

Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata

Bruno Dutertre¹ and Maria Sorea²

¹ System Design Laboratory, SRI International, Menlo Park, CA, USA
bruno@sdl.sri.com

² Abteilung Künstliche Intelligenz, Universität Ulm, Ulm, Germany
sorea@informatik.uni-ulm.de

Abstract. We discuss the modeling and verification of real-time systems using the SAL model checker. A new modeling framework based on event calendars enables dense timed systems to be described without relying on continuously varying clocks. We present verification techniques that rely on induction and abstraction, and show how these techniques are efficiently supported by the SAL symbolic model-checking tools. The modeling and verification method is applied to the fault-tolerant real-time startup protocol used in the Timed Triggered Architecture.

1 Introduction

SAL (Symbolic Analysis Laboratory) is a framework for the specification and analysis of concurrent systems. It consists of the SAL language [1], which provides notations for specifying state machines and their properties, and the SAL system [2] that provides model checkers and other tools for analyzing properties of state machine specifications written in SAL. These tools include a bounded model checker for infinite-state systems that relies on decision procedures for a combination of linear arithmetic, uninterpreted functions, and propositional logic. This tool enables the analysis of systems that mix real-valued and discrete state variables and can then apply to real-time systems with a dense time model.

SAL is a generalist tool, intended for the modeling and verification of discrete transition systems, and not for systems with continuous dynamics. As a consequence, existing models such as timed automata, which employ continuous clocks, do not fit the SAL framework very well. A first contribution of this paper is the definition of a new class of timed transition systems that use dense time but do not require continuously varying state variables, and are then better suited to SAL. The inspiration for these models is the concept of *event calendars* that has been used for decades in computer simulation of discrete event systems. Unlike clocks, which measure delays since the occurrence of past events, a calendar stores information about future events and the time at which they are scheduled to occur. This provides a simple mechanism for modeling time progress: time always advance to the next event in the calendar, that is, to the time where the next discrete transition is enabled. This solves the main difficulty encountered when encoding timed automata via transition systems, namely — ensuring maximal time progress.

The paper shows then how the SAL infinite-state bounded-model checker — which is primarily intended for refutation and counterexample finding — can be used as a verification tool and applied to timed models. A simple technique is to use a bounded model checker to perform proof by induction. We extend this technique by applying bounded model checking (BMC) to proof by abstraction. More precisely, we use BMC for automatically proving that an abstraction is correct. This provides efficient automation to support a proof method based on disjunctive invariants proposed by Rushby [3].

The modeling approach and the verification methods have been applied to the fault-tolerant real-time startup protocol used by the Timed Triggered Architecture (TTA) [4]. We first illustrate the techniques on a simplified version of the startup algorithm, where timing and transmission delays are modeled but where faults are not considered. We then discuss the verification of a more complex version of the protocol in which both timing and node failures are modeled.

Compared to existing approaches, the framework we present lies between fully automated model checking and manual verification using interactive theorem proving. Several special-purpose model checkers (e.g., [5–8]) exist for timed automata and have been applied to nontrivial examples. However, these tools apply only to automata with finite control and, in practice, to relatively small systems. This limitation makes it difficult to apply these tools to fault-tolerant systems, as modeling faults typically leads to automata with a very large number of discrete states. Other real-time formalisms (e.g., [9, 10]) may be more expressive and general, but they have had limited tool support. In such frameworks, proofs are done by hand or, in some cases, with interactive theorem provers (e.g., [11, 12]). The modeling and verification method we discuss is applicable to a larger class of systems than timed automata, including some systems with infinite control, but it remains efficiently supported by a model-checking tool. Proofs are not completely automatic, as the user must provide auxiliary lemmas or candidate abstractions. However, the correctness of these lemmas or abstractions is checked automatically by the bounded model checker, much more efficiently than can be done using an interactive theorem prover. The method is not only efficient at reasoning about timed systems, but, as the startup example illustrates, it also copes with the discrete complexity introduced by faults.

2 Timed Systems in SAL

2.1 An Overview of SAL

SAL is a framework for the specification and analysis of traditional state-transition systems of the form $\langle S, I, \rightarrow \rangle$, where S is a state space, $I \subseteq S$ is the set of initial states, and \rightarrow is a transition relation on S . Each state σ of S is a mapping that assigns a value of an appropriate type to each of the system’s state variables. The core of SAL is a language for the modular specification of such systems. The relatively abstract and high-level specification language provides many of the types and constructs found in PVS, including infinite types such as the reals, the integers, and recursive data types, and therefore allows for specifying systems and their properties in a convenient and succinct manner. The main construct in SAL is the *module*. A module contains the specification of a state machine and can be composed with other modules synchronously or asynchronously.

Several analysis tools are part of the current SAL environment [2]. These include two symbolic model checkers, a SAT-based bounded model checker for finite systems, and a bounded model checker for infinite systems. This model checker, called `sal-inf-bmc`, searches for counterexamples to a given property by encoding transition relation and initialization into logical formulas in the theory supported by the ICS solver. ICS is a decision procedure and satisfiability solver for a combination of quantifier-free theories that include linear arithmetic over the reals and the integers, equalities with uninterpreted function symbols, propositional logic, and others [13, 14]. `sal-inf-bmc` can also use other solvers, if they can decide the appropriate theories.

Although bounded model checking is primarily a refutation method, the symbolic techniques it employs can be extended to proof by induction as discussed in [15]. `sal-inf-bmc` can be used to prove that a system $\mathcal{M} = \langle S, I, \rightarrow \rangle$ satisfies a formula $\Box P$, using k -induction, which consists of the two following stages:

- *Base case*: Show that all the states reachable from I in no more than $k - 1$ steps satisfy P .
- *Induction step*: For all trajectories $\sigma_0 \rightarrow \dots \rightarrow \sigma_k$ of length k , show that

$$\sigma_0 \models P \wedge \dots \wedge \sigma_{k-1} \models P \Rightarrow \sigma_k \models P.$$

The usual induction rule is just the special case where $k = 1$. `sal-inf-bmc` also supports k -induction with auxiliary invariants as lemmas. This allows one to prove $\Box P$ under the assumption that a lemma $\Box Q$ is satisfied.

The k -induction rule can be more successful as a proof technique than standard induction, as it is a form of automated invariant strengthening. Proving the invariance of P by k -induction is equivalent to proving the invariance of $P \wedge \bigcirc P \wedge \dots \wedge \bigcirc^{k-1} P$ by one-step induction. For a sufficiently large k , this stronger property is more likely to be inductive than the original P . However, there are transition systems \mathcal{M} for which k -induction cannot do better than standard induction. For example, if the transition relation is reflexive, then it is easy to show that if $\Box P$ is not provable by standard induction, it is not provable either by k -induction with any $k > 1$.

2.2 Clock-based Models

A first step in applying SAL to timed systems is to find a convenient description of such systems as state machines. One may be tempted to start from an existing formalism — such as timed automata [16] or related models (e.g., [17, 18]) — whose semantics is typically defined by means of transition systems. Encoding such models in SAL is possible and leads to what may be called *clock-based* models. A clock-based system \mathcal{M} is built from a set C of real-valued state variables (the clocks) and a set A of discrete variables, with A and C disjoint. A state σ of \mathcal{M} is a mapping from $A \cup C$ to appropriate domains; in all initial state σ , we have $\sigma(c) = 0$ for every clock $c \in C$; and the transition relation consists of two types of transitions:

- *Time progress*: $\sigma \rightarrow \sigma'$ where, for some $\delta \geq 0$ and all clock c we have $\sigma'(c) = \sigma(c) + \delta$, and, for every discrete variable a , we have $\sigma'(a) = \sigma(a)$.
- *Discrete transitions*: $\sigma \rightarrow \sigma'$ where $\sigma'(c) = \sigma(c)$ or $\sigma'(c) = 0$ for all clock c .

We have experimented with clock-based models when translating and analyzing timed automata in SAL [19, 20] but we encountered several difficulties. First, the clocks vary continuously with time. This means that δ can be arbitrarily small in a time-progress transition. As a consequence, it is difficult to ensure progress. The transition system has infinite trajectories in which no discrete transition ever occurs and time remains bounded. These undesirable trajectories cannot be easily excluded and they make it difficult to analyze liveness properties. Idle steps are possible (i.e., time-progress transitions with $\delta = 0$), which makes k -induction useless, except for $k = 1$. Preserving modularity is another issue, as the SAL composition operators do not match the product of timed automata.

These issues can be solved to some extent, and the analysis of timed automata in SAL is possible using various encoding tricks. A better approach is to avoid continuous clocks and develop timed models that are better suited to SAL. For this purpose, we propose a modeling method inspired from *event calendars*, a concept that has been used for decades in simulation of discrete event systems.

2.3 Timeout-based Models

In discrete event simulation, a calendar (also called event list) is a data structure that stores future events and the times at which these events are scheduled to occur. Unlike a clock, which measures the time elapsed since its last reset, a calendar contains information about the future. By following this principle, we can model real-time systems as standard transition systems with no continuous dynamics. The general idea is to rely on state variables to store the time at which future discrete transitions will be taken.

A first class of models we consider are transition systems with timeouts. Their state variables include a variable t that stores the current time and a finite set T of *timeouts*. The variable t and the timeouts are all real-valued. The initial states and transition relation satisfy the following requirements:

- In any initial state σ , we have $\sigma(t) \leq \sigma(x)$ for all $x \in T$.
- If σ is a state such that $\sigma(t) < \sigma(x)$ for all $x \in T$ then the only transition enabled in σ is a *time progress transition*. It increases t to $\min(\sigma(T)) = \min\{\sigma(x) \mid x \in T\}$ and leaves all other state variables unchanged.
- Discrete transitions $\sigma \rightarrow \sigma'$ are enabled in states such that $\sigma(t) = \sigma(x)$ for some $x \in T$ and satisfy the following conditions
 - $\sigma'(t) = \sigma(t)$
 - for all $y \in T$ we have $\sigma'(y) = \sigma(y)$ or $\sigma'(y) > \sigma'(t)$
 - there is $x \in T$ such that $\sigma(x) = \sigma(t)$ and $\sigma'(x) > \sigma'(t)$.

In all reachable states, a timeout x never stores a value in the past, that is, the inequality $\sigma(t) \leq \sigma(x)$ is an invariant of the system. A discrete transition can be taken whenever the time t reaches the value of one timeout x . Such a transition must increase at least one such x to a time in the future, and if it updates other timeouts than x their new value must also be in the future. Whenever the condition $\forall x \in T : \sigma(t) < \sigma(x)$ holds, no discrete transition is enabled and time advances to the value of the next timeout, that is, to $\min(\sigma(T))$. Conversely, time cannot progress as long as a discrete transition is enabled.

Discrete transitions are instantaneous since they leave t unchanged. Several discrete transitions may be enabled in the same state, in which case one is selected non-deterministically. Several discrete transitions may also need to be performed in sequence before t can advance, but the constraints on timeout updates prevent infinite zero-delay sequences of discrete transitions.

In typical applications, the timeouts control the execution of n real-time processes p_1, \dots, p_n . A timeout x_i stores the time at which the next action from p_i must occur, and this action updates x_i to a new time, strictly larger than the current time t , where p_i will perform another transition. For example, we have used timeout-based modeling for specifying and verifying Fischer's mutual exclusion algorithm [19]. Instances of Fischer's protocol with as many as 53 processes can be verified using this method.

2.4 Calendar-based Models

Timeouts are convenient for applications like Fischer's protocol, where processes communicate via shared variables that they read or write independently. Process p_i has full control of its local timeout, which determines when p_i performs its transitions. Other processes have no access to p_i 's timeout and their actions cannot impact p_i until it "wakes up". To model interaction via message passing, we add *event calendars* to our transition systems.

A calendar is a finite set (or multiset) of the form $C = \{\langle e_1, t_1 \rangle, \dots, \langle e_n, t_n \rangle\}$, where each e_i is an event and t_i is the time when event e_i is scheduled to occur. All t_i s are real numbers. We denote by $\min(C)$ the smallest number among $\{t_1, \dots, t_n\}$ (with $\min(C) = +\infty$ if C is empty). Given a real u , we denote by $\text{Ev}_u(C)$ the subset of C that contains all events scheduled at time u :

$$\text{Ev}_u(C) = \{\langle e_i, t_i \rangle \mid t_i = u \wedge \langle e_i, t_i \rangle \in C\}$$

As before, the state variables of a calendar-based system \mathcal{M} include a real-valued variable t that denotes the current time and a finite set T of timeouts. In addition, one state variable c stores a calendar. These variables control when discrete and time-progress transitions are enabled, according to the following rules:

- In all initial state σ , we have $\sigma(t) \leq \min(\sigma(T))$ and $\sigma(t) \leq \min(\sigma(c))$.
- In a state σ , time can advance if and only if $\sigma(t) < \min(\sigma(T))$ and $\sigma(t) < \min(\sigma(c))$. A time progress transition updates t to the smallest of $\min(\sigma(T))$ and $\min(\sigma(c))$, and leaves all other state variables unchanged.
- Discrete transitions can be enabled in a state σ provided $\sigma(t) = \min(\sigma(T))$ or $\sigma(t) = \min(\sigma(c))$, and they must satisfy the following requirements:
 - $\sigma(t) = \sigma'(t)$
 - for all $y \in T$ we have $\sigma'(y) = \sigma(y)$ or $\sigma'(y) > \sigma(y)$
 - if $\sigma(t) = \min(\sigma(c))$ then $\text{Ev}_{\sigma'(t)}(\sigma'(c)) \subseteq \text{Ev}_{\sigma(t)}(\sigma(c))$
 - we have $\text{Ev}_{\sigma'(t)}(\sigma'(c)) \subset \text{Ev}_{\sigma(t)}(\sigma(c))$, or there is $x \in T$ such that $\sigma(x) = \sigma(t)$ and $\sigma'(x) > \sigma(t)$.

These constraints ensure that $\sigma(t) \leq \min(\sigma(T))$ and $\sigma(t) \leq \min(\sigma(c))$ are invariants: timeout values and the occurrence time of any event in the calendar are never in the past.

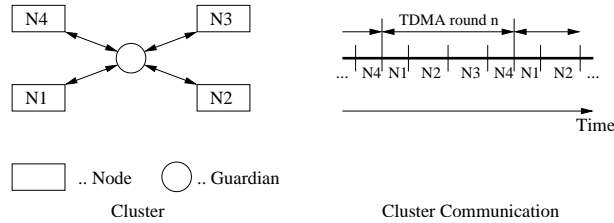


Fig. 1. TTA Cluster and TDMA Schedule.

Discrete transitions are enabled when the current time reaches the value of a timeout or the occurrence time of a scheduled event. The constraints on timeout are the same as before. In addition, a discrete transition may add events to the calendar, provided these new events are all in the future. To prevent instantaneous loops, every discrete transition must either consume an event that occurs at the current time or update a timeout as discussed previously.

Calendars are useful for modeling communication channels that introduce transmission delays. An event in the calendar represents a message being transmitted and the occurrence time is the time when the message will be received. The action of sending a message m to a process p_i is modeled by adding the event “ p_i receives m ” to the calendar, which is scheduled to occur at some future time. Message reception is modeled by transitions enabled when such event occurs, and whose effects include removing the event from the calendar. From this point of view, a calendar can be seen as a set of messages that have been sent but have not been received yet, with each message labeled by its reception time.

The main benefit of timeouts and calendars is the simple mechanism they provide for controlling how far time can advance. Time progress is deterministic. There are no states in which both time-progress and discrete transitions are enabled, and any state in which time progress is enabled has a unique successor: time is advanced to the point where the next discrete transition is enabled. This semantics ensures maximal time progress without missing any discrete transitions. A calendar-based model never makes two time-progress transitions in succession and there are no idle steps. All variables of the systems evolve in discrete steps, and there is no need to approximate continuous dynamics by allowing arbitrarily small time steps.

3 The TTA Startup Protocol

The remainder of this paper describes an application of the preceding modeling principles to the TTA fault-tolerant startup protocol [21]. TTA implements a fault-tolerant logical bus intended for safety-critical applications such as avionics or automotive control functions. In normal operation, N computers or nodes share a TTA bus using a time-division multiple-access (TDMA) discipline based on a cyclic schedule. The goal of the startup algorithm is to bring the system from the power-up state, in which the

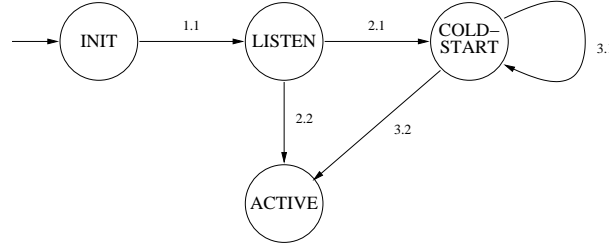


Fig. 2. State-machine of the TTA Node Startup Algorithm

N computers are unsynchronized, to the normal operation mode in which all computers are synchronized and follow the same TDMA schedule. A TTA system or “cluster” with four nodes and the associated TDMA schedule are depicted in Fig. 1. The cluster has a star topology, with a central hub or guardian forwarding messages from one node to the other nodes. The guardian also provides protection against node failures. It prevents faulty nodes from sending messages on the bus outside their allocated TDMA slot and, during startup, it arbitrates message collisions. A full TTA system relies on two redundant hubs and can tolerate the failure of one of them [21].

The startup algorithm executed by the nodes is described schematically in Fig. 2. When a node i is powered on, it performs some internal initializations in the INIT state, and then it transitions to the LISTEN state and listens for messages on the bus. If the other nodes are already synchronized, they each send an i -frame during their TDMA slot. If node i receives such a frame while in the LISTEN state, it can immediately synchronize with the other nodes and moves to the ACTIVE state (transition 2.2). After a delay τ_i^{listen} , if i has not received any message, it sends a cs -frame (coldstart frame) to initiate the startup process and moves to the COLDSTART state (transition 2.1). Node i also enters COLDSTART if it receives a cs -frame from another node while in the LISTEN state. In COLDSTART, node i waits for messages from other nodes. If i receives either an i -frame or a cs -frame, then it synchronizes with the sender and enters the ACTIVE state. Otherwise, if no frame is received within a delay $\tau_i^{\text{coldstart}}$, then i sends a cs -frame and loops back to COLDSTART (transition 3.1). The ACTIVE state represents normal operation. Every node in this state periodically sends an i -frame, during its assigned TDMA slot. The goal of the protocol is to ensure that all nodes in the ACTIVE state are actually synchronized and have a consistent view of where they are in the TDMA cycle.

The correctness of the protocol depends on the relative values of the delays τ_i^{listen} and $\tau_i^{\text{coldstart}}$. These timeouts are defined as follows:

$$\begin{aligned}\tau_i^{\text{listen}} &= 2\tau^{\text{round}} + \tau_i^{\text{startup}} \\ \tau_i^{\text{coldstart}} &= \tau^{\text{round}} + \tau_i^{\text{startup}}\end{aligned}$$

where τ^{round} is the round duration and τ_i^{startup} is the start of i 's slot in a TDMA cycle. Nodes are indexed from 1 to N . We then have $\tau_i^{\text{startup}} = (i - 1) \cdot \tau$ and $\tau^{\text{round}} = N \cdot \tau$ where τ is the length of each slot (this length is constant and all nodes have TDMA slots of equal length).

4 A Simplified Startup Protocol in SAL

We now consider the SAL specification of a simplified version of the startup protocol, where nodes are assumed to be reliable. Under this assumption, the hub has a limited role. It forwards messages and arbitrates collisions, but does not have any fault masking function. Since the hub has reduced functionality, it is not represented by an active SAL module but by a shared calendar.

4.1 Calendar

In TTA, there is never more than one frame in transit between the hub and any node. To model the hub, it is then sufficient to consider a bounded calendar that contains at most one event per node. To simplify the model, we also assume that the transmission delays are the same for all the nodes. As a consequence, a frame forwarded by the hub reaches all the nodes (except the sender) at the same time. All events in the calendar have then the same occurrence time and correspond to the same frame. These simplifications allow us to specify the calendar as shown in Fig. 3.

```
IDENTITY: TYPE = [1 .. N];
TIME: TYPE = REAL;
message: TYPE = { cs_frame, i_frame };

calendar: TYPE = [#
  flag: ARRAY IDENTITY OF bool,
  content: message,
  origin: IDENTITY,
  send, delivery: TIME
#];

empty?(cal: calendar): bool = FORALL (i: IDENTITY): NOT cal.flag[i];
...
i_frame_pending?(cal: calendar, i: IDENTITY): bool =
  cal.flag[i] AND cal.content = i_frame;
...
bcast(cal: calendar, m: message, i: IDENTITY, t: TIME): calendar =
  IF empty?(cal) THEN
    (# flag := [[j: IDENTITY] j /= i],
     content := m,
     origin := i,
     send := t,
     delivery := t + propagation #)
  ELSE cal WITH .flag[i] := false
  ENDIF;

consume_event(cal: calendar, i: IDENTITY): calendar =
  cal WITH .flag[i] := false;
```

Fig. 3. Calendar Encoding for the Simplified Startup Protocol

A calendar stores a frame being transmitted (*content*), the identity of the sender (*origin*), and the time when the frame was sent (*send*) and when it will be delivered (*delivery*). The boolean array *flag* represents the set of nodes that are scheduled to receive the frame. Example operations for querying and updating calendars are shown in Fig. 3. Function *bcast* is the most important. It models the operation “node *i* broadcasts

frame m at time t' and shows how collisions are resolved by the hub. If the calendar is empty when i attempts to broadcast, then frame m is stored and scheduled for delivery at time $t + propagation$, and all nodes except i are scheduled to receive m . If the calendar is not empty, then the frame from i collides with a frame m' from another node, namely, the one currently stored in the calendar. The collision is resolved by giving priority to m' and dropping i 's frame. In addition, node i is removed from the set of nodes scheduled to receive m' because channels between hub and nodes are half-duplex: since i is transmitting a frame m , it cannot receive m' .

4.2 Nodes

Figure 4 shows fragments of a node's specification in SAL. The node module is parameterized by a node identity i . It reads the current `time` via an input state variable, has access to the global calendar `cal` that is shared by all the nodes, and exports three output variables corresponding to its local `timeout`, its current state `pc`, and its view of the current TDMA `slot`. The transitions specify the startup algorithm as discussed previously using SAL's guarded command language. The figure shows two examples of transitions: `listen_to_coldstart` is enabled when time reaches node[i]'s `timeout` while the node is in the `LISTEN` state. The node enters the `COLDSTART` state, sets its `timeout` to ensure that it will wake up after a delay $\tau_i^{coldstart}$, and broadcasts a cs-frame. The other transition models the reception of a cs-frame while node[i] is in the `COLDSTART` state. Node i synchronizes with the frame's sender: it sets its `timeout` to the start of the next slot, compensating for the propagation delay, and sets its `slot` index to the identity of the cs-frame sender.

```

PC: TYPE = { init, listen, coldstart, active };

node[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    OUTPUT timeout: TIME, slot: IDENTITY, pc: PC
    GLOBAL cal: calendar
  INITIALIZATION
    pc = init;
    timeout IN { x: TIME | time < x AND x < max_init_time};
    ...
  TRANSITION
    ...
    [] listen_to_coldstart:
      pc = listen AND time = timeout -->
      pc' = coldstart;
      timeout' = time + tau_coldstart(i);
      cal' = bcast(cal, cs_frame, i, time)
    ...
    [] cs_frame_in_coldstart:
      pc = coldstart AND cs_frame_pending?(cal, i) AND time = event_time(cal, i) -->
      pc' = active;
      timeout' = time + slot_time - propagation;
      slot' = frame_origin(cal, i);
      cal' = consume_event(cal, i)
    ...

```

Fig. 4. Node Specification

4.3 Full Model

The complete startup model is the asynchronous composition of N nodes and a clock module that manages the `time` variable. The clock's input includes the shared calendar and the timeout variable from each node. The module makes time advances when no discrete transition from the nodes is enabled, as discussed in Sect. 2.4.

Since `time` cannot advance beyond the calendar's delivery time, pending messages are all received. For example, transition `cs_frame_in_coldstart` of Fig. 4 is enabled when `time` is equal to the frame reception time `event_time(cal, i)`. Let σ be a system state where this transition is enabled. Since the delivery times are the same for all nodes, the same transition is likely to be enabled for other nodes, too. Let's then assume that `cs_frame_in_coldstart` is also enabled for node j in state σ . In general, enabling a transition does not guarantee that it will be taken. However, the model prevents `time` from advancing as long as the frame destined for i or the frame destined for j is pending. This forces transition `cs_frame_in_coldstart` to be taken in both node i and node j . Since nodes are composed asynchronously, the transitions of node i and j will be taken one after the other from state σ , in a nondeterministic order. For the same reason, transitions that are enabled on a condition of the form `time = timeout` are all eventually taken. Timeouts are never missed.

5 Protocol Verification

5.1 Correctness Property

The goal of the startup protocol is to ensure that all the nodes that are in the `ACTIVE` state are synchronized (safety) and that all nodes eventually reach the `ACTIVE` state (liveness). We focus on the safety property. Our goal is to show that the startup model satisfies the following LTL formula with linear arithmetic constraints:

```
synchro: THEOREM
  system |-
    G(FORALL (i, j: IDENTITY): pc[i] = active AND pc[j] = active AND
      time < time_out[i] AND time < time_out[j] =>
        time_out[i] = time_out[j] AND slot[i] = slot[j])
```

This says that any two nodes in state `ACTIVE` have the same view of the TDMA schedule: they agree on the current slot index and their respective timeouts are set to the same value, which is the start of the next slot. Because nodes are composed asynchronously, agreement between i and j is not guaranteed at the boundary between two successive slots, when `time = time_out[i]` or `time = time_out[j]` holds.

5.2 Proof by Induction

A direct approach to proving the above property is the k -induction method supported by `sal-inf-bmc`. A first attempt with $k = 1$ immediately shows that the property is not inductive. Increasing k does not seem to help. The smallest possible TTA system has two nodes, and the corresponding SAL model has 13 state variables (5 real variables,

6 boolean variables, and 2 bounded integer variables).¹ On this minimal TTA model, k -induction at depth up to $k = 20$ still fails to prove the synchronization property.

However, as long as the number of nodes remains small, we can prove the property using k -induction and a few auxiliary lemmas:

```
time_aux1: LEMMA
  system |- G(FORALL (i: IDENTITY): time <= time_out[i]);

time_aux2: LEMMA
  system |- G(empty?(cal) OR
    (cal.send <= time AND time <= cal.delivery));

delivery_delay1: LEMMA
  system |- G(FORALL (i: IDENTITY):
    event_pending?(cal, i) =>
      event_time(cal, i) = cal.send + propagation);
```

The first two lemmas are invariants that hold for any calendar-based model, the other is an obvious relation between the transmit and reception time of messages. These lemmas are all inductive; they can be proved automatically by `sal-inf-bmc` using k -induction at depth 1.

For $N = 2$, we can then show that the synchronization property holds with the following command:

```
sal-inf-bmc -v 3 -d 8 -i -l time_aux1 -l time_aux2
  -l delivery_delay1 simple_startup4 synchro
...
proved.
total execution time: 258.71 secs
```

This instructs `sal-inf-bmc` to perform a proof by k -induction at depth 8 using the three lemmas. With $N = 3$, an inductive proof at depth 14 with the same lemmas fails; the execution time is of the order of 2 hours. With higher depths, `sal-inf-bmc` runs out of memory, or the user runs out of patience.

5.3 Proof via Abstraction

The previous verification uses only induction and is straightforward, but it has a major limitation: it works only for $N = 2$. The last step in the proof is not scalable, as the induction depth required increases with the number of nodes. To analyze the protocol with a larger number of nodes, we need a less expensive proof method. Since all we can do is proof by induction, our strategy is to strengthen the invariant. We are looking for an invariant ϕ that implies property `synchro`, and can be proved with `sal-inf-bmc` using induction at depth 1.

To obtain an appropriate ϕ , we use the method proposed by Rushby [3]. Given a transition system $\mathcal{M} = \langle S, I, \rightarrow \rangle$, this method amounts to constructing an abstraction of \mathcal{M} (or verification diagram [22]) based on n state predicates $A_1(\sigma), \dots, A_n(\sigma)$. The

¹ The variable `slot` of each process stores an integer in the interval $[1, N]$.

abstraction is a transition system $\mathcal{M}_0 = \langle S_0, I_0, \rightarrow_0 \rangle$ with state space $S_0 = \{a_1, \dots, a_n\}$. The abstract states are in a one-to-one correspondence with the n predicates. Then, the system \mathcal{M}_0 is a correct abstraction of \mathcal{M} if two properties are satisfied:

- For all state σ of I , there is an abstract state a_i of I_0 such that $A_i(\sigma)$ is satisfied.
- For every abstract state a_i , the following formula holds:

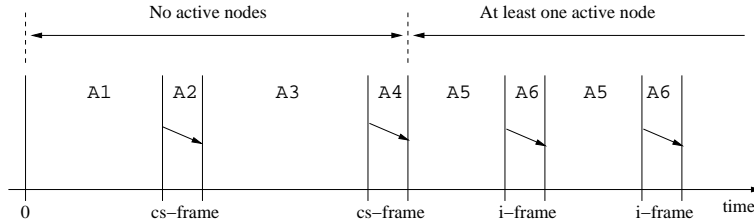
$$\forall \sigma \in S, \sigma' \in S : A_i(\sigma) \wedge \sigma \rightarrow \sigma' \Rightarrow A_{j_1}(\sigma') \vee \dots \vee A_{j_k}(\sigma'),$$

where a_{j_1}, \dots, a_{j_k} are the successors of a_i in \mathcal{M}_0 .

Less formally, the abstract system makes statements about \mathcal{M} of the form “if A_i is true in the current state, then the next state will satisfy A_{j_1} or \dots or A_{j_k} ”. It also states that some of the predicates A_1, \dots, A_n are true in all the initial states of \mathcal{M} . If the abstraction is correct, then clearly the disjunction $A_1 \vee \dots \vee A_n$ is an inductive invariant of \mathcal{M} .

This form of abstraction has two interests for our purposes. First, it is often relatively easy for the user to find adequate predicates A_1, \dots, A_n by “tracing” the execution of \mathcal{M} . Second, it is possible to prove that a candidate abstraction is correct using `sal-inf-bmc`. We illustrate this approach on the simplified startup algorithm.

Discovering the abstraction: By examining how the startup protocol works, one can decompose its execution into successive phases, as shown below:



In the first phase, A1, all nodes are either in the `INIT` or `LISTEN` states and no frame is sent. Phase A2 starts when one node enters `COLDSTART` and broadcasts a `cs-frame`, and ends when that frame is transmitted. Collisions may occur in phase A2 as several nodes may broadcast a `cs-frame` at approximately the same time. In phase A3, at least one node is in the `COLDSTART` state, and all nodes are waiting. In A4 a second `cs-frame` is sent. By definition of the delays $\tau_i^{\text{coldstart}}$, no collision can occur in A4. After A4, all the nodes that have received the second `cs-frame` become active. This leads to phase A5, in which at least one node is active. Phase A6 corresponds to the transmission of an `i-frame` by an active node. After A6, the system returns to phase A5, and so forth.

The six phases A1 to A6 form the basis of our abstraction. For example, the abstraction predicate A2 is defined in SAL as a boolean state variable as follows:

```
A2 = cs_frame?(cal) AND pc[cal.origin] = coldstart
    AND (FORALL (i: IDENTITY):
        pc[i] = init OR pc[i] = listen OR pc[i] = coldstart)
    AND (FORALL (i: IDENTITY): pc[i] = coldstart =>
        NOT event_pending?(cal, i)
        AND time_out[i] - cal.send >= tau_coldstart(i))
```

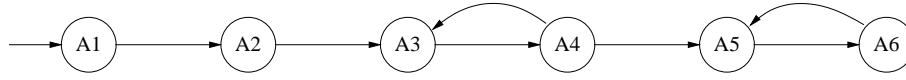


Fig. 5. Verification Diagram for the Simplified Startup

```

AND time_out[i] - time <= tau_coldstart(i)
AND (FORALL (i: IDENTITY): pc[i] = listen =>
  event_pending?(cal, i)
  OR time_out[i] >= cal.send + tau_listen(i));

```

Figure 5 shows the abstract system derived from A1 to A6. The transitions specify which phases may succeed each other. Every abstract state is also its own successor but we omit self loops from the diagram for clarity.

Proving that the abstraction is correct: Several methods can be used for proving in SAL that the diagram of Fig. 5 is a correct abstraction of the startup model. The most efficient technique is to build a monitor module that corresponds to the candidate abstraction extended with an error state. The monitor is defined in such a way that the error state is reached whenever the startup model performs a transition that, according to the abstraction, should not occur. For example, the monitor includes the following guarded command which specifies the allowed successors of abstract state a2:

```

state = a2 -->
  state' = IF A2' THEN a2 ELSIF A3' THEN a3 ELSE bad ENDIF

```

where bad is the error state. This corresponds to the diagram of Fig. 5: a2 and a3 are the only two successors of a2 in the diagram. The abstraction is correct if and only if the error state is not reachable, that is, if the property $\text{state} \neq \text{bad}$ is invariant. Furthermore, if the abstraction is correct, this invariant is inductive and can be proved automatically with `sal-inf-bmc` using k -induction at depth 1. This requires the same auxiliary lemmas as previously and an additional lemma per abstract state.

To summarize, our proof of the startup protocol is constructed as follows:

- An `abstractor` module defines the boolean variables A1 to A6 from the state variables of the concrete `tta` module.
- A `monitor` module whose input variables are A1 to A6 specifies the allowed transitions between abstract states.
- We then construct the synchronous composition of the `tta`, `abstractor`, and `monitor` modules.
- We show that this composition satisfies the invariant property $G(\text{state} \neq \text{bad})$, by induction using `sal-inf-bmc`.
- Finally, using `sal-inf-bmc` again, we show that the previous invariant implies the correctness property `synchro`.

5.4 Results

Table 1 shows the runtime of `sal-inf-bmc` when proving the correctness of the simplified TTA startup protocol, for different numbers of nodes. The runtimes are given in

N	Simplified Startup				Fault-Tolerant Startup			
	lemmas	abstract.	synchro	total	lemmas	abstract.	synchro	total
2	34.85	4.91	3.97	43.73	166.82	31.19	10.60	208.61
3	55.38	14.13	7.02	76.53	234.53	71.44	25.38	331.35
4	87.56	31.56	10.76	129.88	324.94	154.50	67.45	546.89
5	111.23	117.89	17.86	246.98	432.71	456.42	168.75	1057.88
6	154.92	334.31	26.53	515.76	547.51	731.60	346.35	1625.46
7	197.62	642.72	33.41	873.75	739.17	1143.48	648.49	2531.14
8	255.07	1400.34	45.08	1700.49	921.85	1653.10	1100.38	3675.33
9	316.36	2892.85	56.84	3266.05	1213.51	3917.37	1524.91	6655.79
10	378.89	4923.45	84.79	5387.13	1478.82	4943.18	3353.97	9775.97

Table 1. Verification Times

seconds and were measured on a Dell PC with a Pentium 4 CPU (2 GHz) and 1 Gbyte of RAM. The numbers are grouped in three categories: proof of all auxiliary lemmas, proof of the abstraction, and proof of the synchronization property. For small numbers of nodes (less than 5), proving the lemmas is the dominant computation cost, not because the lemmas are expensive to prove but because there are several of them. For larger numbers of nodes, checking the abstraction dominates.

Using the same modeling and abstraction method, we have also formalized a more complex version of the startup algorithm. This version includes an active hub that is assumed to be reliable, but nodes may be faulty. The verification was done under the assumption that a single node is Byzantine faulty, and may attempt to broadcast arbitrary frames at any time. With a TTA cluster of 10 nodes, the model contains 99 state variables, of which 23 variables are real-valued. The simplified protocol is roughly half that size. For a cluster of 10 nodes, it contains 52 state variables, of which 12 are reals.²

Other noticeable results were discovered during the proofs. In particular, the frame propagation delay must be less than half the duration of a slot for the startup protocol to work. This constraint had apparently not been noticed earlier. Our analysis also showed that the constants τ_i^{listen} do not need to be distinct for the protocol to work, as long as they are all at least equal to two round times.

6 Conclusion

We have presented a novel approach to modeling real-time systems based on calendars and timeouts. This approach enables one to specify dense-timed models as standard state-transition systems with no continuous dynamics. As a result, it is possible to verify these timed models using general-purpose tools such as provided by SAL. We have illustrated how the SAL infinite-state bounded model checker can be used as a theorem prover to efficiently verify timed models. Two main proof techniques were used: proof by k -induction and a method based on abstraction and verification diagrams. By decomposing complex proofs in relatively manageable steps, these techniques enable

² The full specifications are available at <http://www.sdl.sri.com/users/bruno/sal/>.

us to verify a nontrivial example of fault-tolerant real-time protocol, namely, the TTA startup algorithm, with as many as ten nodes.

This analysis extends previous work by Steiner, Rushby, Sorea, and Pfeifer [21] who have verified using model checking a discrete-time version of the same algorithm. They modeled a full TTA cluster with redundant hubs, and their analysis showed that the startup protocol can tolerate a faulty node or a faulty hub. This analysis went beyond previous experiments in model-checking fault-tolerant algorithms such as [23] and [24] by vastly increasing the number of scenarios considered. It achieved sufficient performance to support design exploration as well as verification.

Lönn and Pettersson [25] consider startup algorithms for TDMA systems similar to TTA, and verify one of them using UPPAAL [26]. Their model is restricted to four nodes and does not deal with faults. Lönn and Pettersson note that extending the analysis to more than four nodes will be very difficult, as the verification of a four-node system was close to exhausting the 2 Gbyte memory of their computer, and because of the exponential blowup of model checking timed automata when the number of clocks increases.

The model and verification techniques presented in this paper can be extended in several directions, including applications to more complex versions of the TTA startup algorithm with redundant hubs, and verification of liveness properties. Other extensions include theoretical studies of the calendar-automata model and comparison with timed automata.

References

1. Bensalem, S., Ganesh, V., Lakhnech, Y., Muñoz, C., Owre, S., Rueß, H., Rushby, J., Rusu, V., Saïdi, H., Shankar, N., Singerman, E., Tiwari, A.: An overview of SAL. In: Fifth NASA Langley Formal Methods Workshop, NASA Langley Research Center (2000) 187–196
2. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: Tool presentation: SAL 2. In: Computer-Aided Verification (CAV 2004), Springer-Verlag (2004)
3. Rushby, J.: Verification diagrams revisited: Disjunctive invariants for easy verification. In: Computer-Aided Verification (CAV 2000). Volume 1855 of Lecture Notes in Computer Science, Springer-Verlag (2000) 508–520
4. Steiner, W., Paulitsch, M.: The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. The 22nd International Conference on Distributed Computing Systems (ICDCS 2002) (2002)
5. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL: Status and developments. In: Computer-Aided Verification (CAV'97). Volume 1254 in Lecture Notes in Computer Science, Springer-Verlag (1997) 456–459
6. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Computer Aided Verification (CAV'98). Volume 1427 of Lecture Notes in Computer Science, Springer-Verlag (1998) 546–550
7. Wang, F.: Efficient verification of timed automata with BDD-like data-structures. In: 4th International Conference on Verification, Model Checking, and Abstract Interpretation. Volume 2575 of Lecture Notes in Computer Science, Springer-Verlag (2003) 189–205
8. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A tool for BDD-based verification of real-time systems. In: Computer-Aided Verification (CAV 2003), Volume 2725 of Lecture Notes in Computer Science, Springer-Verlag (2003) 122–125

9. Kaynar, D., Lynch, N., Segala, R., Vaandrager, F.: Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In: Real-Time Systems Symposium (RTSS'03). IEEE Computer Society (2003) 166–177
10. Chaochen, Z., Hansen, M.R.: Duration Calculus: A Formal Approach to Real-Time Systems. Springer-Verlag (2004)
11. Skakkebak, J.U., Shankar, N.: Towards a duration calculus proof assistant in PVS. In: Formal Techniques in Real-time and Fault-Tolerant Systems, Volume 863 of Lecture Notes in Computer Science, Springer-Verlag, (1994)
12. Archer, M., Heitmeyer, C.: Mechanical verification of timed automata: A case study. Technical Report NRL/MR/5546-98-8180, Naval Research Laboratory, Washington, DC (1998)
13. Filliâtre, J.C., Owre, S., Rueß, H., Shankar, N.: ICS: Integrated canonizer and solver. In: Computer-Aided Verification (CAV 2001), Volume 2102 of Lecture Notes in Computer Science, Springer-Verlag (2001) 246–249
14. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Fifth International Symposium on the Theory and Applications of Satisfiability Testing, (2002)
15. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Computer-Aided Verification (CAV 2003). Volume 2725 of LNCS., Springer-Verlag (2003) 14–26
16. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* **126** (1994) 183–235
17. Henzinger, T.A., Manna, Z., Pnueli, A.: Temporal proof methodologies for timed transition systems. *Information and Computation* **112** (1994) 273–337
18. Lynch, N., Vaandrager, F.: Forward and backward simulations for timing-based systems. In: REX Workshop. Real-Time: Theory and Practice. Volume 600 of Lecture Notes in Computer Science, Springer-Verlag (1991) 397–446
19. Dutertre, B., Sorea, M.: Timed systems in SAL. Technical report, SRI-SDL-04-03, SRI International, Menlo Park, CA (2004)
20. Sorea, M.: Bounded model checking for timed automata. *Electronic Notes in Theoretical Computer Science* **68** (2002) <http://www.elsevier.com/locate/entcs/volume68.html>
21. Steiner, W., Rushby, J., Sorea, M., Pfeifer, H.: Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. *DSN 2004* (2004)
22. Manna, Z., Pnueli, A.: Temporal verification diagrams. In: International Symposium on Theoretical Aspects of Computer Software (TACS'94). Volume 789 of Lecture Notes in Computer Science, Springer-Verlag (1994) 726–765
23. Yokogawa, T., Tsuchiya, T., Kikuno, T.: Automatic verification of fault tolerance using model checking. In: 2001 Pacific Rim International Symposium on Dependable Computing, Seoul, Korea (2001)
24. Bernardeschi, C., Fantechi, A., Gnesi, S.: Model checking fault tolerant systems. *Software Testing, Verification and Reliability* **12** (2002) 251–275
25. Lönn, H., Pettersson, P.: Formal verification of a TDMA protocol start-up mechanism. In: Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97), IEEE Computer Society (1997) 235–242
26. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer* **1** (1997) 134–152