lemma 4** *Safety_2: Whenever a node reaches the* ***ACTIVE*** *state, a correct hub has also reached either the* ***Tentative ROUND*** *or* ***ACTIVE*** *states.*

```
safety_2: LEMMA system |- G ((EXISTS (i:index): lstates[i] = active) =>
  (hstates[1]=hub_active OR hstates[1]=hub_tentative ));
```

Within our model-checking study additional lemmas were examined to gain confidence in our model. Those lemmas can be found in the source code of the SAL model.

## 5 Experimental Results and Discussion

In this section we present results from our experiment using model checking in development of the new startup algorithm for TTA. Our experiments were performed on an Intel(R) Xeon(TM) with a CPU speed of 2.80GHz and 2GByte memory. We used the Linux distribution of SAL 2.0.

### 5.1 Effectiveness of Statespace Reduction Measures

Our decision to use a discrete model for time was critical to our ability to perform these experiments at all. Although we cannot yet prove the soundness of this abstraction, we gained confidence in it by selectively removing mechanisms from the SAL model of the algorithm and observing that the model checker always detected the expected system failures.

In exploring algorithmic variations, it was crucial for the model checker to deliver results within the human attention span of a few minutes. Our principal “dials” for trading time required against thoroughness of the exploration performed by the model checker were the number of nodes considered (typically from 3 to 6), and the fault degree. The parameter $\delta_{failure}$ selects the fault modes that a faulty node may exhibit. Figure 4 illustrates the verification times in seconds for three lemmas in a 4-node model with $\delta_{failure} = 1, 3, 5$. The results clearly show the increase in verification times with fault degree. A fault degree of 1 is suitable for quick investigation in the inner design loop, while degrees 3 and 5 invite a coffee break.

| $\delta_{failure}$ | safety | liveness | timeliness |
|---|---|---|---|
| 1 | 44.11 | 196.05 | 77.14 |
| 3 | 166.34 | 892.15 | 615.03 |
| 5 | 251.12 | 1324.54 | 921.92 |

Figure 4: Effect of Increasing Fault Degree on Model-Checking Performance

The feedback mechanism (i.e., forcing failed components to a standard state to reduce the statespace) was ineffective or counterproductive in practice for medium to large models, but for very large models it proved essential. For example, one property was successfully model checked in a 6-node model in 30,352 seconds (about 8.5 hours) with feedback on, but had not terminated after 51 hours with it off. We intend to investigate the feedback mechanism further, to better understand when and why it is effective.

### 5.2 Design Exploration: The Big-Bang Mechanism

One area where we performed expensive design exploration was to determine the necessity and effectiveness of the big-bang mechanism. A crucial requirement of the startup algorithm is that it should not establish synchronous operation of a subset of nodes on a faulty hub while the second, correct, channel is available but unsynchronized. In such a case it would be possible for the faulty hub to forward messages only to the synchronous subset but not to the other nodes and hub; other nodes that are not yet synchronized would perform the startup algorithm (since the traffic of the synchronous set is hidden by the faulty hub) and startup independently of the other, already synchronized, nodes thereby establishing a classical clique scenario [13], in which two subsets of nodes are communicating within each subset but not as one coordinated whole. The big-bang mechanism (Section 2) is used to prevent such scenarios.

Our model-checking experiments verified the necessity of the big-bang mechanism by producing the following counterexample in its absence for a cluster of 4 nodes:

1. node $n_2$ and $n_3$ start up with one slot difference;
2. after the listen timeouts expire, $n_2$ and $n_3$ send their cs-frames, resulting in a collision;

3. the correct hub forwards the winning node, say $n_2$, on its channel to all nodes and the second channel;
4. the faulty hub forward the winning node on its channel, $n_3$, only to the correct hub;
5. nodes $n_1$ and $n_4$ receive only one cs-frame (from $n_2$) and synchronize on it, reaching **ACTIVE** state;
6. the correct hub sees a collision, since the faulty hub forwards the other cs-frame to it, and thus will not synchronize to the active set of nodes.

The big-bang mechanism discards the first cs-frame a node receives, since this cs-frame could be part of a collision of two nodes. The model-checking studies showed the necessity and correctness of this mechanism.

There is a class of scenarios similar to the one above that is not directly addressed by the algorithm: this is where nodes start up on a single faulty guardian (believing the other guardian to be unavailable), and only a subset of them achieve synchronous operation. These scenarios are excluded by arranging the power-on sequence so that the guardians are running before the nodes: the algorithm is able to deal with a faulty guardian provided the other guardian is available at the start of its operation.

SAL 2.0 provides both bounded and symbolic model checkers. Bounded model checkers, which are based on propositional satisfiability (SAT) solvers, are specialized for detecting bugs: they explore models only to a specified, bounded depth and can be faster than symbolic model checkers (which effectively explore the entire statespace) when bugs are present that can be detected within the bound. Bounded model checking provides algorithm developers with another analytical "dial": they can explore to increasing depths with a bounded model checker and switch to the "unbounded" depth of a symbolic model checker only when all the "shallow" bugs have been detected and eliminated. In our big-bang experiments, the SAL bounded model checker was sometimes more efficient than the symbolic one at exposing the failing scenarios. For example, it found a violation to the `Safety_2` property in a 5-node system at depth 13 in 93 seconds (solving a SAT problem with 405,398 nodes), whereas the symbolic model checker required 127 seconds (for a model with 682 BDD variables).

## 5.3 Worst-Case Startup Scenarios

We define the worst-case startup time, $\tau^{wcsup}$, as the maximum duration between 2 or more non-faulty nodes entering the **LISTEN** or **COLDSTART** states and 1 or more non-faulty nodes reaching the **ACTIVE** state.

We explored worst-case startup times by model checking the `timeliness` property for different values of `@par_startuptime`, setting it first to some small explicit value (e.g., 12) and increasing it by small steps (e.g., 1) until counterexamples were no longer produced. By exploring different cases and different cluster sizes, we were able to develop an understanding of the worst-case scenarios.

The deduced formula for worst-case startup time $\tau^{wcsup}$ (which occurs when there is a faulty node) is given in the following equations.

$$\begin{aligned}
\tau^{wcsup} &= \tau^{listen}_{max-1} + 2 * \tau^{coldstart}_{max-1} + \tau^{slot} \\
&= 3 * \tau^{round} - 2 * \tau^{slot} + 2 * (2 * \tau^{round} - 2 * \tau^{slot}) + \tau^{slot} \\
&= 7 * \tau^{round} - 5 * \tau^{slot}.
\end{aligned}$$

## 5.4 Automated Verification and Exhaustive Fault Simulation

During exploration of the algorithm we were content to consider modest cluster sizes and fault degrees, but for verification we wanted to examine larger clusters and "exhaustive" modeling of faults. The term *exhaustive fault simulation* was chosen in analogy to fault injection and with respect to the nomenclature given in [6]. While fault injection means actually to insert faults into physical systems, fault simulation is concerned with modeling faulty behavior in a mathematically model. *Exhaustive* fault simulation means that all hypothesized fault modes are modeled and all their possible scenarios are examined. In our case, this means model checking our model of the startup algorithm with the fault degree set to 6. A desirable goal is to be able to check all properties for a reasonable-sized cluster (say 5 nodes) overnight (say 12 hours, or 43,200 seconds). In this section we give formulas to estimate the number of scenarios under test for exhaustive fault simulation and report the performance achieved.

**Different startup delays:** Given a system of $n$ nodes and 2 guardians, where each of the nodes and one of the guardians was allowed to startup at an instant during a period of $\delta_{init}$, the number of scenarios, $|\mathcal{S}_{sup}|$, based on these different startup times is given by $|\mathcal{S}_{sup}| = (\delta_{init})^{n+1}$.

**Worst-case startup scenarios with a faulty node:** Given the worst-case startup time of the system $\tau^{wcsup}$ and the fault degree of a faulty node $\delta_{failure}$, the number of scenarios for one particular startup pattern of nodes and hubs, $|\mathcal{S}_{f.n.}|$, is given by $|\mathcal{S}_{f.n.}| = ((\delta_{failure})^2)^{\tau^{wcsup}}$.

Numerical estimates for these parameters are given in Table 5.

| nodes | $\delta_{init}$ | $\|\mathcal{S}_{sup}\|$ | $\delta_{failure}$ | $\tau^{wcsup}$ | $\|\mathcal{S}_{f.n.}\|$ |
|---|---|---|---|---|---|
| 3 | 24 | $3.3 * 10^5$ | 6 | 16 | $8 * 10^{24}$ |
| 4 | 32 | $3.3 * 10^7$ | 6 | 23 | $6 * 10^{35}$ |
| 5 | 40 | $4.1 * 10^9$ | 6 | 30 | $4.9 * 10^{46}$ |

Figure 5: Number of Scenarios for Different Fault Degrees

The SAL symbolic model checker is able to count the number of reachable states in a model. For the model used in the big-bang tests, these numbers were 1,084,122,880 states for 3 nodes, 508,573,786,112 for 4, and 259,220,300,300,290 for 5; these are approximately $2^{27}$, $2^{35}$, and $2^{43}$ states, respectively, in reasonable agreement with Table 5.

Figures 6(a), 6(b), and 6(c) present the model checker performance for Lemmas 1, 2, and 3 in presence of a faulty node with fault degree $\delta_{failure} = 6$ and startup-delay $\delta_{init} = 8 * \tau^{round}$. The feedback column indicates whether the feedback optimization was turned on or off. Figure 6(d) presents the results for Lemma 4 in presence of a faulty hub with startup-delay $\delta_{init} = 8 * \tau^{round}$. Results are shown for models with $3$, $4$, and $5$ nodes. The eval column indicates if the respective lemma is satisfied.

The cpu time column gives the execution time of the corresponding model-checking run, while the BDD column gives the number of BDD variables for the model (this is equivalent to the number of state bits after eliminating those that are simple combinations of others). 300 or so state bits is usually considered the realm of "industrial" model checking, where skilled tinkering may be needed to obtain a result in reasonable time. Yet all these results were obtained with no special efforts beyond those described.

| nodes # | feedback | eval. | cpu time (sec) | BDD # |
|---|---|---|---|---|
| 3 | on | true | 62.45 | 248 |
| 4 | on | true | 259.53 | 316 |
| 5 | on | true | 920.74 | 422 |

(a) Results for Lemma `safety`

| nodes # | feedback | eval. | cpu time (sec) | BDD # |
|---|---|---|---|---|
| 3 | on | true | 228.03 | 250 |
| 4 | on | true | 1242.73 | 318 |
| 5 | on | true | 41264.08 | 424 |

(b) Results for Lemma `liveness`

| nodes # | feedback | wcsup (slots) | eval. | cpu time (sec) | BDD # |
|---|---|---|---|---|---|
| 3 | on | 16 | true | 47.81 | 268 |
| 4 | on | 23 | true | 907.61 | 336 |
| 5 | on | 30 | true | 4480.90 | 442 |

(c) Results for Lemma `timeliness`

| nodes # | eval. | cpu time (sec) | BDD # |
|---|---|---|---|
| 3 | true | 56.65 | 272 |
| 4 | true | 82.95 | 348 |
| 5 | true | 4289.77 | 462 |

(d) Results for Lemma `safety_2`

Figure 6: Performance Results for Model Checking the Lemmas

## 6 Conclusion

We have presented the verification model and results of a model-checking study for a new startup algorithm for the TTA. The startup algorithm guarantees a safe and timely system startup in the presence of any one faulty component. Our model-checking experiments showed the robustness of the algorithm in the presence of a faulty node or a faulty hub.

We described modeling concepts for abstracting the problem to discrete time, and for exhaustive fault simulation. The resulting models have billions or even trillions of reachable states, yet the symbolic model checker of SAL is able to examine these in a few tens of minutes (for billions of states) or hours (for trillions). This combination of an effective modeling approach and an efficient tool allowed us to use model checking over an exhaustive fault model in the design loop for the algorithm, and also helped us establish the worst-case startup times. Thus, this approach extends previous experiments in model-checking fault-tolerant algorithms such as [14] and [4] by vastly increasing the number of scenarios considered, while achieving performance that allows the method to be used in design exploration as well as for verification.

Ongoing design work is concerned with a shift of complexity from the guardian algorithms to the node algorithms to make the interlink connections unnecessary. In ongoing formal methods studies, we are exploring the use of the infinite-bounded model checker of SAL (which combines a SAT solver with decision procedures for theories including real arithmetic) to develop and analyze models that use continuous time [10], while still allowing rich fault models. We are also using the PVS theorem prover to formally verify the algorithm and its fault hypothesis in their most general forms: even with exhaustive fault simulation, a model checker still requires us explicitly to model each fault within the fault hypothesis. We may fail to be truly exhaustive if our models are insufficiently aggressive—whereas a theorem prover can allow us simply to state the properties assumed of each fault class, and thereby fully explore their consequences.

## References

[1] A. Ademaj, G. Bauer, H. Sivencrona, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proc. of International Conference on Dependable Systems and Networks (DSN 2003), San Francisco*, Jun. 2003.

[2] G. Bauer, H. Kopetz, and P. Puschner. Assumption coverage under different failure modes in the time-triggered architecture. In *Proc. of International Conference on Emerging Technologies and Factory Automation*, pages 333–341, Oct. 2001.

[3] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. In *Proc. of 6th International Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 37 –44, Pisa, Italy, Apr. 2003.

[4] C. Bernardeschi, A. Fantechi, and S. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability*, 12:251–275, Dec. 2002.

[5] Formal Methods Program. Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 2003. Available at `http://fm.csl.sri.com/doc/roadmap03`.

[6] J.C.Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[7] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proc. of The 6th International Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 139–146, Pisa, Italy, Apr. 2003.

[8] H. L¨onn and P. 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, Dec. 1997. IEEE Computer Society.

[9] H. Pfeifer, D. Schwier, and F. W. von Henke. Formal verification for time-triggered clock synchronization. In C. B. Weinstock and J. Rushby, editors, *Dependable Computing for Critical Applications—7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, pages 207–226, San Jose, CA, Jan. 1999. IEEE Computer Society.

[10] M. Sorea. Bounded model checking for timed automata. *Electronic Notes in Theoretical Computer Science*, 68(5), 2002. Available at `http://www.elsevier.com/locate/entcs/volume68.html`.

[11] W. Steiner. SAL model of a TTA startup algorithm. Research Report 52/2003, Technische Universit¨at Wien, Institut f¨ur Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2003.

[12] W. Steiner and M. Paulitsch. 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.

[13] W. Steiner, M. Paulitsch, and H. Kopetz. Multiple failure correction in the time-triggered architecture. *Proc. of 9th Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003f)*, Oct. 2003.

[14] T. Yokogawa, T. Tsuchiya, and T. Kikuno. Automatic verification of fault tolerance using model checking. In *Proc. of 2001 Pacific Rim International Symposium on Dependable Computing*, page 95, Seoul, Korea, Dec. 2001.