SRI International

SDL Technical Report SRI-SDL-04-03 • October 21, 2004

Timed Systems in SAL

Bruno Dutertre and Maria Sorea*

* Abteilung Künstliche Intelligenz, Universität Ulm, Germany

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

Abstract

The Symbolic Analysis Laboratory (SAL) is a set of tools for the specification, exploration, and verification of state-transition systems. SAL includes symbolic model-checking tools based on solvers and decision procedures for linear arithmetic, uninterpreted functions, and propositional logic, among others. This enables the analysis of a variety of infinite-state systems. In particular, SAL can be used to model and verify timed systems, which combine real-valued and discrete state variables.

This document reports on several examples and experiments in modeling and verification of timed system in SAL. Different specification approaches are presented and compared, from a direct encoding of traditional timed automata to a novel modeling method based on event calendars. 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SAL, the Symbolic Analysis Laboratory [BGL+00], is a framework for the specification and analysis of concurrent systems. SAL is intended to combine different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. SAL complements the PVS verification system by providing more automated forms of analysis of systems that can be specified as transition relations (see [For03] for a description of SAL and a discussion of its relation to other SRI tools). SAL supports a relatively abstract and high-level specification language that includes many of the types and constructs found in PVS, and this allows for specifying systems and their properties in a convenient and succinct manner. The SAL language is not restricted to finite state systems and supports infinite types such as the reals, the integers, or recursive data types.

The current SAL environment includes tools for constructing and checking the well-formedness of specifications, an interactive simulator, and several model checkers [dMOR+04]. In particular SAL includes a tool that performs bounded model checking of infinite-state systems. This model checker relies on a solver for deciding the validity of logical formulas that can mix linear arithmetic, equalities with uninterpreted function symbols, and propositional logic. The default solver used by SAL is ICS [FORS01], which integrates a SAT solver and decision procedures for a combination of logical theories. The theories decided by ICS include linear arithmetic over reals and integers, equality with uninterpreted function symbols, and others. The SAL bounded model checker can also use other solvers provided they can decide the appropriate logical theories.

Many formalisms have been proposed for modeling and verifying real-time systems. Examples include different types of timed transition systems [AD94, KPSY93, HMP94, LV91, MMT91, JM94], timed process algebras [MT90, NS94, DS95], and real-time logics [CHR91, AH93, FMS88], among others. All these formalisms have in common the ability to specify and analyze infinite state-transition systems where delays are modeled via numerical variables that range over the reals or the integers. Since the SAL tools can reason about systems that mix real-valued and discrete variables, SAL is a natural candidate for the specification of real-time systems. This document examines several examples and their specification and analysis using the SAL tools. Because SAL has a rich language, it can support many different modeling approaches. We present and compare several possible approaches, and discuss their benefits and applicability to several types of real-time systems.

Chapter 2 considers the encoding in SAL of an existing formalism, namely, timed automata. It presents a translation of timed-automata models into SAL state-transition systems, and shows how the SAL infinite-state bounded model checker can be applied to the verification of such systems. The approach is illustrated on a simple train-gate controller introduced in [Alu91]. This approach works well as long as one is concerned with safety properties, but not so well if one is interested in liveness. Be-

cause SAL is intended for the modeling and analysis of discrete transition systems, not for systems with continuous dynamics, it is difficult to accurately capture the semantics of timed automata in SAL.

Chapters 3 and 4 introduce new classes of timed transition systems that do not require continuously varying clocks 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.

Chapter 3 considers a first class of models that rely on *timeouts*. This formalism is applied to a classic example of real-time systems, Fischer's mutual exclusion protocol. We prove the correctness of the protocol via a sequence of lemmas, all of which can be proved automatically by SAL's bounded model checker. We also show how the train-gate controller can be formalized using timeout automata.

A second class of timed transition systems is defined in Chapter 4. It extends timeout automata by adding *event calendar* to model inter-process communication. This model is applied to a more substantial example of real-time system, based on the fault-tolerant TTA startup protocol. A new verification technique is also presented that relies on a form of abstraction. This technique uses bounded model checking for automatically proving that an abstraction is correct, and provides efficient automation to support a proof method based on disjunctive invariants proposed by Rushby [Rus00].

The presentation assumes some familiarity with the SAL language and tools. An overview of the main aspects of SAL relevant to our examples is given in Appendix A. More complete descriptions of the language and tools are available via the SAL web site at `http://sal.csl.sri.com/`.

Chapter 2

Timed Automata in SAL

Timed automata [AD94] are one of the most widely used models of real-timed systems. This chapter discusses the translation of timed automata in SAL, and the verification of safety properties using the SAL infinite-state bounded model checker.

2.1 Timed Automata

Timed Automata [AD94] are state-transition graphs augmented with a finite set of real-valued variables called *clocks*. The states of the transition graphs are called *control locations* or simply *locations*. The full state of a timed automaton consists of its current control location and the values of all its clocks. Since the clocks are real-valued, timed automata have an infinite state space. A run of a timed automaton is the interleaving of two types of state transitions. *Discrete transitions* have zero duration; they update the control location and may reset some of the clocks. *Time-progress transitions* model the continuous evolution of the clocks as time passes. At a discrete transition, clocks keep their value or are reset to zero. During a time-progress transition, all the clocks increase at a uniform rate but the control location remains unchanged. States of the control graph and transition guards may be labeled with clock constraints, which determine when discrete transitions may occur.

Figure 2.1 shows a simple system built as the synchronous composition of three timed automata. The example, originally introduced in [Alu91], models a railway-crossing system that consists of a train, a gate, and a controller. The edges represent discrete transitions that change the control locations of each component. All transitions are labeled by events (e.g., *approach*, *exit*, or *lower*), and optionally by a clock constraint and by one or more clock-reset actions. For example, the event *approach* changes the train location from t_0 to t_1, and sets clock x to zero. The transition from t_1 to t_2 is labeled by the clock constraint $x > 2$, which means that the transition may be taken only if x is larger than 2. During time-progress transitions, the system locations do not change but the three clocks x, y, and z increase by the same amount. Some locations such as t_1 are labeled with a clock constraint or *location invariant*. The invariant constrains how long the system may stay in the location. For example, the train component may remain in location t_1 as long as the invariant $x \leqslant 5$ is satisfied.

Informally, the train component specifies that the event *approach* must occur at least two time units before the train enters the crossing (which is symbolized by the occurrence of event *in*), and that the delay between the train signaling its *approach* and its *exit* is at most five time units. Similarly, the controller component specifies that the command *lower* must be issued exactly one time unit after the event *approach*, and that *raise* must be issued at most one time unit after *exit*. The gate component specifies

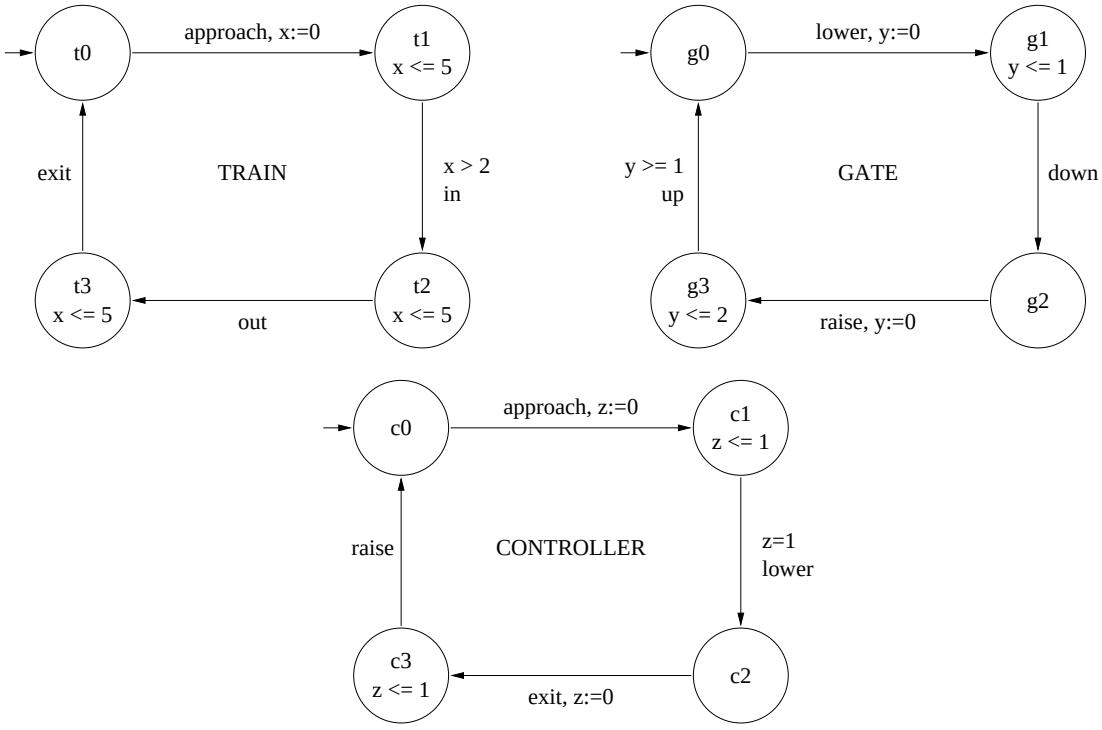


Figure 2.1: Train gate controller.

delays for lowering or raising the barriers. It takes at most one time unit for the barriers to close after the command *lower* is issued by the controller, and between one and two time units for the barriers to open after the command *raise* is sent.

The components function mainly on their own and communicate with each other only to perform specific tasks related to the crossing. This communication is modeled by letting the components synchronize on common events. The train and the controller components must synchronize on the events *approach* and *exit*, while the controller and the gate synchronize on *lower* and *raise*.

Timed automata are one of the most successful formalisms for real-time system verification because model checking timed automata is decidable. Given a timed automaton and a property expressed in a timed logic such as TCTL [ACD90] or T_μ [HNSY94], algorithms exist to automatically answer the question "does this timed automaton satisfy the given property?". The fundamental graph-theoretic model checking algorithm by Alur, Courcoubetis, and Dill [ACD90] constructs a finite quotient, the so-called *region graph*, of the infinite state graph. Algorithms directly based on the explicit construction of this quotient are rarely efficient in practice, since the number of equivalence classes of states of the region graph grows exponentially with the largest time constant and the number of clocks. Better algorithms use symbolic representations of the region graph obtained by characterizing regions as Boolean combinations of linear inequalities over clocks [HNSY94]. Tools specialized to the verification of timed automata — such as Uppaal [LPY97], Kronos [DOTY96], HyTech [HHWT97], and Tempo [Sor01] — employ these algorithms.

2.2 Translation to SAL

We now investigate a translation of timed automata to SAL, and the verification of properties using the SAL model-checking tools. The approach is illustrated using the train-gate-controller example of Figure 2.1 and follows closely the method proposed in [Sor02]. The translation attempts to preserve the structure of the example as much as possible: The three components are modeled as base SAL modules and the full system is the composition of these three modules.

Constructing this composition in SAL requires some care, as none of SAL's composition operators exactly matches the product of timed automata. The SAL language is designed for the specification of *unlabeled* state-transition systems that communicate via common state variables, while timed automata are *labeled* transition systems that communicate by synchronizing on common *events*. For example, in Figure 2.1, discrete transitions such as *in* or *out* are performed asynchronously. They cause the train component to change its control location but the other two components do nothing. The other discrete transitions are performed simultaneously by two components while the other is idle. On the other hand, during any time-progress transition, time advances by the same amount for all three components. Time-progress transitions are then performed synchronously by all components.

SAL allows one to compose state-transition systems either synchronously or asynchronously. In a synchronous composition, all components perform all transitions simultaneously, in locked steps. In an asynchronous composition, only one component performs a transition at a time while the others stay idle. The two types of composition can be mixed to build complex systems from base modules. For example, one may combine two SAL modules asynchronously and compose the result synchronously with another module. However, neither operation exactly fits the product of timed automata.

2.2.1 Synchronizer

Our solution is to adjoin a synchronizer component to the timed automata. The composition of n timed automata $M_1, \ldots, M_n$ is modeled in SAL as the synchronous composition of n modules $M_1, \ldots, M_n$ and a synchronizer S. At each step, the output variables of the synchronizer determine the type of transition to be performed — either a time-progress or a discrete transition. Each timed automaton M_i receives these variables as input. On a time-progress transition, the synchronizer outputs a delay δ, and every M_i increases its local clocks by δ. Time increases then by the same amount for all timed automata. On a discrete transition, the synchronizer outputs a label that specifies which action must be taken. Actions that require synchronization between two or more components are performed jointly by these components, while the others execute an idle step. Actions local to a component M_i are executed by M_i alone while the other components synchronously perform an idle step.

The synchronizer for the train-gate-controller example is the module `transition_module` shown in Figure 2.2. The figure also shows the definition of `TIME` and the enumerated types that represent actions and transition types. Since we use a dense-time model, time is represented by the reals. The type `ACTION` contains all the actions performed by the three components of Figure 2.1. The type `TransitionType` contains two entries: `regular` denotes a discrete step, where time does not elapse, while `elapse` specifies a time-progress step.

The function `next_trans_type` switches between discrete and time-progress transitions and is used by `transition_module` to alternate between time progress and discrete steps. This strict alternation between the two transition types is not absolutely necessary but it has the advantage of preventing consecutive time-progress steps from occurring. This considerably improves performance during analysis via bounded model checking, and has other advantages discussed in Section 2.4.

Apart from the alternation between `regular` and `elapse`, the synchronizer is completely non-deterministic. At each step, it may select an arbitrary action or let time advance by any non-negative

```
tgc: CONTEXT =
BEGIN

  TIME: TYPE = REAL;
  ACTION:  TYPE = {approach, in, out, exit, lower, down, raise, up};
  TransitionType : TYPE = {regular, elapse};
  ...
  next_trans_type(t: TransitionType): TransitionType =
      IF t = regular THEN elapse ELSE regular ENDIF;
  ...
  transition_module: MODULE =
  BEGIN
    OUTPUT
      delta:  REAL,
      action: ACTION,
      trans:  TransitionType
    TRANSITION
      delta' IN { x : REAL | x >= 0 };
      action' IN { approach, in, out, exit, lower, down, raise, up };
      trans' = next_trans_type(trans)
  END;
 ...
END
```

Figure 2.2: Synchronizer for the Train Gate Controller

value δ. Once `transition_module` is composed synchronously with other SAL modules, the latter can restrict the set of possible actions, or constrain the range of possible values for δ.

2.2.2 Base Modules

Each component of the train-gate-controller example is specified as a base SAL module. To illustrate the translation, we consider the train component. The possible control location of the corresponding module are defined via the following enumeration type:

```
T_STATE: TYPE = {t0,t1,t2,t3};
```

The input variables of the `train` module are as follows.

```
train : MODULE =
BEGIN
  INPUT
    delta: TIME,
    action: ACTION,
    trans: TransitionType
```

The input variables `delta`, `action`, and `trans` are the output variables of `transition_module`. They control which type of transitions the `train` module performs at each step.

Local variables represent the control location and clocks of the timed automaton. For example, the clock of the train module is represented by a real-valued variable `x` and its control location by the local variable `t_state`. Initially the train is in location `t0` and the value of the clock `x` is zero.

```
LOCAL
  t_state: T_STATE,
  x : TIME
INITIALIZATION
  t_state = t0;
  x = 0
```

The remainder of the SAL specification defines discrete and timed-progress transitions. The main issue is to make sure that the `train` module and the synchronizer component interact properly. To prevent deadlocks, `train` must specify the discrete events it accepts in every location, and it must also control how far time can advance when a time-progress step is taken.

For example, `approach` is the only discrete event that `train` can accept in location `t0`; the occurrence of events `in`, `out`, or `exit` while the train is in `t0` would cause a deadlock. In SAL, the events are selected non-deterministically by the synchronizer `transition_module` and are received by `train` on its input variable `action`. The specification must then ensure that this variable has a value other than `in`, `out`, and `exit` if the train location is `t0`. This requirement is not ensured by the synchronizer, but because `train` and `transition_module` are composed synchronously, the `train` module can specify the values of `action` it accepts.

A subtlety is that `action`, like any other module variable in SAL, is a state variable. At the start of a discrete transition, variable `action` has a current value. This value is what it is, and `train` cannot constrain it. On the other hand, `train` can impose constraints on the value `action` will have in the next state, that is, on `action'`. This enables `train` to refuse certain events. For example, by making sure that no discrete transition is enabled when `t_state = t0` and `action'` is equal to either `in`, `out`, or `exit`, the `train` module prevents the occurrence of these events when it is in location `t0`.

```
TRANSITION
  [ t0_t1:
    trans' = regular AND t_state = t0 AND action' = approach -->
      t_state' = t1;
      x' = 0

 [] t1_t2:
    trans' = regular AND t_state = t1 AND action' = in AND x > 2 -->
      t_state' = t2

 [] t2_t3:
    trans' = regular AND t_state = t2 AND action' = out -->
      t_state' = t3

 [] t3_t0:
    trans' = regular AND t_state = t3 AND action' = exit -->
      t_state' = t0
```

Figure 2.3: Discrete Transitions in SAL

The SAL specification of the discrete steps of the train automaton of Figure 2.1 follows these general principles. In addition to constraints on `action'`, one must also ensure that discrete transitions are enabled only when `trans'` is equal to `regular`. This guarantees that `train` does not perform a discrete transition while the other modules perform a time-progress step. The discrete transitions are specified as shown in Figure 2.3. Each guarded command in the figure encodes a discrete transition of the timed automaton. Clock constraints are specified in the guards, and each transition updates the control location `t_state` and possibly resets the clock `x`.

The time-progress transitions are specified in the same manner, but the `train` module must also constrain how far time can advance. This is necessary to make sure that location invariants are not violated. For example, let us assume that the train is in location t_1, with the clock x equal to some real number α. From such a state, a time-progress transition of duration δ leads to the state $(t_1, \alpha + \delta)$. By the location invariant associated with t_1, the train must leave t_1 before x becomes larger than 5 (cf. Figure 2.1). Thus, a time-progress transition taken from state (t_1, α) must be such that $\alpha + \delta \leqslant 5$. As previously, this is specified in SAL via constraints on `delta'`.

The constraints induced by the location invariants and the effect of the time-progress transition on the clock `x` are specified as follows.

```
[] delay_train:
   trans' = elapse AND
   (t_state = t1 => x + delta' <= 5) AND
   (t_state = t2 => x + delta' <= 5) AND
   (t_state = t3 => x + delta' <= 5) -->
      x' = x + delta'
```

This time-progress transition for the `train` module advances the train's clock `x` by `delta'`. The three invariant associated with locations `t1`, `t2`, and `t3` are specified in the guard. As a result the transition is enabled only if `x+delta'` satisfies the current location invariant.

To complete the `train` module, one must add an idle transition that encodes inactivity. This transition is taken when the other two components perform a discrete step on an action that does not require participation of the train:

```
[] skip_train:
   trans' /= elapse AND
   NOT (action' = approach OR action' = in OR
        action' = out OR action' = exit) -->
     x' = x
```

2.2.3 Full System

The gate and controller are encoded in the same way as the train component, as shown in Appendix B. The entire system is specified as the synchronous composition of the `train`, `gate`, and `controller` modules, together with the `transition_module`.

```
system: MODULE =
   transition_module || train || gate || controller;
```

2.3 Example Verifications

The specification of the train-gate-controller system is now complete. The safety property we wish to check says that the gate should be closed whenever the train is in the crossing. This is specified as follows:

```
safe: LEMMA system |- G(t_state = t2 => g_state = g2);
```

The symbol `G` represents the *always* modality of linear temporal logic.

As a first example of analysis, we run the infinite-state bounded model checker `sal-inf-bmc` to search for counterexamples to property `safe`:

```
sal-inf-bmc -d 1 -v 1 tgc safe
...
no counterexample between depths: [0, 1].
total execution time: 0.46 secs
```

The option `-d 1` specifies the depth of the search, in this case, only one step, and the option `-v 1` controls how much information `sal-inf-bmc` outputs. Argument `tgc` is the name of the SAL context in which the property `safe` is specified. SAL searches for this context in a file named `tgc.sal`.

As expected, `sal-inf-bmc` does not find any counterexample at depth one. We can increase the search depth by hand, but it is more efficient to run the bounded model checker with iterative deepening. With this option, `sal-inf-bmc` searches for counterexamples of increasing length. It stops when either a counterexample is detected or the search range is covered:

```
sal-inf-bmc -v 1 -d 56 -it tgc safe
...
no counterexample between depths: [0, 56].
total execution time: 975.68 secs
```

Iterated deepening is indicated by the option `-it` and `-d 56` specifies the maximal depth. It turns out that, for this example, searching for counterexamples up to depth 56 is sufficient. The completeness threshold is 56. There are 28 reachable symbolic states, and between every discrete step there is at most one time-progress step, which gives a maximal path length of 56. Since no counterexample of length 56 or less has been found, we can conclude that our system is safe. Of course, this argument requires user's knowledge of the size of the symbolic state space.

Bounded model checking is a primarily a refutation method, intended to discover counterexamples to properties, but it can also be used for performing proofs by k-induction [dMRS03]. For example, property `safe` can be proved by k-induction at depth $k = 9$. For k-induction, `sal-inf-bmc` must be invoked with the `-i` flag and the depth k is specified using the `-d` option:

```
sal-inf-bmc -v 1 -d 9 -i tgc safe
...
proved.
total execution time: 2.01 secs
```

This is much more efficient than the previous approach.

We can also introduce a bug in the model, for example, by changing the guard $x > 2$ in the transition from t_1 to t_2 to $x > 0$. With this modification, `sal-inf-bmc` with iterative deepening finds a counterexample at depth 4, in 1.76 secs.

2.4 Discussion

As the example illustrates, it is possible to specify timed automata in SAL in a systematic fashion. The translation preserves modularity: each component of the example is translated into a base module in SAL, and the product of the timed automata corresponds to the synchronous composition of SAL modules and a synchronizer.

The translation is intended to preserve all safety properties of the original system. Given a predicate P on clocks and control locations, one can show that the timed-automata system satisfies $\Box P$ if and only if the SAL translation also satisfies $\Box P$. Preserving safety properties of this form is sufficient in most applications. However, one should be aware that the translation may introduce deadlocks in SAL that do

not exist in the original system. This is caused by the strict alternation between time-progress and discrete transitions enforced by the synchronizer. The encoding of time-progress steps ensures that location invariant are satisfied, that is, that time does not advance too far. On the other hand, no lower bound is imposed on `delta'`; a time-progress step can be of arbitrarily small duration. In some cases, a time step that is too small may lead to a state in which no discrete transition is enabled, and thus to a deadlock.

For example, it is possible for the train-gate-controller system to reach a state in which the train is in location t_1, the controller is in c_1, and the gate is in g_3. From such a state, a discrete transition is enabled only if one of the clock constraints $x > 2$ and $y \geqslant 1$ is satisfied. In the SAL model, a time-progress step taken in such a state increases both `x` and `y` by `delta'`. If `delta'` is too small, the system may reach a state in which `x'` is less than 2 and `y'` is less than 1. From this state, no discrete transition is enabled and the SAL system is in a deadlock. The original timed automaton can also reach the same state, but this does not cause a deadlock as it can perform another time step.

The introduction of extra deadlocks in the SAL translation rarely matters in practice, as the SAL model still has the same set of reachable states as the original timed automaton. It is also clear that one could modify the synchronizer definition so that several time-progress steps may be taken in succession. One just needs to replace the line

```
trans' = next_trans_type(trans)
```

by the following non-deterministic assignment

```
trans' IN { elapse, regular }.
```

With this modification, the SAL system agrees with the traditional semantics of timed automata and does not have extra deadlock states anymore. However, this revised specification has two disadvantages.

First, it considerably increases the cost of bounded model checking. When the train-gate-controller example is modified to permit successive time steps, `sal-inf-bmc` is considerably slower than when successive time-steps are disallowed. A search for counterexamples to property `safe` using iterative deepening takes more than six hours:

```
sal-inf-bmc -v 1 -d 35 -it tgc safe
...
no counterexample between depths: [0, 35].
total execution time: 22932.49 secs
```

A similar verification at depth 56 was completed in less than 20 min using the previous specifications.

Allowing several time-progress steps to occur in succession has a more serious drawback: it makes most proofs by k-induction fail. More precisely, if P is a state-predicate and the safety property $\Box P$ cannot be proved by k-induction when $k = 1$, then $\Box P$ is not provable either by k-induction for any $k > 1$. For example, property `safe` cannot be proved by k-induction if successive time-progress steps are allowed, while it is provable by k-induction at depth 9 if discrete and time-progress steps are forced to alternate.

The reason why k-induction fails for $k > 1$ if it fails for $k = 1$ is that a trajectory of length one can always be extended into a trajectory of length k by adding a succession of time-progress steps of duration zero, which behave like idle steps. In particular, if $\Box P$ cannot be proved by k-induction at depth one, then there are two states σ and σ' of the SAL model such that

- σ satisfies P,
- σ' is a successor state of σ,
- σ' does not satisfy P.

In other words, the trajectory $\sigma \longrightarrow \sigma'$ is a counterexample to the induction step in the proof of $\Box P$. The trajectory $\sigma \longrightarrow \sigma'$ can be extended into another trajectory of the form

$$\sigma \xrightarrow{0} \sigma_1 \ldots \sigma_{k-2} \xrightarrow{0} \sigma_{k-1} \longrightarrow \sigma',$$

where the first $k-1$ steps are time-progress transitions of length zero. We then have $\sigma_1 = \ldots = \sigma_{k-1} = \sigma$, and the transition $\sigma_{k-1} \longrightarrow \sigma'$ is the same as the transition that leads from σ to σ'. The existence of this trajectory means that k-induction at depth k will fail for $\Box P$.

Removing idle steps, that is, requiring all time steps to have a positive duration does not fix this problem in general (and causes other problems since zero-delay time steps are sometimes necessary). In typical examples, a discrete transition that is enabled in a state σ remains enabled if the clocks progress in σ by a small enough amount. In such a case, a sequence of time-progress transitions with sufficiently small durations is essentially the same as an idle step. By the same reasoning as before, a counterexample of length 1 can be extended into a counterexample of length k by adding small time-progress steps. As a consequence, induction at depth $k > 1$ cannot do better than induction at depth 1.

In summary, adopting the traditional semantics of timed automata — which allows several time steps to occur in succession — is a poor choice if one intends to prove properties via k-induction. It is acceptable if one is interested only in finding counterexamples by bounded model checking, but this typically incurs a significant loss of performance. A synchronizer that alternates between time-progress and discrete transitions as presented in Figure 2.2 is generally preferable as it preserves all safety properties, makes k-induction possible, and is more efficient for bounded model checking.

Overall, we can conclude that it is possible to describe timed automata in SAL in a systematic way that preserves modularity and safety properties. The translation amounts to encoding the semantics of timed automata as state-transition systems, with an explicit distinction between time-progress and discrete transitions. Using this translation, one can analyze timed automata with the SAL tools, which support bounded model checking and verification by k-induction. However, the translation is not as straightforward as one would wish. The product of timed automata requires a synchronizer module and the addition of idle transitions to all timed modules. This can become unwieldy if the number of events and timed modules is large. The encoding of time-progress transition must also be done with care to ensure that the location invariants are not violated. Furthermore, the translation we use does not exactly match the standard timed-automata semantics as it does not allow successive time-progress transitions to occur. This modified semantics is necessary for making k-induction useful, but its drawback is to introduce deadlocks that do not exist in the traditional semantics. The translation is not perfect but it is sufficient for the properties most often encountered in practice.

Chapter 3

Timeout Automata

Real-time systems are *discrete-event systems*, that is, systems whose state does not change continuously but only at discrete points in time. Any evolution or trajectory of such a system can be represented by an increasing sequence of time points at which events occur (and where the state may change) together with the state of the system at each of these time points. Such a real-time trajectory may be written as a sequence of pairs

$$\langle t_0, \sigma_0 \rangle, \langle t_1, \sigma_1 \rangle, \langle t_2, \sigma_2 \rangle, \ldots$$

where $t_0, t_1, t_2, \ldots$ is an increasing sequence of time points, and $\sigma_0, \sigma_1, \sigma_2, \ldots$ are the system states.

Timed automata are one possible formalism for specifying such trajectories. A trajectory of a timed automaton is any sequence as above that satisfies the two following conditions:

- $t_0 = 0$ and σ_0 is an initial state of the automaton.
- For all $i \in \mathbb{N}$, either σ_{i+1} is a successor of σ_i by a time-progress transition of duration $\delta \geqslant 0$ and $t_{i+1} = t_i + \delta$, or σ_{i+1} is a successor of σ_i by a discrete step and $t_{i+1} = t_i$.

Starting with timed automata is not the only option for defining such trajectories. In the remainder of this report, we examine two variant specification approaches that are better suited to SAL and have other advantages over timed automata. Both approaches are inspired by a very successful modeling method that has been used for decades in discrete-event system simulation, namely, the use of *event calendars*.

Such 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, one can model real-time systems as standard state-transition systems of the form 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. The state space S is built from a collection of state variables: each state σ of S is a mapping that assigns a value of an appropriate type to each of the system's state variables. To specify delays and timing constraints, we rely on special state variables to store the time at which future discrete transitions will be taken.

3.1 Definition

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 $\sigma(t) \leqslant \sigma(x)$ 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 $\sigma \to \sigma'$ 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 $\sigma(t) \leqslant \sigma(x)$ 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 $p_1, \ldots, p_n$. 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. As an illustration, the following section presents a SAL model of Fischer's mutual exclusion algorithm based on timeout automata, and a correctness proof developed with SAL's infinite-state bounded model checker. We then show how the timeout-automata model can be applied to the train-gate-controller example described previously.

3.2 Fischer's Mutual Exclusion Protocol

Fischer's protocol has become a benchmark in the evaluation of real-time formalisms and tools. The protocol ensures mutual exclusion among N processes using real-time clocks and a shared variable `lock`, whose value range from 0 to N. Initially the value of `lock` is zero. Processes are indexed by an integer between 1 and N (the process id) and behave as follows:

```
loop
  wait until lock = 0;          % sleeping state
  wait for a delay <= delta1;   % waiting state
  set lock to process id;
  wait for a delay >= delta2;   % trying state
  if lock = process id
     critical section;          % critical state
     set lock to 0;
  end
end
```

A process can then be modeled as an automaton with four control states as shown in Figure 3.1. The states correspond to the `wait` statements and to the critical section.

The parameters `delta1` and `delta2` are two positive constants that determine the length of the delays:

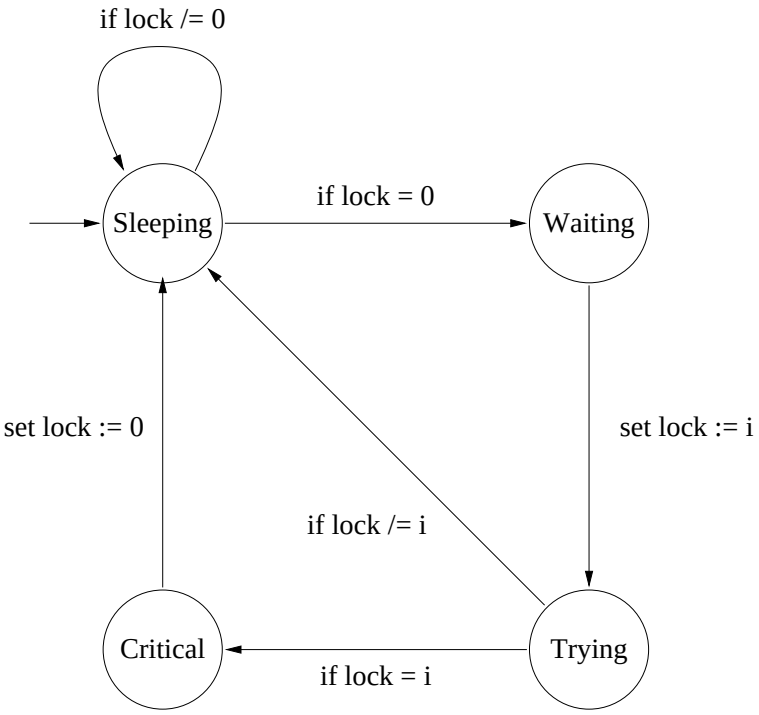


Figure 3.1: Fischer's Protocol

- a process can stay in state `waiting` for a delay at most `delta1`
- a process must stay in state `trying` for at least a delay `delta2` before transitioning to state `critical`

Mutual exclusion is ensured provided `delta1` is strictly smaller than `delta2`.

3.2.1 SAL Model

We begin by specifying the number of processes, N, and the types over which the state variables range.

```
fischer: CONTEXT =
BEGIN
  N: NATURAL = 4;
  IDENTITY: TYPE = [1 .. N];
  LOCK_VALUE: TYPE = [0 .. N];
  TIME: TYPE = REAL;
  PC: TYPE = {sleeping, waiting, trying, critical};
```

For the time being, we give explicit values to the two parameters `delta1` and `delta2`. An alternative specification where the parameters are left uninterpreted is discussed in Section 3.2.4.

```
  delta1: TIME = 2;
  delta2: TIME = 4;
```

Process Module

Figure 3.2 shows how the processes are specified in SAL. A process is modeled as a SAL module parameterized by the process's identity `i`. All the processes have a common input variable `time` that keeps the current time in the system. The global variable `lock` is shared by all the processes (i.e., every process can read and write to `lock`). Each process also outputs its `timeout` variable, which gives the time of the process's next transition. The local variable `pc` holds the process's current control state.

The initialization section sets the `lock` to zero, and assigns a value larger than the initial `time` to the process's `timeout` variable.

The transition relation consists of six guarded commands. These six transitions are enabled only when the global `time` equals the process's `timeout`. Every transition also updates `timeout` to a value that is strictly larger than `time`. The updates are non-deterministic: every transition assigns to `timeout` an arbitrary value in a real-valued interval. The bounds of this interval specify maximal or minimal delays between successive transitions from the process. For example, on transition `waking_up`, the `timeout` is set non-deterministically to a time point in the interval (`time`, `time` + `delta1`]. The process will perform its next transition within a delay less than or equal to `delta1`, and will then stay in control state `waiting` for a delay no more than `delta1`. Similarly, transition `setting_lock` specifies that the process must stay in control state `trying` for a delay at least equal to `delta2`. In all other cases, the transition updates `timeout` to an arbitrary time in the future.

Clock Module

To complete the specification, a clock component is introduced to perform the time-progress transitions. This `clock` module is shown in Figure 3.3. Its unique input variable is an array of N timeouts, one per process. The `clock` advances the global `time` variable to the smallest element of the `time_out` array, whenever `time` is strictly smaller than all timeouts. This relies on the function `min` that computes the minimum of an array:

```
recur_min(x: TIMEOUT_ARRAY, min_sofar: TIME, idx: [0 .. N]): TIME =
  IF idx = 0 THEN min_sofar
     ELSE recur_min(x, min(min_sofar, x[idx]), idx-1)
  ENDIF;

min(x: TIMEOUT_ARRAY): TIME = recur_min(x, x[N], N-1);
```

Full System

The complete system is the asynchronous composition of the N processes and the `clock` module:

```
processes: MODULE =
  WITH OUTPUT time_out: TIMEOUT_ARRAY
     ([] (i: IDENTITY): (RENAME timeout TO time_out[i] IN process[i]));

system: MODULE = clock [] processes;
```

In the full system, the global variable `time` keeps the current time and the variable `time_out[i]` contains the time when the next discrete transition of `process[i]` is scheduled to occur. As discussed in Section 3.1, there are two types of transition:

```
process[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    GLOBAL lock: LOCK_VALUE
    OUTPUT timeout: TIME
    LOCAL pc: PC
  INITIALIZATION
    pc = sleeping;
    timeout IN { x: TIME | time < x };
    lock = 0
  TRANSITION
    [ waking_up:
      pc = sleeping AND time = timeout AND lock = 0 -->
           pc' = waiting;
           timeout' IN { x: TIME | time < x AND x <= time + delta1 }
   [] try_again_later:
      pc = sleeping AND time = timeout AND lock /= 0 -->
           timeout' IN { x: TIME | time < x }
   [] setting_lock:
      pc = waiting AND time = timeout -->
           pc' = trying;
           lock' = i;
           timeout' IN { x: TIME | time + delta2 <= x }
   [] entering_cs:
      pc = trying AND time = timeout AND lock = i -->
           pc' = critical;
           timeout' IN { x: TIME | time < x }
   [] lock_changed:
      pc = trying AND time = timeout AND lock /= i -->
           pc' = sleeping;
           timeout' IN { x: TIME | time < x }
   [] exiting_cs:
      pc = critical AND time = timeout -->
           pc' = sleeping;
           lock' = 0;
           timeout' IN { x: TIME | time < x }
    ]
  END;
```

Figure 3.2: Fischer's Protocol: Process Module

```
TIMEOUT_ARRAY: TYPE = ARRAY IDENTITY OF TIME;

clock: MODULE =
  BEGIN
    INPUT time_out: TIMEOUT_ARRAY
    OUTPUT time: TIME
  INITIALIZATION
    time = 0
  TRANSITION
    [ time_elapses: time < min(time_out) --> time' = min(time_out) ]
  END;
```

Figure 3.3: Fischer's Protocol: Clock Module

- If `time < time_out[i]` for all `i`, then `clock` advances time to to the smallest of `time_out[1]`, ..., `time_out[N]` (*time-progress transition*).
- Otherwise, the component `process[i]` for which `time_out[i] = time` performs a *discrete transition*. This transition updates `time_out[i]` to a new value strictly larger than the current time.

Several processes may have discrete transitions enabled at the same instant, that is, we may have `time_out[i] = time_out[j] = time` for two distinct processes `i` and `j`. Since `time` cannot progress as long as `time_out[i] = time` or `time_out[j] = time`, both discrete transitions will be taken, one after the other, in a non-deterministic order.

By construction, the property `time <= time_out[i]` is an invariant of the system. As a consequence, in every reachable state, either the clock or one of the processes has a transition enabled. Since the modules are composed asynchronously, the full system cannot deadlock.

3.2.2 Analysis

The goal of Fischer's protocol is to ensure mutual exclusion, that is, that two distinct processes cannot be in their critical sections at the same time:

```
mutual_exclusion: THEOREM
    system |- G(FORALL (i, j: IDENTITY):
                  i /= j AND pc[i] = critical => pc[j] /= critical);
```

Bounded Model Checking

It is good practice to first search for possible errors in the SAL specifications by using the bounded model checker. For this purpose, we search for trajectories that violate the mutual-exclusion property. Since we expect the property to hold, such a trajectory is likely to indicate a bug in the SAL model. For a small-size system ($N = 2$), the bounded model checker shows that there are no counterexamples of length 20 or less:

```
sal-inf-bmc -v 3 -it -d 20 fischer mutual_exclusion
...
no counterexample between depths: [0, 20].
total execution time: 697.62 secs
```

We can repeat this analysis with $N = 3$ and we still obtain the expected result: no counterexample is found.

A second validation step is to introduce bugs in the specifications and check that the bounded model checker finds them. For example, we have stated that mutual exclusion is satisfied provided `delta1` is strictly smaller than `delta2`. We can now examine what happens if the condition `delta1 < delta2` does not hold. First, we modify the definitions:

```
  delta1: TIME = 2;
  delta2: TIME = 2;
```

then, we check that `mutual_exclusion` is not satisfied with these parameters:

```
sal-inf-bmc -v 3 -it -d 20 fischer mutual_exclusion
...
Counterexample:
=======================
Path
=======================
Step 0:
--- System Variables (assignments) ---
lock = 0;
pc[1] = sleeping;
pc[2] = sleeping;
pc[3] = sleeping;
time = 0;
--- Constraints ---
time_out[1] > 0;
time_out[3] >= time_out[2];
time_out[3] > 0;
time_out[3] >= 0;
time_out[1] >= 0;
time_out[2] > 0;
time_out[1] > time_out[2];
time_out[2] >= 0;
-----------------------
...
Step 10:
--- System Variables (assignments) ---
lock = 1;
pc[1] = critical;
pc[2] = critical;
pc[3] = sleeping;
--- Constraints ---
time_out[1] >= 0;
time_out[2] >= 0;
time_out[3] = PRE(time_out)[3];
time = PRE(time);
time >= 0;
time_out[3] >= 0;
time_out[2] = PRE(time_out)[2];
time_out[1] > PRE(time);

total execution time: 12.32 secs
```

The shortest counterexample has length 10. It is displayed by the bounded model checker in a symbolic form that represents a sequence of states and transitions. As can be seen above, the mutual exclusion property is not satisfied.

A First Proof

The analysis by bounded-model checking discussed above gives us some confidence that mutual exclusion is satisfied, as no counterexample was found. We now examine how the property can be proved by k-induction.

We start with a small-sized example, by setting the number of processes N to 2. However, even for such a small N, direct attempts at proving `mutual_exclusion` by k-induction fail. For example, with $k = 14$, we obtain

```
sal-inf-bmc -v 3 -d 15 -i fischer mutual_exclusion
...
k-induction rule failed, please try to increase the depth.
total execution time: 31.99 secs
```

Increasing k does not help as `mutual_exclusion` is not inductive, whatever k. For any $k > 0$, one can find a trajectory

$$\sigma_0 \rightarrow \ldots \rightarrow \sigma_k,$$

of the transition system such that mutual exclusion holds in the states $\sigma_0, \ldots, \sigma_{k-1}$ but not in σ_k. As a consequence, the inductive step in the proof of `mutual_exclusion` by k-induction fails.

It is not very hard to find the trajectory mentioned above by hand, but, for a fixed k, it is even easier to let the model checker do it. For this purpose, we invoke `sal-inf-bmc` with the `-ice` option:

```
sal-inf-bmc -v 3 -d 14 -i -ice fischer mutual_exclusion
...
k-induction rule failed, please try to increase the depth.
Counterexample:
=======================
Path
=======================
...
-----------------------
Step 15:
--- System Variables (assignments) ---
lock = 1;
pc[1] = critical;
pc[2] = critical;
--- Constraints ---
time = PRE(time);
time_out[1] > PRE(time);
time >= 0;
time_out[1] >= 0;
time_out[2] >= 0;
time_out[2] = PRE(time_out)[2];

total execution time: 51.28 secs
```

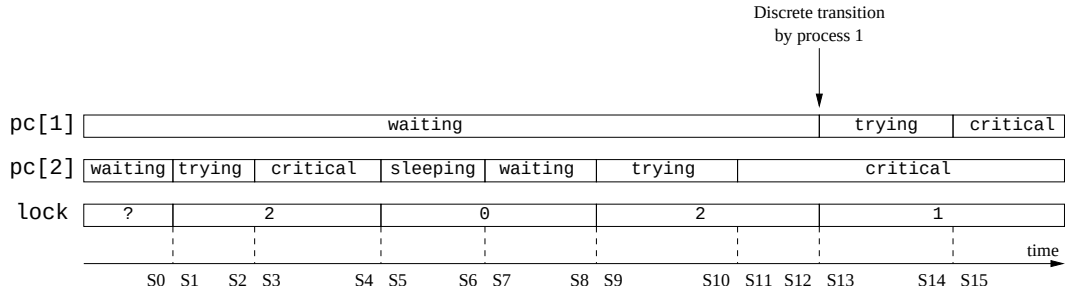


Figure 3.4: Counterexample to *k*-induction

Figure 3.4 depicts the counterexample to *k*-induction found by the model checker. The diagram shows the evolution of the system's three discrete variables with time. The counterexample consists of the states S_0 to S_{15} shown in the figure; discrete and time-progress transitions alternate: state S_1 is reached from S_0 by a discrete transition, then S_2 is reached from S_1 by a time-progress step, and so forth. The main features of this trajectory are the following:

- It starts from a state S_0 in which `pc[1]=waiting`.
- Process 1 stays idle until the discrete transition from S_{12} to S_{13} and, after two more steps, we have `pc[1]=critical`.
- From S_0 to S_{12}, the trajectory consists exclusively of discrete steps by process 2 and time-progress transitions, and after the last discrete step from process 2 (from S_{11} to S_{15}), we have `pc[2]=critical`.

One can construct a trajectory $S_0, \ldots, S_k$ with the same essential features for any k:

- S_0 is a state where `pc[1]=waiting`.
- From S_0 to S_{k-3}, only process 2 executes, and we have `pc[2]=critical` in states S_{k-3} to S_k.
- In the last three steps $S_{k-3} \to S_{k-2} \to S_{k-1} \to S_k$, process 1 enters its critical section.

The existence of such a trajectory means that *k*-induction cannot prove the mutual-exclusion property for any k. The property is not inductive.

The trajectory of Figure 3.4 causes a direct proof by *k*-induction to fail, but it also points to a solution. In the figure, process 1 stays in state `waiting` for a long time before executing its first discrete step. This time is more than `delta2` since it includes an interval during which process 2 is in state `trying`. In state S_0, the difference `time_out[1] - time` must then be greater than `delta2`, which is itself greater than `delta1`. But, by construction, a process cannot stay in control state `waiting` for more than the delay `delta1`. Thus, S_0 is not reachable by the transition system; it does not satisfy the property `pc[1]=waiting => time_out[1] - time <= delta1`, which we know to be invariant.

To prove `mutual_exclusion`, we can then attempt to use the following auxiliary invariant:

```
time_aux2: LEMMA
    system |- G(FORALL (i: IDENTITY):
                  pc[i] = waiting => time_out[i] - time <= delta1);
```

This lemma can be proved by induction at depth $k = 1$.

```
sal-inf-bmc -v 10 -d 1 -i fischer time_aux2
...
proved.
total execution time: 0.69 secs
```

Then, the mutual-exclusion property is provable as follows:

```
sal-inf-bmc -v 3 -d 9 -i -l time_aux2 fischer mutual_exclusion
...
proved.
total execution time: 4.16 secs
```

This uses k-induction at depth 9, with `time_aux2` as an auxiliary invariant. This is the smallest depth at which the proof by k-induction succeeds.

This proof works only for Fischer's protocol with two processes. In a system with at least three processes, the auxiliary invariant `time_aux2` is not sufficient for k-induction to succeed. A few attempts with relatively small k (say $k = 6$) show that there is a counterexample to k-induction that starts from a state S_0 that satisfies the following constraints:

```
pc[1]=waiting, pc[2]=critical, pc[3]=sleeping, lock /= 0,
  time_out[3] < time_out[1], time_out[3] < time_out[2].
```

A little analysis shows that this trajectory can be extended into a counterexample to k-induction for arbitrary large length k by letting process 3 perform a sequence of `try_again_later` transitions. Transition `try_again_later` does not change any of the discrete state variables, and increments `time_out[3]` by a positive amount ϵ, which can be arbitrarily small. From state S_0, a `try_again_later` transition by process 3, followed by a time-progress transition of length ϵ leads to a state S_0' which is the same as S_0, except that `time_out[3]` has increased by ϵ. If ϵ is sufficiently small, all transitions enabled in S_0 remain enabled in S_0': the two transitions that lead from S_0 to S_0' are essentially the same as an idle step from S_0. By adding as many of these steps as necessary, a counterexample to k-induction can be constructed for arbitrary large k.

A More Scalable Proof

We now describe a proof of the mutual-exclusion property that works for an arbitrary number of processes. The proof consists of a sequence of lemmas that all describe invariants of the system and are all proved by induction at depth 1 (or 0). Not requiring induction at higher depths avoids the difficulties caused by the `try_again_later` transition discussed previously.

The sequence of invariants that lead to the proof can be found by reasoning backward. Our goal is to prove the property `mutual_exclusion`:

```
mutual_exclusion: THEOREM
   system |- G(FORALL (i, j: IDENTITY):
                i /= j AND pc[i] = critical => pc[j] /= critical);
```

A first step is to replace this property by the following stronger invariant:

```
mutex: THEOREM
   system |- G(FORALL (i: IDENTITY): pc[i] = critical => lock = i);
```

This is easily seen to imply mutual_exclusion. Property mutex is not inductive, but it can be strengthened further into the following inductive lemma:

```
logical_aux1: LEMMA
   system |- G(FORALL (i, j: IDENTITY):
                 pc[i] = critical => lock = i AND pc[j] /= waiting);
```

Proving logical_aux1 requires two auxiliary lemmas:

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

time_aux3: LEMMA
   system |- G(FORALL (i, j: IDENTITY):
       lock = i AND pc[j] = waiting => time_out[i] > time_out[j]);
```

Lemma time_aux1 is a property that is true by construction for all timeout automata and it is provable by induction at depth 1. The last step is to find a proof of time_aux3, but this lemma is provable using property time_aux2 encountered previously.

Overall, the mutual-exclusion property is proved by a sequence of calls to sal-inf-bmc as follows. First, we prove time_aux1 and time_aux2 by induction:

```
sal-inf-bmc -v 3 -i -d 1 fischer time_aux1
...
proved.
total execution time: 0.94 secs

sal-inf-bmc -v 3 -i -d 1 fischer time_aux2
...
proved
total execution time: 0.96 secs
```

Then, we prove time_aux3 using time_aux2 as a lemma:

```
sal-inf-bmc -v 3 -i -d 1 -l time_aux2 fischer time_aux3
...
proved.
total execution time: 1.18 secs
```

Now, logical_aux1 is provable using time_aux1 and time_aux3:

```
sal-inf-bmc -v 3 -i -d 1 -l time_aux3 -l time_aux1 fischer logical_aux1
...
proved.
total execution time: 1.11 secs
```

In the last step, we show that logical_aux1 implies mutual_exclusion. The proof is done by induction at depth 0, which amounts to showing that invariant logical_aux1 is stronger than mutual_exclusion:

```
sal-inf-bmc -v 3 -i -d 0 -l logical_aux1 fischer mutual_exclusion
...
proved.
total execution time: 0.67 secs
```

3.2.3 Performance

Tables 3.1 and 3.2 show the CPU time for verifying the mutual exclusion property using SAL 2.0 with two of the solvers supported by SAL, namely, ICS 2.0 and UCLID. The runtimes are given in seconds, and were measured for a number of processes N ranging from 2 to 20. The results were measured on a desktop PC with a Pentium 4 CPU, a clock speed of 2 GHz, and 1 GByte of RAM. The PC was running the Linux operating system (Kernel 2.4.18). The time reported in the table was the total proof time as displayed by `sal-inf-bmc`. Since all the proofs are by induction, the total proof time includes two calls to the solver — one for the base case and one for the induction step — plus the processing time used by `sal-inf-bmc` itself. The tests were performed with each call to ICS or UCLID limited by a timeout of 1000 s (16 min 40 s) and a memory limit of 1 GByte. The middle columns in Tables 3.1 and 3.2 correspond to the following five proof commands:

ta1: `sal-inf-bmc -v 10 -d 1 -i -s ics fischer time_aux1`

ta1: `sal-inf-bmc -v 10 -d 1 -i -s <solver> fischer time_aux2`

ta3: `sal-inf-bmc -v 10 -d 1 -i -s <solver> -l time_aux2 fischer time_aux3`

la1: `sal-inf-bmc -v 10 -d 1 -i -s <solver> -l time_aux3 -l time_aux1 fischer logical_aux1`

me: `sal-inf-bmc -v 10 -i -d 0 -s <solver> -l logical_aux1 fischer mutual_exclusion`

where `<solver>` is either `ics` or `uclid`. Tables 3.3 and 3.4 report the results of similar experiments using the `cvc` and `svc` solvers, respectively. Failure of a solver is reported as either "time" (timeout) or "memory" (memory exhaustion) in the tables.

The following table indicates the solver versions that were used in this experiment. We used the latest versions of each tool, as available in December 2003:

Solver	Version	URL
ICS	ICS 2.0 beta	`http://www.icansolve.com/`
UCLID + zchaff	UCLID 1.0 alpha	`http://www-2.cs.cmu.edu/~uclid/`
	zchaff.2001.2.17	`http://ee.princeton.edu/~chaff/zchaff.html`
CVC	CVC 1.0a	`http://verify.stanford.edu/CVC/`
SVC	SVC 1.11	`http://verify.stanford.edu/SVC/`

As can be seen, the performance of the bounded model checker varies significantly depending on the solver used. On this SAL formalization of Fischer's protocol, both UCLID and ICS do much better than CVS or SCV. With ICS or UCLID, one can prove mutual exclusion for as many as 13 or 10 processes, respectively. CVC or SVC cannot do better than 6 and 4 processes, respectively. It is also notable that ICS and UCLID do not fail on the same properties. ICS fails to prove property `time_aux1` for $N \geqslant 14$, while UCLID can prove it for all N between 2 and 20. Conversely, lemma `time_aux3` is proved by ICS for N between 2 and 19, while UCLID runs out of time for $N \geqslant 11$. By selecting the best solver for each property, one can then prove the protocol correct for a system with as many as 19 processes.

All these numbers must be taken with caution. They reflect the performance of `sal-inf-bmc` on the *specific formalization* of Fischer's protocol given in Appendix E. As is well known, the performance of a satisfiability solver is very sensitive to minor variations in a problem presentation. For example, different variable orderings can make a huge difference. It has been our experience that small syntactic changes in the SAL specifications can cause substantial differences in proof times. The next section shows how performance can be considerably improved by rewriting parts of SAL model.

N	ta1	ta2	ta3	la1	me	total
2	0.85	0.84	0.92	0.91	0.63	4.15
3	1.62	1.05	1.25	1.16	0.80	5.88
4	1.91	1.35	1.75	1.55	0.94	7.50
5	1.96	1.77	2.41	1.86	1.08	9.08
6	3.37	2.02	3.40	2.25	1.31	12.35
7	6.83	3.02	4.22	2.75	1.52	18.34
8	11.78	3.77	5.09	3.23	1.78	25.65
9	26.94	6.08	4.86	3.91	2.09	43.88
10	60.68	8.77	5.73	4.40	2.31	81.89
11	161.93	18.05	6.08	5.01	2.52	193.59
12	420.56	21.43	6.49	5.32	2.59	456.39
13	784.17	22.86	7.01	6.10	2.97	823.11
14	time	62.67	11.87	7.03	3.30	N/A
15	time	63.30	10.24	8.33	3.64	N/A
16	time	64.63	14.17	9.58	4.01	N/A
17	time	65.01	14.95	10.55	4.45	N/A
18	time	64.57	74.73	11.49	5.01	N/A
19	time	283.12	433.25	12.41	5.16	N/A
20	time	221.07	time	12.92	5.68	N/A

Table 3.1: Proof Times Using ICS 2.0

N	ta1	ta2	ta3	la1	me	total
2	1.32	1.36	1.40	1.31	0.52	5.91
3	1.90	1.98	3.22	1.82	0.66	9.58
4	2.39	3.80	13.62	2.61	0.85	23.27
5	15.30	5.97	61.07	5.49	1.05	88.88
6	16.32	9.18	74.62	7.00	1.29	108.41
7	33.06	19.86	204.80	7.74	1.56	267.02
8	13.00	29.15	317.22	17.71	1.85	378.93
9	14.00	30.23	410.88	22.87	2.17	480.15
10	16.91	49.99	614.08	34.09	2.30	717.37
11	60.80	332.94	time	61.91	2.63	N/A
12	70.92	255.22	time	65.15	2.81	N/A
13	88.53	819.70	time	140.97	3.14	N/A
14	132.14	839.70	time	199.72	3.53	N/A
15	147.71	706.21	time	207.20	4.08	N/A
16	204.60	603.11	time	247.12	4.49	N/A
17	125.19	765.99	time	326.94	4.97	N/A
18	150.41	611.01	time	454.17	5.55	N/A
19	157.93	579.01	time	401.45	5.73	N/A
20	328.75	376.96	time	478.97	6.37	N/A

Table 3.2: Proof Times Using UCLID

N	ta1	ta2	ta3	la1	me	total
2	0.66	0.73	0.82	0.72	0.41	3.34
3	1.04	1.52	1.61	1.03	0.55	5.75
4	1.75	5.17	3.69	1.50	0.71	12.82
5	3.29	26.26	8.49	1.95	0.85	40.84
6	7.17	163.02	18.47	2.54	1.06	192.26
7	15.72	time	58.03	3.29	1.29	N/A
8	33.85	memory	209.71	4.16	1.55	N/A
9	77.82	memory	775.02	5.17	1.84	N/A
10	181.98	memory	time	6.19	2.06	N/A
11	434.19	memory	memory	7.57	2.29	N/A
12	1185.71	memory	memory	8.79	2.40	N/A

Table 3.3: Proof Times Using CVC

N	ta1	ta2	ta3	la1	me	total
2	1.39	1.40	0.71	0.64	0.36	4.50
3	13.20	21.92	2.49	1.00	0.47	39.08
4	113.21	527.04	39.98	1.83	0.65	682.71
5	time	time	611.46	3.63	0.80	N/A
6	time	time	time	8.71	0.99	N/A
7	time	time	time	22.47	1.22	N/A
8	time	time	time	62.36	1.48	N/A
9	time	time	time	176.70	1.79	N/A
10	time	time	time	483.50	2.11	N/A

Table 3.4: Proof Times Using SVC

3.2.4 Variant Specifications

As can be seen in Table 3.1, lemma `time_aux1` is the bottleneck in the proof with ICS. A straightforward modification of the SAL specifications removes this bottleneck. With the revised specifications, much larger instances of the protocol, with as many as 53 processes, can be verified. We then examine a more general version of the protocol, where the two parameters `delta1` and `delta2` are uninterpreted.

Removing the Recursive Definition

Lemma `time_aux1` states a basic invariant that is true for all timeout automata, namely, the fact that `time <= time_out[i]` for all `i`. Proving this property by induction should be straightforward: it follows immediately from the definition of `process`, and the fact that the minimum of the `time_out` array is smaller than or equal to any of its elements. However, the latter fact is expensive to establish by ICS because of the recursive definition of `min` that we have employed so far:

```
recur_min(x: TIMEOUT_ARRAY, min_sofar: TIME, idx: [0 .. N]): TIME =
  IF idx = 0 THEN min_sofar
     ELSE recur_min(x, min(min_sofar, x[idx]), idx-1)
  ENDIF;

min(x: TIMEOUT_ARRAY): TIME = recur_min(x, x[N], N-1);
```

For any fixed N, SAL unfolds this definition into a cascade of if-then-elses. For example, for $N = 3$, `min(x)` expands to (something equivalent to) the following expression.

```
   LET min_sofar1 = x[3] IN
     LET min_sofar2 = IF min_sofar1 < x[2] THEN min_sofar1 ELSE x[2] ENDIF
       IN IF min_sofar2 < x[1] THEN min_sofar2 ELSE x[1] ENDIF
```

This expansion results in an expression with $N-1$ cascading conditionals. Furthermore, for all the solvers supported by SAL, including ICS and UCLID, proving inequalities such as `min(x) <= x[2]` from this expression requires an exponential number of case splits.

The following trick allows one to eliminate inefficient recursive definitions from the SAL specifications. First, we define a predicate `is_min(x, t)` that is true if and only if `t = min(x)`.

```
  is_min(x: TIMEOUT_ARRAY, t: TIME): bool =
              (FORALL (i: IDENTITY): t <= x[i]) AND
              (EXISTS (i: IDENTITY): t = x[i]);
```

This definition makes explicit all the properties of the minimum. Then, we rewrite the `clock` module as follows:

```
clock: MODULE =
  BEGIN
    ...
  TRANSITION
    [ time_elapses:
         (FORALL (i: IDENTITY): time < time_out[i]) -->
                        time' IN { t: TIME | is_min(time_out, t) }
    ]
  END;
```

This is clearly equivalent to the previous specifications: the transition is enabled whenever `time` is strictly smaller than all the timeouts, and it advances `time` to the smallest element in the `time_out` array.

Removing the recursive definition of `min` has a dramatic impact on performance. The proof of lemma `time_aux1` is now the cheapest of the five verification steps. The most expensive step by far is now proving lemma `time_aux3`. We found that another trick can accelerate the proof of `time_aux3`: although `time_aux3` is provable from `time_aux2` alone, adding the following lemma reduces proof time significantly.

```
time_aux0: LEMMA
  system |- G(time >= 0 AND FORALL (i: IDENTITY): time_out[i] > 0);
```

The full verification includes then a new step, namely, the proof of `time_aux0` by induction:

```
ta0: sal-inf-bmc -v 10 -d 1 -i -s <solver> fischer time_aux0
```

The proof of `time_aux3` is now as follows:

```
ta3: sal-inf-bmc -v 10 -d 1 -i -s <solver> -l time_aux2 -l time_aux0
     fischer time_aux3
```

Table 3.5 shows the proof times obtained with ICS for the revised SAL model. The numbers were measured on the same machine, with the same timeout and memory limit as previously, but with a more recent SAL version (SAL 2.3). The proof succeeds for all N between 2 and 43, and for some values of N above 44. The largest example that could be successfully verified with ICS included $N = 53$ processes.

Removing the inefficient recursive definition also reduces proof times when solvers other than ICS are used, but the improvement is not as dramatic. The largest examples we could verify with UCLID, CVC, and SVC contained 12, 6, and 4 processes, respectively.

Uninterpreted Constants

So far, all the SAL models of Fischer's protocol we have discussed gave explicit values to the two protocol parameters `delta1` and `delta2`. It is also possible to analyze the protocol without assigning specific values to these parameters. We just need to specify that the two parameters satisfy the constraint `delta1 < delta2`. This can be done by relying on the SAL type system, and declaring the two constants as follows:

```
  delta1: {x: REAL | 0 < x};

  delta2: {x : REAL | delta1 < x};
```

Thus, `delta1` is an arbitrary positive real and `delta2` is an arbitrary real larger than `delta1`.

With this modification, the mutual exclusion property can be proved exactly as before. The performance obtained with ICS is analoguous to the verification with explicit constants. All instances with $N \leqslant 45$ can be verified, and the largest verified example has size $N = 54$. The proof can also be done with CVC or SVC for small examples. UCLID cannot be used, as the theory decided by UCLID — the so-called *separation logic* — is not expressive enough for dealing with uninterpreted `delta1` and `delta2`. For example, lemma `time_aux2` is not in the separation logic if `delta1` is uninterpreted.

N	ta0	ta1	ta2	ta3	la1	me	total
2	0.60	0.57	0.61	0.68	0.69	0.44	3.59
3	0.76	0.74	0.77	1.04	0.98	0.56	4.85
4	0.95	0.94	0.99	1.54	1.29	0.72	6.43
5	1.17	1.15	1.20	2.07	1.70	0.87	8.16
6	1.36	1.36	1.40	3.08	2.06	1.03	10.29
7	1.52	1.52	1.63	2.75	2.49	1.23	11.14
8	1.83	1.85	1.97	4.10	2.99	1.52	14.26
9	2.00	1.96	2.20	3.94	3.74	1.75	15.59
10	2.30	2.31	2.43	9.29	4.51	2.03	22.87
11	2.44	2.48	2.83	8.04	5.76	2.37	23.92
12	2.75	2.76	3.21	6.60	5.26	2.70	23.28
13	3.00	3.09	3.59	10.83	6.28	3.01	29.80
14	3.19	3.20	3.78	9.35	6.87	3.03	29.42
15	3.53	3.54	4.29	11.03	8.00	3.42	33.81
16	3.82	3.74	4.60	16.00	8.82	3.68	40.60
17	4.20	4.32	5.20	17.82	10.12	4.15	45.81
18	4.55	4.63	5.53	20.62	11.58	4.66	51.57
19	4.77	4.94	6.14	20.68	12.63	5.18	54.34
20	5.13	5.05	6.61	18.14	13.12	5.72	53.77
21	5.28	5.39	6.90	74.31	15.32	6.19	113.39
22	5.81	5.92	7.76	27.84	16.35	6.73	70.41
23	6.23	6.34	8.40	279.63	18.66	6.64	325.90
24	6.66	6.76	9.18	44.88	20.60	7.98	96.06
25	7.17	6.89	9.74	88.71	20.39	7.86	140.76
26	7.68	7.38	10.33	50.27	22.33	8.31	106.30
27	7.77	8.03	11.31	126.88	24.20	9.16	187.35
28	8.20	8.37	12.02	119.04	25.64	9.77	183.04
29	8.73	8.75	12.61	111.68	28.49	10.79	181.05
30	9.35	9.54	14.20	252.60	29.14	11.76	326.59
31	9.49	10.16	15.29	56.38	36.49	12.29	140.10
32	10.15	10.82	15.85	120.48	35.32	12.85	205.47
33	11.24	11.05	17.04	141.23	37.03	14.02	231.61
34	10.89	11.85	18.10	591.98	40.96	13.22	687.00
35	11.52	12.22	19.64	199.78	47.40	14.34	304.90
36	12.18	12.54	20.56	444.16	46.59	15.24	551.27
37	12.54	13.41	21.78	586.76	56.08	16.67	707.24
38	13.80	14.16	23.02	401.76	57.94	17.30	527.98
39	13.56	13.61	24.38	738.99	66.67	18.24	875.45
40	14.15	14.42	26.24	346.09	60.29	19.69	480.88
41	14.69	15.07	27.61	391.54	65.05	21.03	534.99
42	15.72	16.14	29.34	288.12	67.82	20.83	437.97
43	16.02	16.76	31.69	206.23	81.03	22.10	373.83
44	16.38	17.49	32.98	time	89.77	21.16	N/A
45	17.26	17.68	36.84	623.13	78.31	22.26	795.48
46	18.75	19.12	36.64	967.27	95.39	23.71	1160.88
47	17.32	17.48	37.23	time	112.34	24.93	N/A
48	19.88	18.39	43.23	time	133.98	26.14	N/A
49	18.66	20.48	41.96	time	102.78	27.37	N/A
50	19.46	21.34	44.19	927.81	104.09	29.23	1146.12
51	20.25	21.04	46.94	time	132.29	29.85	N/A
52	21.73	22.13	50.98	time	115.19	31.98	N/A
53	22.25	23.13	53.21	689.94	137.64	32.63	958.80
54	23.38	23.72	61.39	time	187.79	33.18	N/A

Table 3.5: Proof Times Using ICS 2.0, Revised Specifications

3.3 The Train-Gate-Controller Revisited

The SAL specification for the train-gate-controller example from Section 2 is a direct translation from the timed automata model. We now examine a timeout-automata model of the same example. This illustrates a slight generalization that can be useful when components synchronize on simultaneous actions.

The components of the train-gate-controller communicate via action *synchronization*. Edges labeled with the same event correspond to actions that must happen simultaneously. This was modeled in SAL by using a global type `actions`, that comprises all the actions of the system. To model this form of synchronization using timeout automata, we use a message-passing approach, where messages are communicated with zero delay. For example, the `train` sends the `controller` the signal `approach`. Upon receiving this signal the `controller` sends another signal to `lower` the `gate`. The type `SIGNAL` contains all the signals necessary for the communication between the three components.

```
SIGNAL: TYPE = {approach, exit, lower, raise};
```

As in the model we used for Fischer's protocol, a global variable `time` keeps the current time and the variables `time_out[i]` contain the time when the next discrete transition of the `i`th component is scheduled to occur. Here, the index `i` is used to enumerate the three components, that is, `i=1` represents the `train` component, `i=2` the `gate`, and finally `i=3` the `controller`.

The `clock` module is responsible for performing the time-progress transitions, by advancing time up to the next timeout when a discrete transition is due. This information is provided as input from the `train`, the `gate`, and the `controller`. The function `min` computes the new value of `time` as minimum of `time_out[1]`, `time_out[2]`, `time_out[3]`. This value is then communicated to every component, through the output variable `time`. In the case of the train gate controller, the clock module is an extension of the basic clock structure as depicted in Figure 3.3, with two additional global Boolean variables, `flag1` and `flag2`. These variables are used to coordinate the time-progress steps; after a component has sent a signal, time is not allowed to elapse until the signal has been received. This models communication via instantaneous message passing. The values of the two flags are maintained by the three components.

```
tgc: CONTEXT =
BEGIN
  SIGNAL: TYPE = {approach, exit, lower, raise};
  TIME: TYPE = REAL;
  N: NATURAL = 3;
  INDEX: TYPE = [1..N];
  TIMEOUT_ARRAY: TYPE = ARRAY INDEX OF TIME;
  ...
  clock: MODULE =
  BEGIN
    INPUT time_out: TIMEOUT_ARRAY
    GLOBAL flag1, flag2: BOOLEAN
    OUTPUT time: TIME
  INITIALIZATION
    time = 0
  TRANSITION
    [ time_elapses:
         time < min(time_out) AND (NOT flag1) AND (NOT flag2)
            --> time' = min(time_out) ]
  END;
  ...
END
```

The input variable time is the output variable of clock. This variable controls the current time. The train has two output variables, timeout contains the time when the next discrete transition is enabled, while msg1 is the message signaled to the controller.

```
train: MODULE =
BEGIN
  INPUT
    time: TIME
  OUTPUT
    timeout: TIME,
    msg1: SIGNAL,
  GLOBAL flag1: BOOLEAN
  LOCAL
    t_state: T_STATE
  INITIALIZATION
    t_state = t0;
    timeout IN { x: TIME | time < x };
    flag1 = FALSE
```

The discrete transition of the train automaton are specified as follows. One must ensure that every step is performed at the corresponding time point, which is guaranteed by the constraint time = timeout associated with every transition. For example, in the transition from t0 to t1 the train outputs the signal approach (msg1' = approach), sets the Boolean variable flag1 to true (flag1' = TRUE), and the new value of timeout determines when the next discrete step will be taken. The timeout is updated non-deterministically to any time in the interval (time + 2, time + 5]. This reflects the constraints imposed by the invariant of location t1 and the guard of the edge from t1 to t2 in the timed automata model of Figure 2.1.

```
TRANSITION
    [ t0_t1:
      t_state = t0 AND time = timeout -->
        t_state' = t1;
        msg1' = approach;
        flag1' = TRUE;
        timeout' IN {x: TIME | time + 2 < x AND x <= time + 5}
```

The controller has msg1 as an input variable, over which it receives signals from the train, and msg2 as an output, with which it sends signals to the gate.

```
controller : MODULE =
BEGIN
  INPUT
    time: TIME,
    msg1: SIGNAL,
  OUTPUT
    timeout: TIME,
    msg2: SIGNAL
  GLOBAL flag1, flag2: BOOLEAN
```

Upon receiving the signal approach from the train (msg1 = approach and flag1 is true), the controller moves to location c1, sets flag1 to false, and increases its timeout variable. On the transition from c1 to c2 the controller sends the lower signal to the gate (msg2' = lower) and sets flag2 to true.

```
TRANSITION
  [ c0_c1:
      c_state = c0 AND msg1 = approach AND flag1 -->
        c_state' = c1;
        flag1' = FALSE;
        timeout' = time + 1
 [] c1_c2:
      c_state = c1 AND time = timeout -->
        c_state' = c2;
        msg2' = lower;
        flag2' = TRUE;
        timeout' IN {x: TIME | time < x}
```

The gate changes `flag2` to false after receiving the `msg2 = lower` signal.

```
TRANSITION
    [ g0_g1:
        g_state = g0 AND msg2 = lower AND flag2 -->
          g_state' = g1;
          flag2' = FALSE;
          timeout' IN {x: TIME | time < x AND x <= time + 1}
```

The three components are composed asynchronously (i.e., using the `[]` operator in SAL). The `WITH` construction introduces a new state variable `time_out` as an array for the output state variables `timeout` used in each component. The `RENAME` construction picks out the member of the `time_out` array to be wired up to the corresponding component.

```
tgc_module: MODULE =
  WITH OUTPUT time_out: TIMEOUT_ARRAY
       (RENAME timeout TO time_out[1] IN train)
      []
       (RENAME timeout TO time_out[2] IN gate)
      []
       (RENAME timeout TO time_out[3] IN controller);
```

Finally, the `tgc_module` is composed asynchronously with the `clock`.

```
system: MODULE = clock [] tgc_module;
```

The safety property of the train gate controller `safe` is specified as in Section 2, and can be proved by *k*-induction at depth 14 in 46.15 seconds. Other interesting properties can be specified that describe the timing behavior of the train gate controller system.

```
time_aux1: LEMMA system |- G(FORALL (i:INDEX): time <= time_out[i]);
time_aux2: LEMMA system |- G(t_state = t2 => time_out[1] - time <= 5);
time_aux3: LEMMA system |- G((t_state = t1 AND g_state = g1) =>
                                  time_out[1] > time_out[2]);
```

Chapter 4

Calendar Automata

Timeouts are convenient for application like Fischer's protocol, where n processes $p_1, \ldots, p_n$ 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 with transmission delays, we add *event calendars* to our transition systems.

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

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

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 $\sigma(t) \leqslant \min(\sigma(T))$ and $\sigma(t) \leqslant \min(\sigma(c))$.
- 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 are enabled in states where $\sigma(t) = \min(\sigma(T))$ or $\sigma(t) = \min(\sigma(c))$ and must satisfy the following requirements:
  - $\sigma(t) = \sigma'(t)$
  - for all $y \in T$ we have $\sigma'(y) = \sigma(y)$ or $\sigma'(y) > \sigma'(t)$
  - if $\sigma(t) = \min(\sigma(c))$ then $\mathrm{Ev}_{\sigma'(t)}(\sigma'(c)) \subseteq \mathrm{Ev}_{\sigma(t)}(\sigma(c))$
  - there is $x \in T$ such that $\sigma(x) = \sigma(t)$ and $\sigma'(x) > \sigma'(t)$, or we have $\mathrm{Ev}_{\sigma'(t)}(\sigma'(c)) \subset \mathrm{Ev}_{\sigma(t)}(\sigma(c))$.

These constraints ensure that $\sigma(t) \leqslant \min(\sigma(T))$ and $\sigma(t) \leqslant \min(\sigma(c))$ are invariants: timeout values and the occurrence time of any event in the calendar are never in the past. Discrete transitions are enabled when the current time reaches the value of a timeout or the occurrence time of a scheduled event. The

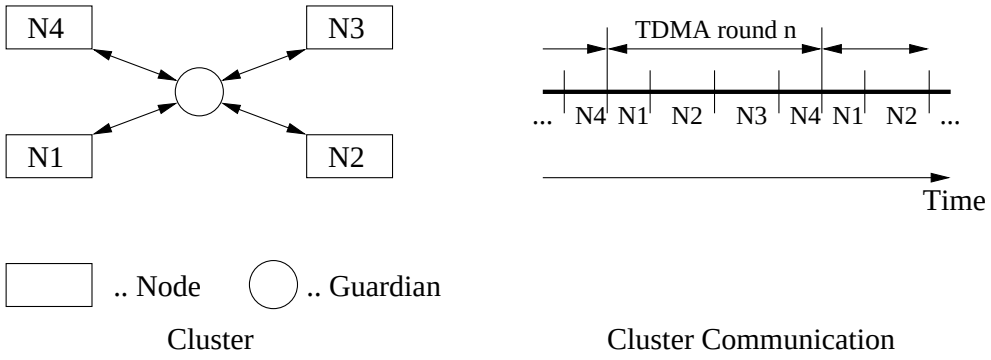


Figure 4.1: TTA cluster and TDMA schedule.

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.

4.1 The TTA Startup Protocol

The remainder of this report describes an application of the preceding modeling principles to the TTA fault-tolerant startup protocol [SRSP04]. 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 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 Figure 4.1. 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

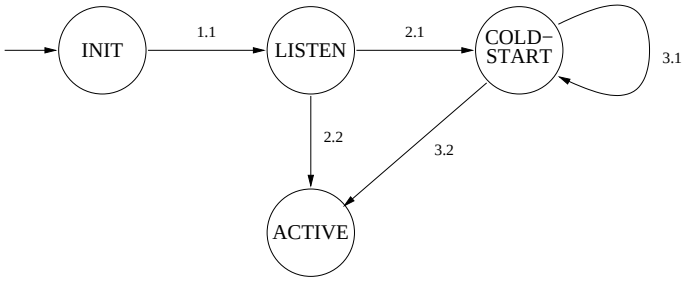


Figure 4.2: State-machine of the TTA node startup algorithm

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 [SRSP04].

The startup algorithm executed by the nodes is described schematically in Figure 4.2. When a node *i* is powered on, it performs some internal initializations in the INIT state, 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 $\tau_i^{coldstart}$, 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 $\tau_i^{coldstart}$. These timeouts are defined as follows:

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

where τ^{round} is the round duration and $\tau_i^{startup}$ is the start of *i*'s slot in a TDMA cycle. Nodes are indexed from 1 to N. For a fixed slot time τ we then have $\tau_i^{startup} = (i-1).\tau$ and $\tau^{round} = N.\tau$.

4.2 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.

```
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;
```

Figure 4.3: Calendar Encoding for the Simplified Startup Protocol

4.2.1 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 Figure 4.3.

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 Figure 4.3. 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'.

4.2.2 Nodes

Figure 4.4 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 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)
     ...
```

Figure 4.4: Node Specification

4.2.3 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 Section 4.

Because time cannot advance beyond the calendar's delivery time, the full model ensures that all pending messages are received. For example, transition cs_frame_in_coldstart of Figure 4.4 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 non-deterministic 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.

4.3 Protocol Verification

4.3.1 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.

4.3.2 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));
```

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

```
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: 80.35 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.

4.3.3 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 [Rus00]. Given a transition system $\mathcal{M} = \langle S, I, \rightarrow \rangle$, this method amounts to constructing an abstraction of $\mathcal{M}$ (or verification diagram [MP94]) based on n state predicates $A_1(\sigma), \ldots, A_n(\sigma)$. The abstraction is a transition system $\mathcal{M}_0 = \langle S_0, I_0, \rightarrow_0 \rangle$ with state space $S_0 = \{a_1, \ldots, a_n\}$. 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:

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

  where $a_{j_1}, \ldots, a_{j_k}$ 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 ... or A_{j_k}". It also states that some of the predicates $A_1, \ldots, A_n$ are true in all the initial states of $\mathcal{M}$. If the abstraction is correct, then clearly the disjunction $A_1 \vee \ldots \vee A_n$ 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 $A_1, \ldots, A_n$ 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.

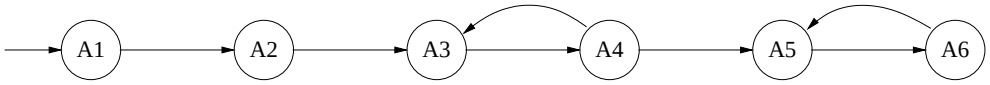
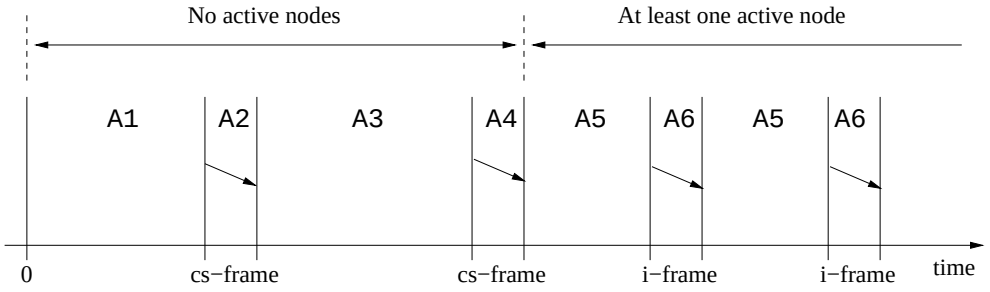


Figure 4.5: Verification Diagram for the Simplified Startup

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 $\tau_i^{coldstart}$, 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)
             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 4.5 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 Figure 4.5 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 no 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 Figure 4.5: 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 state /= 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 module, the abstractor, and the monitor.
- We show that this composition satisfies the invariant property G(state /= bad), by induction using sal-inf-bmc.
- Finally, using sal-inf-bmc again, we show that the previous invariant implies the correctness property synchro.

4.3.4 Results

	Simplified Startup				Fault-Tolerant Startup			
N	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 4.1: Verification Times

Table 4.1 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 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 before. 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.

[2]The full specifications are available at `http://www.sdl.sri.com/users/bruno/sal/`. The simplified model is given in Appendix F.

Chapter 5

Conclusion

We have described different methods for specifying and analyzing timed systems in SAL. In particular, 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 us to verify a non-trivial 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 [SRSP04] 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 [YTK01] and [BFG02] by vastly increasing the number of scenarios considered. It achieved sufficient performance to support design exploration as well as verification.

Lönn and Pettersson [LP97] consider startup algorithms for TDMA systems similar to TTA, and verify one of them using UPPAAL [LPY97]. 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 nodes 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 here 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.

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *5th Symp. on Logic in Computer Science (LICS 90)*, pages 414–425, 1990.

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

[AH93] R. Alur and T. A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104(1):35–77, May 1993.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[BFG02] C. Bernardeschi, A. Fantechi, and St. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability*, 12:251–275, December 2002.

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

[CHR91] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. Tool presentation: SAL 2. To be presented at CAV 2004, 2004.

[dMOS03] Leonardo de Moura, Sam Owre, and Natarajan Shankar. The SAL Language Manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, 2003. Available at `http://sal.csl.sri.com/documentation.html`.

[dMR02a] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. *Annals of Mathematics and Artificial Intelligence*, 2002.

[dMR02b] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*. Cincinnati, Ohio, 2002.

[dMRS02] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In Andrei Voronkov, editor, *18th Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, July 27-30 2002.

[dMRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Andrei Voronkov, editor, *Computer-Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer-Verlag, 2003.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *Lecture Notes in Computer Science*, 1066:208–219, 1996.

[DS95] J. Davies and S. Schneider. A Brief History of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, February 1995.

[FMS88] J. Farnam, A. Mok, and D. Stuart. Formal Specification of Real-Time Systems. Technical Report UTCS-TR-88-25, Department of Computer Science, University of Texas at Austin, 1988. Available at `http://www.cs.utexas.edu/users/cpg/RTS/pubs.html`.

[For03] Formal Methods Program. Formal methods roadmap: Pvs, ics, and sal. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003. Available at `http://fm.csl.sri.com/doc/roadmap03`.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254:460–463, 1997.

[HMP94] T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal Proof Methodologies for Timed Transition Systems. *Information and Computation*, 112(2):273–337, 1994.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.

[JM94] F. Jahanian and A. Mok. Modechart: A Specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[KPSY93] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration Graphs: A class of decidable hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 179–208. Springer-Verlag, 1993.

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

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

[LV91] N. Lynch and F. Vaandrager. Forward and Backward Simulations for Timing-Based Systems. In *Proceedings of REX Workshop. Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446, Mook, The Netherlands, June 1991. Springer-Verlag.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. Timed-Constrained Automata. In *CONCUR'91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Amsterdam, The Netherlands, August 1991. Springer-Verlag.

[MP94] Z. Manna and A. Pnueli. Temporal Verification Diagrams. In *International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765, Sendai, Japan, April 1994. Springer-Verlag.

[MT90] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proceedings of CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 410–415, Amsterdam, The Netherlands, August 1990. Springer-Verlag.

[NS94] X. Nicollin and J. Sifakis. The Algebra of Timed Processes, ATP: Theory and application. *Information and Computation*, 114(1):131–178, October 1994.

[Pet81] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[Rus00] J. Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In *Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.

[Sor01] Maria Sorea. Tempo: A model-checker for event-recording automata. In *Proceedings of RT-TOOLS'01*, Aalborg, Denmark, August 2001. Also available as Technical Report SRI-CSL-01-04, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.

[Sor02] Maria Sorea. Bounded model checking for timed automata. In *Proceedings of the Third Workshop on Models for Time-Critical Systems (MTCS 2002)*, volume 68 of *Electronic Notes in Theoretical Computer Science*, 2002. `http://www.elsevier.com/locate/entcs/volume68.html`.

[SRSP04] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. To be presented at DSN 2004, 2004.

[YTK01] 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, December 2001.

Appendix A

An Overview of SAL

We give a brief overview of the main features of the SAL language and verification tools. More complete descriptions can be found in [dMOS03] and [dMOR+04]. The SAL system and documentation can be downloaded at `http://sal.csl.sri.com`.

A.1 Specification Language

In SAL, specifications are organized in modules called *contexts*. Figure A.1 shows the context `peterson` that describes Peterson's mutual exclusion algorithm [Pet81]. The specification is contained in the file `peterson.sal`; the names of the file and the context must match. The `peterson` example illustrates the syntax of the SAL language and the general organization of typical SAL contexts. First, a context introduces a number of types, constants, and possibly function definitions. Then it defines one or more state-transition systems called *modules*. These definitions are followed by a set of lemmas or theorems about the modules.

In Figure A.1, the context starts by defining an enumerated type `PC`, which consists of the three elements `sleeping`, `trying`, and `critical`. This definition is followed by the description of a base module called `process`. Another module called `system` is then constructed as the asynchronous composition of two instances of `process`. The rest of the context lists a number of theorems about `system`.

Base Modules and Guarded Commands

The description of a base module includes the module's variables and parameters, an *initialization* section that specifies initial values of these variables, and a *transition* section that defines the module's state-transition relation. In most examples, the transition relation of a base module is defined using guarded commands.

A base module's variables and their types define the state space of the module. For example, the state space of module `process` is the set of all tuples of the form $\langle x_1, x_2, pc_1, pc_2 \rangle$, where x_1 and x_2 are booleans, and pc_1 and pc_2 are values of type `PC`. In SAL, a module M can have four disjoint sets of variables:

- *input variables* can be observed but not modified by M;
- *output variables* can be modified by M and can be observed but not modified by other modules;

```
peterson: CONTEXT =
BEGIN
  PC: TYPE = {sleeping, trying, critical};

  process[tval : BOOLEAN]: MODULE =
    BEGIN
      INPUT pc2 : PC, x2 : BOOLEAN
      OUTPUT pc1 : PC, x1 : BOOLEAN
      INITIALIZATION
         pc1 = sleeping
      TRANSITION
        [ wakening:
           pc1 = sleeping --> pc1' = trying; x1' = (x2 = tval)

       [] entering_critical:
           pc1 = trying AND (pc2 = sleeping OR x1 = (x2 /= tval))
           --> pc1' = critical

       [] leaving_critical:
           pc1 = critical --> pc1' = sleeping; x1' = (x2 = tval)
        ]
    END;

  system: MODULE =
     process[FALSE]
     []
     RENAME pc2 TO pc1, pc1 TO pc2,
            x2 TO x1, x1 TO x2 IN process[TRUE];

  mutex: THEOREM system |- G(NOT(pc1 = critical AND pc2 = critical));

  invalid: THEOREM system |- G(NOT(pc1 = trying AND pc2 = critical));

  livenessbug1: THEOREM system |- G(F(pc1 = critical));

  livenessbug2: THEOREM system |- G(F(pc2 = critical));

  liveness1: THEOREM system |- G(pc2 = trying => F(pc2 = critical));

  ...

END
```

Figure A.1: Example SAL Specification

- *local variables* can be modified by *M* but are not visible to other modules;
- *global variables* are shared between modules; they can be observed and modified by several modules.

For example, module `process` can read but not update the two input variables `pc2` and `x2`. These variables are controlled by the module's environment. On the other hand, variables `pc1` and `x1` are under the module's control. They can be observed but not modified by the environment.

A guarded command has the following form

```
<label>: <guard> --> <assignments>
```

The label is optional but helps identify the transition when SAL tools display execution traces (for example, when model checkers display counterexamples to LTL properties). The guard is a boolean condition that specifies when the transition is enabled, and the assignments define how the transition updates the module's variables. Primed variables refer to the "new" state and unprimed variables to the "old" state. Any variable that does not occur primed in the assignment is not modified by the transition. For example, transition `wakening` of Figure A.1 is as follows

```
wakening: pc1 = sleeping --> pc1' = trying; x1' = (x2 = tval)
```

It is enabled in any state *s* where the condition `pc1 = sleeping` is satisfied. From such an *s*, transition `wakening` moves the module to a new state s' where `pc1` is equal to `trying` and where `x1` is `true` if `x2` was equal to `tval` in state *s* or `false` otherwise. The components `pc2` and `x2` are left unchanged by the transition; they have the same value in s' and in *s*.

The `process` module is deterministic as only one of its three guarded commands may be enabled in any state. More generally, several guarded commands of a module may be enabled at the same time. In such a case, one of them is chosen non-deterministically. Conversely, if all guards are false, the module cannot perform any transition.

Composition Operators

SAL provides two composition operators for building complex systems from other modules. In Figure A.1, module `system` is the asynchronous composition of two instances of `process`: one with the parameter `tval` instantiated to `FALSE` and the other with the parameter set to `TRUE`.

Modules communicate via their common state variables, that is, via the input, output, and global variables that have the same names. It is possible to hide or rename variables to wire modules together. For example, in the definition of `system`, one must rename the input and output variables of `process[TRUE]` so that the output variables of `process[FALSE]` are input to `process[TRUE]` and vice versa.

The asynchronous composition operator is denoted by `[]`. SAL also provides a synchronous composition operator denoted by `||`. The two types of compositions can be freely mixed. For example, one may construct a system as the synchronous composition of two modules, themselves built by asynchronous composition of other submodules.

The composition operators have the usual semantics. In an asynchronous composition, only one module makes a transition at a time. In a synchronous composition all modules must make simultaneous transitions.

Assertions

Properties are written as shown in Figure A.1, in the form

```
<property name>: THEOREM <module name> |- <temporal formula>
```

This asserts that a module satisfies a property written in temporal logic. Other keywords than `THEOREM` can be used (e.g., `LEMMA` or `CLAIM`). As far as the verification tools are concerned, there is no semantic difference between claims, lemmas, and theorems, but the different keywords may help the user identify more or less important properties.

In this report, we always linear-time temporal logic (LTL) to express properties, although SAL also allows one to use CTL. The LTL modalities are denoted by `G` (*henceforth*), `F` (*eventually*), and `X` (*next*).

A.2 Analysis Tools

SAL is intended to be an open environment, in which it is easy to integrate a variety of analysis tools. The current SAL distribution includes a translator from textual SAL specifications to XML, a lightweight well-formedness checker, a deadlock checker, and several model checkers.

The most useful SAL tool for analyzing timed system is a bounded model checker for infinite-state systems, that relies on the ICS decision procedures and solver. SAL also provides a bounded model checker for finite state systems, which uses a SAT solver. The infinite-state bounded model checker was developed at SRI by Leonardo de Moura and Harald Rueß and is based on lazy theorem proving and lemmas on demand [dMRS02, dMR02a, dMR02b]. More details on the tools are available at `http://ics.csl.sri.com/` and `http://sal.csl.sri.com/`.

In general, a SAL state-transition system M is characterized by its state space X, a set of initial states $I \subseteq X$, and a transition relation $T \subseteq X \times X$. Analysis tools such as the SAL bounded model checkers represent the set of initial states and the transition relation symbolically as two predicates $I(x)$ and $T(x, x')$. A state $x \in X$ is an initial state if and only if it satisfies the predicate $I(x)$. A state x' is a successor of x by the transition relation T if and only if the pair (x, x') satisfies $T(x, x')$. Similarly, a state property P can be represented symbolically via a predicate $P(x)$.

Bounded Model Checking

In its basic form, bounded model checking searches for counterexamples to a safety property. Given a state-transition system $M = (X, I, T)$ and a state property P, bounded model checking at depth $k \in \mathbb{N}$ amounts to finding a finite sequence of states $x_0, \ldots, x_k$ that satisfies the formula

$$\phi \quad = \quad I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-1}, x_k) \wedge \neg P(x_k). \tag{A.1}$$

If such a sequence exists then one can conclude that M does not satisfy $\Box P$ (written `G`P in SAL), since x_k is reachable from the initial state x_0 and does not satisfy P. Bounded model checking requires determining the satisfiability of a formula such as ϕ. SAL provides two bounded model checkers that rely on two different types of solvers. A bounded model checker for finite-state systems, `sal-bmc`, encodes a transition system as boolean formulas and relies on a SAT solver. A bounded model checker for infinite state systems, `sal-inf-bmc`, encodes I and T as quantifier-free first-order formulas that combines linear-arithmetic constraints, boolean constraints, and equalities of terms built from uninterpreted functions. The

satisfiability of the resulting formula ϕ is then determined by an external solver. The default solver is ICS[1] but `sal-inf-bmc` can also use UCLID,[2] CVC,[3] SVC,[4] and CVC Lite.[5]

With the formula ϕ defined in (A.1), bounded model checking searches for a state x_k that violates P and is reachable in k transition steps from one of the initial states. The unsatisfiability of ϕ means only that no such state exists. It does *not* imply that all the states reachable in fewer than k steps satisfy P. For example, the system may deadlock after $k-1$ steps and have no trajectories of length k at all, or it may happen that all states reachable in k steps satisfy P, while some states reachable in $k-1$ steps do not. Thus, the absence of counterexamples to $\Box P$ at depth k does not imply that no counterexample exists at a lower depth. Although perfectly logical given the definition of ϕ, this may be somewhat unintuitive. One may prefer searching for a sequence of states $x_0, \ldots, x_k$ that satisfies the following formula

$$\psi \quad = \quad I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-1}, x_k) \wedge \neg(P(x_0) \vee \ldots \vee P(x_k)). \tag{A.2}$$

If such a sequence exists, one can conclude as above that $\Box P$ is not satisfied. There is a trajectory of length k that passes through a state x_i that violates P. With this new formulation — and *provided trajectories of length k exist* — the absence of counterexample at depth k implies the absence of counterexample at lower depths. If the system deadlocks, and has no trajectory of length k, then bounded model checking at depth k will answer "no counterexample found" for both formulations, but counterexamples may exist at lower depths.

SAL gives the user options to choose between the two formulations of bounded model checking. With the "iterative deepening" option, `sal-bmc` and `sal-inf-bmc` uses formula ϕ to search for a counterexample to $\Box P$ of minimal length: the satisfiability of ϕ is determined for $k = 0$, then for $k = 1$, and so forth, until either a counterexample is found or a maximal depth d is reached. In the default mode, both tools use the other formulation and search for the satisfiability of formula ψ for a user-specified depth k.

Bounded model checking is not limited to safety properties of the form $\Box P$. The SAL bounded model checkers can find counterexamples to other LTL properties. This relies on a translation of LTL properties to Büchi automata.

K-Induction

Bounded model checking searches for counterexamples of bounded length to an LTL property such as $\Box P$. If no counterexample is found, one cannot conclude in general that $\Box P$ is satisfied. In other words, bounded model checking cannot prove $\Box P$. However, it can be easily adapted to support proofs by induction. This allows one to do more than searching for counterexamples and actually prove safety properties.

The standard induction rule for a property $\Box P$ consists of proving that the following two formulas are valid:

- Base case:
$$I(x) \Rightarrow P(x)$$
- Induction step:
$$P(x) \wedge T(x, x') \Rightarrow P(x')$$

[1] `http://www.icansolve.com/`
[2] `http://www-2.cs.cmu.edu/~uclid/`
[3] `http://verify.stanford.edu/CVC/`
[4] `http://verify.stanford.edu/SVC/`
[5] `http://verify.stanford.edu/CVCL/`

This can be transformed into two satisfiability problems that can be solved using a SAT solver or a solver such as ICS, provided the formulas belong to the theory that these tools can decide.

This induction rule can be generalized to induction at depth k (k-induction) for any $k \in \mathbb{N}$. The k-induction rule is as follows:

- Base case:
$$I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-2}, x_{k-1}) \Rightarrow P(x_0) \wedge \ldots \wedge P(x_{k-1})$$
- Induction step:
$$P(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-2}, x_{k-1}) \wedge P(x_{k-1}) \wedge T(x_{k-1}, x_k) \Rightarrow P(x_k)$$

Again, the validity of both formulas can be determined using ICS or similar solvers. The previous rule is the special case of k-induction where $k = 1$.

Both `sal-bmc` and `sal-inf-bmc` support the proof of safety properties via k-induction. It is also possible to provide auxiliary lemmas to the k-induction rules. Such lemmas must themselves be safety properties of the form $\Box Q$. Assuming one has shown that such a lemma is satisfied by a system M, then $\Box Q$ can be used as an auxiliary invariant in the k-induction proof of $\Box P$. This amounts to modifying the k-induction rule as follows:

- Base case:
$$\Big[I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-2}, x_{k-1}) \wedge Q(x_0) \wedge \ldots \wedge Q(x_{k-1})\Big] \Rightarrow P(x_0) \wedge \ldots \wedge P(x_{k-1})$$
- Induction step:
$$\Big[P(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-2}, x_{k-1}) \wedge P(x_{k-1}) \wedge T(x_{k-1}, x_k) \wedge Q(x_0) \wedge \ldots \wedge Q(x_k)\Big] \Rightarrow P(x_k)$$

It is sometimes useful to apply the k-induction rule with $k = 0$, to show that an invariant $\Box Q$ implies another invariant $\Box P$. When $k = 0$, and using $\Box Q$ as a lemma, the base of the induction rule reduces to *true* and the inductive step becomes $Q(x_0) \Rightarrow P(x_0)$. Hence, induction at depth $k = 0$ amounts to proving that $\Box Q$ is a stronger invariant than $\Box P$, by showing that the formula $Q(x_0) \Rightarrow P(x_0)$ is valid.

Appendix B

The Train-Gate-Controller in SAL

```
tgc: CONTEXT =

BEGIN

  TIME: TYPE = REAL;
  ACTION: TYPE = {approach, in, out, exit, lower, down, raise, up};
  TransitionType : TYPE = {regular, elapse};

  % alternatively state transition step / elapse step

  next_trans_type(t: TransitionType): TransitionType =
      IF t = regular THEN elapse ELSE regular ENDIF;

  %----------------
  %  Synchronizer
  %----------------

  transition_module: MODULE =
  BEGIN
    OUTPUT
      delta:  TIME,
      action: ACTION,
      trans:  TransitionType
    TRANSITION
      delta' IN { x : TIME | x >= 0 };
      action' IN {approach, in, out, exit, lower, down, raise, up};
      trans' = next_trans_type(trans)
  END;


  %---------
  %  Train
  %---------

  T_STATE: TYPE = {t0,t1,t2,t3};

  train : MODULE =
  BEGIN
    INPUT
      delta: TIME,
      action: ACTION,
```

```
    trans: TransitionType
  LOCAL
    t_state: T_STATE,
    x: TIME
  INITIALIZATION
    t_state = t0;
    x = 0
  TRANSITION
    [ t0_t1:
      trans' = regular AND t_state = t0 AND action' = approach -->
        t_state' = t1;
        x' = 0
   [] t1_t2:
      trans' = regular AND t_state = t1 AND action' = in AND x > 2 -->
        t_state' = t2
   [] t2_t3:
      trans' = regular AND t_state = t2 AND action' = out -->
        t_state' = t3
   [] t3_t0:
      trans' = regular AND t_state = t3 AND action' = exit -->
        t_state' = t0
   [] delay_train:
      trans' = elapse AND
      (t_state = t1 => x + delta' <= 5) AND
      (t_state = t2 => x + delta' <= 5) AND
      (t_state = t3 => x + delta' <= 5) -->
         x' = x + delta'
   [] skip_train:
      trans' /= elapse AND
      NOT (action' = approach OR action' = in OR
           action' = out OR action' = exit) --> x' = x
    ]
END;


%--------
%  Gate
%--------

G_STATE: TYPE = {g0, g1, g2, g3};

gate: MODULE =
BEGIN
  INPUT
    delta: TIME,
    action: ACTION,
    trans: TransitionType
  LOCAL
     g_state: G_STATE,
     y: TIME
  INITIALIZATION
     g_state = g0;
     y = 0
  TRANSITION
    [ g0_g1:
      trans' = regular AND g_state = g0 AND action' = lower -->
        g_state' = g1;
        y' = 0
   [] g1_g2:
      trans' = regular AND g_state = g1 AND action' = down -->
```

```
        g_state' = g2
   [] g2_g3:
      trans' = regular AND g_state = g2 AND action' = raise -->
        g_state' = g3;
        y' = 0
   [] g3_g0:
      trans' = regular AND g_state = g3 AND action' = up AND y >= 1 -->
        g_state' = g0
   [] delay_gate:
      trans' = elapse AND
      (g_state = g1 => y + delta' <= 1) AND
      (g_state = g3 => y + delta' <= 2) -->
         y' = y + delta'
   [] skip_gate:
      trans' /= elapse AND
      NOT (action' = lower OR action' = down OR
           action' = raise OR action' = up) --> y' = y
    ]
END;


%--------------
%  Controller
%--------------

C_STATE: TYPE = {c0, c1, c2, c3};

controller : MODULE =
BEGIN
  INPUT
    delta: TIME,
    action: ACTION,
    trans: TransitionType
  LOCAL
    c_state: C_STATE,
    z: TIME
  INITIALIZATION
    c_state = c0;
    z = 0
  TRANSITION
    [ c0_c1:
      trans' = regular AND c_state = c0 AND action' = approach -->
        c_state' = c1;
        z' = 0
   [] c1_c2:
      trans' = regular AND c_state = c1 AND z = 1 AND action' = lower -->
        c_state' = c2
   [] c2_c3:
      trans' = regular AND c_state = c2 AND action' = exit -->
        c_state' = c3;
        z' = 0
   [] c3_c0:
      trans' = regular AND c_state = c3 AND action' = raise -->
        c_state' = c0
   [] delay_controller:
      trans' = elapse AND
      (c_state = c1 => z + delta' <= 1) AND
      (c_state = c3 => z + delta' <= 1) -->
         z' = z + delta'
   [] skip_controller:
```

```
        trans' /= elapse AND
        NOT (action' = approach OR action' = lower
              OR action' = exit OR action' = raise) --> z' = z
      ]
  END;


  %----------------------------------------
  %  Full system: synchronous composition
  %----------------------------------------

  system: MODULE =
    transition_module || train || gate || controller;


  %-------------
  % Properties
  %-------------

  safe: LEMMA system |- G(t_state = t2 => g_state = g2);

END
```

Appendix C

The Train-Gate-Controller with Timeouts

```
tgc_with_timeout: CONTEXT =

BEGIN
  SIGNAL: TYPE = { approach, exit, lower, raise };
  TIME: TYPE = REAL;
  N: NATURAL = 3;
  INDEX: TYPE = [1..N];
  TIMEOUT_ARRAY: TYPE = ARRAY INDEX OF TIME;

  %-----------------------
  %  Minimum of an array
  %-----------------------

  recur_min(x: TIMEOUT_ARRAY, min_sofar: TIME, idx: [0 .. N]): TIME =
    IF idx = 0 THEN min_sofar
       ELSE recur_min(x, min(min_sofar, x[idx]), idx-1)
    ENDIF;

  min(x: TIMEOUT_ARRAY): TIME = recur_min(x, x[N], N-1);

  %-----------------------------------------------------------
  %  Clock module: makes time elapse up to the next timeout
  %-----------------------------------------------------------

  clock: MODULE =
    BEGIN
      INPUT time_out: TIMEOUT_ARRAY
      GLOBAL flag1, flag2: BOOLEAN
      OUTPUT time: TIME
    INITIALIZATION
      time = 0
    TRANSITION
      [ time_elapses: time < min(time_out) AND (NOT flag1) AND (NOT flag2)
                --> time' = min(time_out) ]
    END;

  %---------
  %  Train
```

```
%---------

T_STATE: TYPE = { t0, t1, t2, t3 };

train: MODULE =

BEGIN
  INPUT
    time: TIME
  OUTPUT
    timeout: TIME,
    msg1: SIGNAL
  GLOBAL flag1: BOOLEAN
  LOCAL
    t_state: T_STATE
  INITIALIZATION
    t_state = t0;
    timeout IN { x: TIME | time < x };
    flag1 = FALSE
  TRANSITION
    [ t0_t1:
      t_state = t0 AND time = timeout -->
        t_state' = t1;
        msg1' = approach;
        flag1' = TRUE;
        timeout' IN { x: TIME | time + 2 < x AND x <= time + 5 }
   [] t1_t2:
      t_state = t1 AND time = timeout -->
        t_state' = t2;
        timeout' IN { x: TIME | time < x AND x <= time + 5 }
   [] t2_t3:
      t_state = t2 AND time = timeout -->
        t_state' = t3;
        timeout' IN { x: TIME | time < x AND x <= time + 5 }
   [] t3_t0:
      t_state = t3 AND time = timeout -->
        t_state' = t0;
        msg1' = exit;
        flag1' = TRUE;
        timeout' IN { x: TIME | time < x }
    ]
  END;

%---------
%  GATE
%---------

G_STATE: TYPE = { g0, g1, g2, g3 };

gate: MODULE =

BEGIN
  INPUT
    time: TIME,
    msg2: SIGNAL
  OUTPUT
    timeout: TIME
  GLOBAL flag2: BOOLEAN
  LOCAL
    g_state: G_STATE
```

```
  INITIALIZATION
    g_state = g0;
    timeout IN { x: TIME | time < x }
  TRANSITION
    [ g0_g1:
      g_state = g0 AND msg2 = lower AND flag2 = TRUE -->
        g_state' = g1;
        flag2' = FALSE;
        timeout' IN { x: TIME | time < x AND x <= time + 1 }
   [] g1_g2:
      g_state = g1 AND time = timeout -->
        g_state' = g2;
        timeout' IN { x: TIME | time < x }
   [] g2_g3:
      g_state = g2 AND msg2 = raise AND flag2 = TRUE -->
        g_state' = g3;
        flag2' = FALSE;
        timeout' IN { x: TIME | time + 1 <= x AND x <= time + 2 }
   [] g3_g0:
      g_state = g3 AND time = timeout -->
        g_state' = g0;
        timeout' IN { x: TIME | time < x }
    ]
  END;

%-------------
%  Controller
%-------------

C_STATE: TYPE = { c0, c1, c2, c3 };

controller : MODULE =

BEGIN
  INPUT
    time: TIME,
    msg1: SIGNAL
  OUTPUT
    timeout: TIME,
    msg2: SIGNAL
  GLOBAL flag1, flag2: BOOLEAN
  LOCAL
    c_state: C_STATE
  INITIALIZATION
    c_state = c0;
    timeout IN { x: TIME | time < x };
    flag2 = FALSE
  TRANSITION
    [ c0_c1:
      c_state = c0 AND msg1 = approach AND flag1 = TRUE -->
        c_state' = c1;
        flag1' = FALSE;
        timeout' = time + 1
   [] c1_c2:
      c_state = c1 AND time = timeout -->
        c_state' = c2;
        msg2' = lower;
        flag2' = TRUE;
        timeout' IN { x: TIME | time < x }
   [] c2_c3:
```

```
        c_state = c2 AND msg1 = exit AND flag1 = TRUE -->
          c_state' = c3;
          flag1' = FALSE;
          timeout' IN { x: TIME | time < x AND x <= time + 1 }
     [] c3_c0:
        c_state = c3 AND time = timeout -->
          c_state' = c0;
          msg2' = raise;
          flag2' = TRUE;
          timeout' IN { x: TIME | time < x }
      ]
    END;

  %-----------------------------------------------------
  %  Asynchronous composition: all processes together
  %    time_out[i] = timeout variable of train (i=1),
  %    gate (i=1), controller (i=3)
  %-----------------------------------------------------

  tgc: MODULE =
    WITH OUTPUT time_out: TIMEOUT_ARRAY
       (RENAME timeout TO time_out[1] IN train)
      []
       (RENAME timeout TO time_out[2] IN gate)
      []
       (RENAME timeout TO time_out[3] IN controller);

  system: MODULE = clock [] tgc;

  %-----------
  % properties
  %-----------

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

  time_aux2: LEMMA system |- G(t_state = t2 => time_out[1] - time <= 5);

  time_aux3: LEMMA
     system |- G(t_state = t1 AND g_state = g1 => time_out[1] > time_out[2]);

  safe: LEMMA system |- G(t_state = t2 => g_state = g2);

  nosafe: LEMMA system |- G(t_state = t2 => g_state = g3);

END
```

Appendix D

Fischer's Mutual Exclusion Protocol

```
fischer: CONTEXT =

BEGIN
  N: NATURAL = 2;
  IDENTITY: TYPE = [1 .. N];
  LOCK_VALUE: TYPE = [0 .. N];
  TIME: TYPE = REAL;
  delta1: TIME = 2;
  delta2: TIME = 4;
  TIMEOUT_ARRAY: TYPE = ARRAY IDENTITY OF TIME;

  %-----------------------
  %  Minimum of an array
  %-----------------------

  recur_min(x: TIMEOUT_ARRAY, min_sofar: TIME, idx: [0 .. N]): TIME =
    IF idx = 0 THEN min_sofar
       ELSE recur_min(x, min(min_sofar, x[idx]), idx-1)
    ENDIF;

  min(x: TIMEOUT_ARRAY): TIME = recur_min(x, x[N], N-1);

  %-----------------------------------------------------------
  %  Clock module: makes time elapse up to the next timeout
  %-----------------------------------------------------------

  clock: MODULE =
    BEGIN
      INPUT time_out: TIMEOUT_ARRAY
      OUTPUT time: TIME
    INITIALIZATION
      time = 0
    TRANSITION
      [ time_elapses: time < min(time_out) --> time' = min(time_out) ]
    END;

  %--------------
  %  Process[i]
  %--------------

  PC: TYPE = { sleeping, waiting, trying, critical };
```

```
process[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    GLOBAL lock: LOCK_VALUE
    OUTPUT timeout: TIME
    LOCAL pc: PC
  INITIALIZATION
    pc = sleeping;
    timeout IN { x: TIME | time < x };
    lock = 0
  TRANSITION
    [ waking_up:
      pc = sleeping AND time = timeout AND lock = 0 -->
           pc' = waiting;
           timeout' IN { x: TIME | time < x AND x <= time + delta1 }
   [] try_again_later:
      pc = sleeping AND time = timeout AND lock /= 0 -->
           timeout' IN { x: TIME | time < x }
   [] setting_lock:
      pc = waiting AND time = timeout -->
           pc' = trying;
           lock' = i;
           timeout' IN { x: TIME | time + delta2 <= x }
   [] entering_cs:
      pc = trying AND time = timeout AND lock = i -->
           pc' = critical;
           timeout' IN { x: TIME | time < x }
   [] lock_changed:
      pc = trying AND time = timeout AND lock /= i -->
           pc' = sleeping;
           timeout' IN { x: TIME | time < x }
   [] exiting_cs:
      pc = critical AND time = timeout -->
           pc' = sleeping;
           lock' = 0;
           timeout' IN { x: TIME | time < x }
    ]
  END;

 %-----------------------------------------------------
 %  Asynchronous composition: all processes together
 %    time_out[i] = timeout variable of process[i]
 %-----------------------------------------------------

 processes: MODULE =
   WITH OUTPUT time_out: TIMEOUT_ARRAY
      ([] (i: IDENTITY): (RENAME timeout TO time_out[i] IN process[i]));

 system: MODULE = clock [] processes;


 %--------------
 %  Properties
 %--------------

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

 time_aux2: LEMMA
```

```
      system |- G(FORALL (i: IDENTITY):
                    pc[i] = waiting => time_out[i] - time <= delta1);

   time_aux3: LEMMA
      system |- G(FORALL (i, j: IDENTITY):
                    lock = i AND pc[j] = waiting => time_out[i] > time_out[j]);

   logical_aux1: LEMMA
      system |- G(FORALL (i, j: IDENTITY):
                    pc[i] = critical => lock = i AND pc[j] /= waiting);

   mutex: THEOREM
      system |- G(FORALL (i: IDENTITY): pc[i] = critical => lock = i);

   mutual_exclusion: THEOREM
      system |- G(FORALL (i, j: IDENTITY):
                    i /= j AND pc[i] = critical => pc[j] /= critical);

END
```

Appendix E

Fischer's Protocol: Revised Specifications

```
fischer2: CONTEXT =

BEGIN

  N: NATURAL = 2;

  IDENTITY: TYPE = [1 .. N];

  LOCK_VALUE: TYPE = [0 .. N];

  TIME: TYPE = REAL;

  delta1: {x: REAL | 0 < x};

  delta2: {x : REAL | delta1 < x};

  TIMEOUT_ARRAY: TYPE = ARRAY IDENTITY OF TIME;

  %-----------------------
  %  Minimum of an array
  %-----------------------

  is_min(x: TIMEOUT_ARRAY, t: TIME): bool =
    (FORALL (i: IDENTITY): t <= x[i]) AND (EXISTS (i: IDENTITY): t = x[i]);


  %-----------------------------------------------------------
  %  Clock module: makes time elapse up to the next timeout
  %-----------------------------------------------------------

  clock: MODULE =
    BEGIN
      INPUT time_out: TIMEOUT_ARRAY
      OUTPUT time: TIME
    INITIALIZATION
      time = 0
    TRANSITION
      [ time_elapses:
```

```
        (FORALL (i: IDENTITY): time < time_out[i]) -->
           time' IN { t: TIME | is_min(time_out, t) }
    ]
  END;


%--------------
%  Process[i]
%--------------

PC: TYPE = { sleeping, waiting, trying, critical };

process[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    GLOBAL lock: LOCK_VALUE
    OUTPUT timeout: TIME
    LOCAL pc: PC
  INITIALIZATION
    pc = sleeping;
    timeout IN { x: TIME | time < x };
    lock = 0
  TRANSITION
    [ waking_up:
      pc = sleeping AND time = timeout AND lock = 0 -->
           pc' = waiting;
           timeout' IN { x: TIME | time < x AND x <= time + delta1 }
   [] try_again_later:
      pc = sleeping AND time = timeout AND lock /= 0 -->
           timeout' IN { x: TIME | time < x }
   [] setting_lock:
      pc = waiting AND time = timeout -->
           pc' = trying;
           lock' = i;
           timeout' IN { x: TIME | time + delta2 <= x }
   [] entering_cs:
      pc = trying AND time = timeout AND lock = i -->
           pc' = critical;
           timeout' IN { x: TIME | time < x }
   [] lock_changed:
      pc = trying AND time = timeout AND lock /= i -->
           pc' = sleeping;
           timeout' IN { x: TIME | time < x }
   [] exiting_cs:
      pc = critical AND time = timeout -->
           pc' = sleeping;
           lock' = 0;
           timeout' IN { x: TIME | time < x }
    ]
  END;

%-----------------------------------------------------
%  Asynchronous composition: all processes together
%    time_out[i] = timeout variable of process[i]
%-----------------------------------------------------

processes: MODULE =
  WITH OUTPUT time_out: TIMEOUT_ARRAY
     ([] (i: IDENTITY): (RENAME timeout TO time_out[i] IN process[i]));
```

```
   system: MODULE = clock [] processes;


   %--------------
   %  Properties
   %--------------

   time_aux0: LEMMA
      system |- G(time >= 0 AND FORALL (i: IDENTITY): time_out[i] > 0);

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

   time_aux2: LEMMA
      system |- G(FORALL (i: IDENTITY):
                    pc[i] = waiting => time_out[i] - time <= delta1);

   time_aux3: LEMMA
      system |- G(FORALL (i, j: IDENTITY):
                    lock = i AND pc[j] = waiting => time_out[i] > time_out[j]);

   logical_aux1: LEMMA
      system |- G(FORALL (i, j: IDENTITY):
                    pc[i] = critical => lock = i AND pc[j] /= waiting);

   mutex: THEOREM
      system |- G(FORALL (i: IDENTITY): pc[i] = critical => lock = i);

   mutual_exclusion: THEOREM
      system |- G(FORALL (i, j: IDENTITY):
                    i /= j AND pc[i] = critical => pc[j] /= critical);

END
```

Appendix F

Simplified TTA Startup

```
%
% Simplified TTA startup protocol
% - Does not model failures
% - Assumes a simple reliable hub
% - Includes timing
%

simple_startup2: CONTEXT =

BEGIN

  N: NATURAL = 10;

  % Ugly type definition to work around ICS's limitations
  % (incompleteness when dealing with integers)

  IDENTITY: TYPE = { x: [1 .. N] | x=1 OR x=2 OR x=3 OR x=4 OR x=5 OR
                                   x=6 OR x=7 OR x=8 OR x=9 OR x=10 };

  TIME: TYPE = REAL;

  TIMEOUT_ARRAY: TYPE = ARRAY IDENTITY OF TIME;


  %--------------------------------------------------
  %  Delays, assuming all slots have the same length
  %--------------------------------------------------

  slot_time: TIME = 3;

  round_time: TIME = slot_time * N;

  % propagation delay  (must be smaller than half the slot time)

  propagation: { x : TIME | 0 < x AND x < slot_time/2 };

  % maximal time in init state

  max_init_time: TIME = 30;
```

```
  % timeouts in listen and coldstart states

  tau_startup(i: IDENTITY): TIME = slot_time * (i - 1);

  tau_listen(i: IDENTITY): TIME = 2 * round_time + tau_startup(i);

%  tau_listen(i: IDENTITY): TIME = 2 * round_time;

  tau_coldstart(i: IDENTITY): TIME = round_time + tau_startup(i);


  %-------------------------------------------
  %  Hub/communication channel model
  %-------------------------------------------

  message: TYPE = { cs_frame, i_frame };

  calendar: TYPE =
    [# flag: ARRAY IDENTITY OF bool,  % which nodes have to receive the message
       content: message,              % message
       origin: IDENTITY,              % sender
       send: TIME,                    % transmission time (useful for proofs)
       delivery: TIME                 % reception time (the same for all recipients)
     #];


  %---------------------------
  %  Operations on calendars
  %---------------------------

  %----------------------------------------------------------------
  % empty calendar: content, origin, and delivery are irrelevant
  %----------------------------------------------------------------

  empty_cal: calendar = (# flag := [[i: IDENTITY] false],
                           content := i_frame,
                           origin := 1,
                           send := 0,
                           delivery := 0 #);

  empty?(cal: calendar): bool = FORALL (i: IDENTITY): NOT cal.flag[i];


  %---------------------------------------
  % Check for pending events and messages
  %---------------------------------------

  event_pending?(cal: calendar, i: IDENTITY): bool = cal.flag[i];

  i_frame_pending?(cal: calendar, i: IDENTITY): bool =
     cal.flag[i] AND cal.content = i_frame;

  cs_frame_pending?(cal: calendar, i: IDENTITY): bool =
     cal.flag[i] AND cal.content = cs_frame;

  cs_frame?(cal: calendar): bool =
     NOT empty?(cal) AND cal.content = cs_frame;

  i_frame?(cal: calendar): bool =
```

```
   NOT empty?(cal) AND cal.content = i_frame;


%-----------------------------------------------------------
% occurrence time and origin of the pending events
% both are meaningful only if the calendar is not empty
%-----------------------------------------------------------

event_time(cal: calendar, i: IDENTITY): TIME = cal.delivery;

frame_origin(cal: calendar, i: IDENTITY): IDENTITY = cal.origin;


%-----------------------------
% remove event received by i
%-----------------------------

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


%-----------------------------------------------------
% broadcast a message from i to all nodes except i
%  - t is the transmission time
% if there is already a message m0 being sent then
% a collision occurs and is resolved as follows:
%  - m0 remains the transmitted messages
%  - node i will not receive m0
%  - message m is dropped
%-----------------------------------------------------

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;


%-----------------------------------------------
% time of the next event in the calendar
% only meaningful if the calendar is not empty
%-----------------------------------------------

first_event(cal: calendar): TIME = cal.delivery;


%-------------------------------------------------
%  Clock module
%  - input: timeout of each node + calendar
%  - if the calendar is empty, the clock module
%  advances time up to the smallest timeout
%  - if a message is in the bus, time advances
%  to the smallest timeout or to the bus delivery
%  time, whichever is smaller
%-------------------------------------------------
```

```
time_can_advance(cal: calendar, time_out: TIMEOUT_ARRAY, t: TIME): BOOLEAN =
    IF empty?(cal) THEN
        (FORALL (i: IDENTITY): t < time_out[i])
    ELSE
        (FORALL (i: IDENTITY): t < time_out[i]) AND t < first_event(cal)
    ENDIF;

is_next_event(cal: calendar, time_out: TIMEOUT_ARRAY, t: TIME): BOOLEAN =
    IF empty?(cal) THEN
              (FORALL (i: IDENTITY): t <= time_out[i])
          AND (EXISTS (i: IDENTITY): t = time_out[i])
    ELSE
              (FORALL (i: IDENTITY): t <= time_out[i])
          AND t <= first_event(cal)
          AND (t = first_event(cal) OR (EXISTS (i: IDENTITY): t = time_out[i]))
    ENDIF;


clock: MODULE =
  BEGIN
    INPUT time_out: TIMEOUT_ARRAY
    INPUT cal: calendar
    OUTPUT time: TIME
  INITIALIZATION
    time = 0
  TRANSITION
    [ time_elapses:
      time_can_advance(cal, time_out, time) -->
           time' IN { t: TIME | is_next_event(cal, time_out, t) } ]
  END;




%------------------------
% Next slot after slot i
%------------------------

inc(i: IDENTITY): IDENTITY = IF i=N THEN 1 ELSE i+1 ENDIF;


%-------------------------------------------------
% Number of slots between slot i and next slot j
%     slot_delay(i, i + 1) = 0 if i<N
%     slot_delay(N, 1) = 0
%     slot_delay(i, i) = N-1
%-------------------------------------------------

slot_delay(i, j: IDENTITY): [0 .. N-1]  =
   IF i < j THEN j - i - 1 ELSE N - i + j - 1 ENDIF;



%----------------------------------
%  States of a node during startup
%----------------------------------
```

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


%-------------------
%  Node[i] process
%-------------------

node[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    OUTPUT timeout: TIME
    OUTPUT slot: IDENTITY     % slot and pc need to be output
    OUTPUT pc: PC             % to be read by the abstraction module
    GLOBAL cal: calendar
 INITIALIZATION
    pc = init;
    timeout IN { x: TIME | time < x AND x < max_init_time};
    cal = empty_cal;
 TRANSITION
    [ init_to_listen:
      pc = init AND time = timeout -->
         pc' = listen;
         timeout' = time + tau_listen(i)

   % reception of a frame in the init state ==> drop it
   [] frame_in_init:
      pc = init AND event_pending?(cal, i) AND time = event_time(cal, i) -->
         cal' = consume_event(cal, i)

   % end of listen phase: send cs frame, move to coldstart state
   % bcast function takes care of collisions
   [] listen_to_coldstart:
      pc = listen AND time = timeout -->
         pc' = coldstart;
         timeout' = time + tau_coldstart(i);
         cal' = bcast(cal, cs_frame, i, time)

   % reception of a cs_frame in the listen state:
   % move to coldstart and set timeout
   [] cs_frame_in_listen:
      pc = listen AND cs_frame_pending?(cal, i) AND time = event_time(cal, i) -->
         pc' = coldstart;
         timeout' = time + tau_coldstart(i) - propagation;
         cal' = consume_event(cal, i)

   % for reception of an i_frame in the listen state: see below

   % reception of a cs_frame in the coldstart state:
   % synchronize on the sender and move to active state
   [] 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)

   % end of coldstart phase (timeout tau_coldstart(i) is reached)
   % broadcast a cs_frame and loop back to coldstart state
   % --> TO DO: check if it's OK to go directly to active from here
   [] coldstart_to_coldstart:
```

```
    pc = coldstart AND time = timeout -->
       % pc' = coldstart;
       timeout' = time + tau_coldstart(i);
       cal' = bcast(cal, cs_frame, i, time)

  % reception of an i_frame in listen or coldstart state: synchronize and move
  % to the active state
  [] i_frame_processed:
     (pc = listen OR pc = coldstart) AND i_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)


  % active state: end of current slot, new slot /= i
  [] passive_slot:
     pc = active AND time = timeout AND inc(slot) /= i -->
       timeout' = time + slot_time;
       slot' = inc(slot)

  % active state: end of current slot, new slot = i
  % broadcast an i_frame
  [] active_slot:
     pc = active AND time = timeout AND inc(slot) = i -->
       timeout' = time + slot_time;
       slot' = inc(slot);
       cal' = bcast(cal, i_frame, i, time)

  % reception of an i_frame
  % in active state: just consume the event. No action
  [] i_frame_ignored:
     pc = active AND i_frame_pending?(cal, i) AND time = event_time(cal, i) -->
         cal' = consume_event(cal, i)

   ]
 END;


%-----------------------------------------------------
%  Asynchronous composition: all processes together
%    time_out[i] = timeout variable of process[i]
%-----------------------------------------------------

nodes: MODULE =
  WITH OUTPUT time_out: TIMEOUT_ARRAY,
              pc: ARRAY IDENTITY OF PC,
              slot: ARRAY IDENTITY OF IDENTITY
     ([] (i: IDENTITY): (RENAME timeout TO time_out[i],
                                pc TO pc[i], slot TO slot[i] IN node[i]));

tta: MODULE = clock [] nodes;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ABSTRACTION AND MONITORS  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%-----------------------------------------------------------
% Abstraction module: define the abstract state variables
%-----------------------------------------------------------

abstractor: MODULE =
  BEGIN
    INPUT
      time: TIME,
      cal: calendar,
      time_out: TIMEOUT_ARRAY,
      pc: ARRAY IDENTITY OF PC,
      slot: ARRAY IDENTITY OF IDENTITY
    OUTPUT
      A1, A2, A3, A4, A5, A6: BOOLEAN
  DEFINITION
    A1 = empty?(cal) AND (FORALL (i: IDENTITY): pc[i] = init OR pc[i] = listen);

    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)
                 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));

    A3 = empty?(cal)
         AND (EXISTS (i: IDENTITY): pc[i] = coldstart)
         AND (FORALL (i: IDENTITY):
                 pc[i] = init OR pc[i] = listen OR pc[i] = coldstart)
         AND (FORALL (i: IDENTITY):
                 pc[i] = coldstart => time_out[i] - time <= tau_coldstart(i))
         AND (FORALL (i, j: IDENTITY): pc[i] = coldstart AND pc[j] = coldstart
                 AND i < j => time_out[j] - time_out[i] > propagation)
         AND (FORALL (i, j: IDENTITY): pc[i] = coldstart AND pc[j] = listen =>
                              time_out[j] - time_out[i] > propagation);

    A4 = cs_frame?(cal)
         AND pc[cal.origin] = coldstart
         AND time_out[cal.origin] = cal.send + tau_coldstart(cal.origin)
         AND NOT event_pending?(cal, cal.origin)
         AND (FORALL (i: IDENTITY): pc[i] = coldstart AND i /= cal.origin =>
                  (event_pending?(cal, i) AND event_time(cal, i) < time_out[i])
               OR (time_out[i] - cal.send >= tau_coldstart(i) AND
                         time_out[i] - time <= tau_coldstart(i)))
         AND (FORALL (i: IDENTITY): pc[i] = listen =>
                  (event_pending?(cal, i) AND event_time(cal, i) < time_out[i])
               OR time_out[i] >= cal.send + tau_listen(i))
         AND (FORALL (i: IDENTITY): pc[i] = active =>
               slot[i] = cal.origin AND time_out[i] = cal.send + slot_time);

    A5 = empty?(cal)
         AND (EXISTS (i: IDENTITY): pc[i] = active)
         AND (FORALL (i: IDENTITY):
                 pc[i] = active => time_out[i] <= time + slot_time)
         AND (FORALL (i, j: IDENTITY): pc[i] = active AND pc[j] = active =>
               (time < time_out[i] AND time < time_out[j] =>
```

```
                  slot[i] = slot[j] AND time_out[i] = time_out[j])
          AND (time = time_out[i] AND time = time_out[j] => slot[i] = slot[j])
          AND (time < time_out[i] AND time = time_out[j] =>
                 slot[i] = inc(slot[j]) AND time_out[i] = time_out[j] + slot_time))
        AND (FORALL (i, j: IDENTITY):
               pc[i] = active AND (pc[j] = listen OR pc[j] = coldstart) =>
                    time_out[j] > time_out[i] +
                          slot_delay(slot[i], i) * slot_time + propagation);

    A6 = i_frame?(cal)
         AND pc[cal.origin] = active AND slot[cal.origin] = cal.origin
         AND time_out[cal.origin] = cal.send + slot_time
         AND (FORALL (i: IDENTITY): pc[i] = active AND time < time_out[i] =>
               slot[i] = cal.origin AND time_out[i] = time_out[cal.origin])
         AND (FORALL (i: IDENTITY): pc[i] = active AND time = time_out[i] =>
                inc(slot[i]) = cal.origin AND
                    time_out[cal.origin] = time_out[i] + slot_time)
         AND (FORALL (i: IDENTITY): pc[i] = listen =>
                 (event_pending?(cal, i) AND event_time(cal, i) < time_out[i])
               OR (time_out[i] >= cal.send + tau_listen(i)))
         AND (FORALL (i: IDENTITY): pc[i] = coldstart =>
                 event_pending?(cal, i) AND event_time(cal, i) < time_out[i]);

  END;


%------------
%  Monitors
%------------

abstract_state: TYPE = {a1, a2, a3, a4, a5, a6, bad };

monitor: MODULE =
  BEGIN
    INPUT A1, A2, A3, A4, A5, A6: BOOLEAN
    LOCAL state: abstract_state
  INITIALIZATION
    state = a1
  TRANSITION
     [ state = a1 -->
         state' = IF A1' THEN a1 ELSIF A2' THEN a2 ELSE bad ENDIF

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

    [] state = a3 -->
         state' = IF A3' THEN a3 ELSIF A4' THEN a4 ELSE bad ENDIF

    [] state = a4 -->
         state' = IF A4' THEN a4 ELSIF A3' THEN a3 ELSIF A5' THEN a5 ELSE bad ENDIF

    [] state = a5 -->
         state' = IF A5' THEN a5 ELSIF A6' THEN a6 ELSE bad ENDIF

    [] state = a6 -->
         state' = IF A6' THEN a6 ELSIF A5' THEN a5 ELSE bad ENDIF

    [] ELSE --> state' = bad
     ]
  END;
```

```
%--------------
%  Properties
%--------------

system: MODULE = tta || abstractor || monitor;

%
% time_aux0 to time_aux2 are provable by induction at depth 1
% time_aux3 is provable by induction at depth 4, or by induction
%   at depth 1 using time_aux0 as a lemma
%

time_aux0: LEMMA
  system |- G(time >= 0);

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 <= first_event(cal)));

time_aux3: LEMMA
  system |- G(FORALL (i: IDENTITY): time_out[i] > 0);


%
% delivery_delay: all by induction at depth 1
%

delivery_delay: LEMMA
  system |- G(empty?(cal) OR first_event(cal) <= cal.send + propagation);

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

delivery_delay2: LEMMA
  system |-
     G(FORALL (i: IDENTITY):
          i_frame_pending?(cal, i) => event_time(cal, i) <= cal.send + propagation);

delivery_delay3: LEMMA
  system |-
     G(FORALL (i: IDENTITY):
          cs_frame_pending?(cal, i) => event_time(cal, i) <= cal.send + propagation);


%
% a sender does not receive its own frame: by induction at depth 1
%

calendar_aux1: LEMMA
  system |- G(NOT event_pending?(cal, cal.origin));



%----------------------
```

```
  %  Abstraction lemmas
  %----------------------

  abstract_init: LEMMA system |- A1;

  abstract_a1: LEMMA system |- G(state = a1 => A1);
  abstract_a2: LEMMA system |- G(state = a2 => A2);
  abstract_a3: LEMMA system |- G(state = a3 => A3);
  abstract_a4: LEMMA system |- G(state = a4 => A4);
  abstract_a5: LEMMA system |- G(state = a5 => A5);
  abstract_a6: LEMMA system |- G(state = a6 => A6);

  abstract_invar: LEMMA system |- G(state /= bad);

  %
  %  Safety property
  %

  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]);

END
```