Mechanized Verification of Real-Time Systems Using PVS[1]

N. Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025
Phone: (415)859-5272
shankar@csl.sri.com

March 2, 1993

[1]This work was supported by National Aeronautics and Space Administration Langley Research Center and the US Naval Research Laboratory under contract NAS1-18969 and by the US Naval Research Laboratory contract N00015-92-C-2177.

Abstract

Some important properties of certain safety-critical systems can only be established by a careful analysis of their real-time behavior. These systems are typically also designed to meet other functional requirements where the real-time behavior is irrelevant. A suitable verification framework for such systems must therefore be capable of coping with functional as well as real-time behavior. We present an approach to the verification of the real-time behavior of concurrent programs and describe its mechanization using the PVS proof checker. Our approach to real-time behavior extends previous verification techniques for concurrent programs by introducing a new operator for reasoning about absolute time. This operator returns the amount of time that has elapsed since the given predicate last held in the current computation. This approach is mechanized by first encoding the computational behavior of real-time systems within the higher-order logic of PVS and employing the interactive proof checker of PVS to develop the proofs. Our approach is illustrated using two examples: Fischer's real-time mutual exclusion protocol and a railroad crossing controller. We also discuss related approaches to formalizing real-time behavior.

Contents

Chapter 1

Introduction

Time is used in several ways in computing. Consider the following simple problem specifications involving the use of time:

1. No two processes can be in their critical sections at the same time.
2. The backup computer is not to be brought online unless one of the primary computers fails, and it must be brought online within 5 seconds of the failure.
3. Messages are delivered in the order received.
4. The screen is refreshed every minute.
5. All local clocks must always be synchronized to within δ.
6. Messages are delivered within γ time units.
7. Successive requests are at least ρ time units apart.
8. The emergency lights start to flash only when the fluid level reaches the *low* mark.
9. The sender times out if no acknowledgment is received within 30 seconds.

Several observations can be made about the formalization of systems meeting the sort of requirements shown above. In some of the above scenarios, time is used only to order events, for example, if a process has entered its critical section, then no other process must be allowed to enter its critical section until the first process has exited its critical section. Techniques based on Hoare logics [Hoa69] and temporal logics [Pnu77] are capable of describing and proving such properties. In other examples, there are constraints on the absolute time between events as with the message delays and timeouts that involve the explicit use of real time. Notice also that there are time-varying quantities such as fluid levels and local clocks whose values vary (discretely or continuously) with time. We consider only those systems where the state changes in discrete transitions, but that could interact with an environment where there are continuously varying quantities. Indeed, time itself could be regarded as such a quantity.

Since real-time systems are often used in critical contexts, it is important to rigorously demonstrate that certain crucial requirements are met by the system. We briefly survey some of these approaches to formalizing the behavior of real-time systems. A number of excellent references [AL91, AH91] provide additional background [dBHdRR91].

1.1 Related Work

There are already several competing formalisms for expressing real-time behavior. We survey only a few of these in order to contrast these with the approach used in this paper.

Real-time logic (RTL) is an extension of the traditional approach to formalizing time [JM86]. It is a multi-sorted predicate logic with a time domain so that variables and quantifiers can range over time. Events are state changes that occur instantaneously at a point in time. An occurrence function indicates the time of the i'th occurrence of an event e by $@(e, i)$. RTL is convenient for reasoning purely about the timing aspects of a system but can be quite cumbersome when program behavior is also relevant since there is no notion of a discrete system state in RTL. Modecharts [Stu90] is a graphical framework for state machine specification (based on Statecharts) that builds on RTL.

Typical state-based program logics include the various temporal logics [Eme90, Pnu77], Unity [CM88], and TLA (temporal logic of actions) [Lam90]. Pure temporal logic is concerned with temporal ordering rather than with temporal distance. It can be used to express such properties as deadlock freedom, mutual exclusion, and reachability. There have been numerous proposals to extend propositional temporal logic for reasoning about real time. The simplest such proposal is Metric Temporal Logic (MTL) where bounded versions of the temporal operators are introduced [Koy90]. There is a similar extension to the branching time temporal logic CTL [EMSS91]. Timed Propositional Temporal Logic (TPTL) [AH89] is another extension to temporal logic that involves the use of a freeze quantifier whose bound variable binds the "current" value of time and uses this value in inequalities within the scope of the quantifier. The *explicit time* temporal logics (XCTL, RTTL) [HLP90, Ost90] use a special program variable `T` to indicate the value of time at any state and use first-order quantification over time to make similar assertions. Alur and Henzinger [AH91] survey the above variants of temporal logic from the point of view of expressiveness and decidability

The Duration Calculus (DC) [CHR92] is a very expressive interval temporal logic that can be used to reason about time-varying quantities. Here, the formulas contain the implicit parameters b and e that mark the beginning and end, respectively, of a finite interval. The integral $\int P$ returns the duration for which the predicate P was true in the interval from b to e. In particular, $\int \mathbf{1}$ (abbreviated as ℓ) marks the length of the interval since $\mathbf{1}$ is the truth value **true**. The predicate $\lceil P \rceil$ on intervals identifies a nonempty interval where P is almost always true. If B and C are predicates on intervals, then $B; C$ is true of an interval that has two contiguous subintervals so that B is true of the first subinterval and C holds of the second. The temporal Diamond operator $\Diamond A$ is defined as $true; A; true$, which is true on any interval containing a subinterval where A holds. The $\Box$ operator is the dual of $\Diamond$. Skakkebæk, Ravn, Rischel, and Chen [SRRC92] illustrate the use of the duration calculus for specifying and verifying a realistic railroad crossing controller.

As a brief illustration, the assertion "an acknowledgment will be received within 3 seconds of a send," is coded by these various logics as:

RTL	$(\forall i \colon @(Ack, i) \leq @(Send, i) + 3)$
MTL	$\Box(Send \supset \Diamond_{\leq 3} Ack)$
TPTL	$\Box x.(Send \supset \Diamond y.Ack \wedge y \leq x + 3)$
XCTL	$(\forall t \colon \Box(Send \wedge t = T \supset \Diamond(Ack \wedge T \leq t + 3)))$
DC	$\Box((\lceil Send \rceil; true) \wedge \ell \geq 3 \supset \Diamond \lceil Ack \rceil)$

While the above real-time extensions to propositional temporal logic are interesting in their own right, they place too great an emphasis on timed finite-state systems. We are interested in a broader approach encompassing finite and infinite state systems but with extensions to handle time as well. Lamport's Temporal Logic of Actions (TLA) [Lam90] and Chandy and Misra's Unity logic [CM88] are both elegant logics dealing with arbitrary state transition systems. Unity is a highly simplified temporal logic for nondeterministic state transition systems. It provides a small set of useful temporal operators that can be applied only to assertions on states and not on entire temporal formulas as is the case in temporal logic. TLA is a modification of temporal logic; it avoids the next-state operator and replaces it with a notion of *action* that is a binary relation between adjacent states. Real-time extensions to these logics have been discussed in the past. Abadi and Lamport [AL91] present what they call "an old-fashioned recipe for real time" where they model time within TLA using a special process that increments the value of time in discrete steps. Special counter variables that track the value of time are used to specify timing constraints. Schneider, Bloom, and Marzullo [SBM91] have extended Proof Outline Logic to handle real time. The logic contains control predicates to indicate where the control is in a program. They employ an operator that records the time when a control predicate last became true. Carruth and Misra [CM92] employ a similar extension to Unity where for any assertion P, $\overline{P}$ (read "punch P") records the absolute time at which P last went from being false to true. In this approach, $\overline{P}$ is initially equal to the time in the initial state if P is true in the initial state, otherwise, $\overline{P}$ is some negative value (since time ranges only over nonnegative values). Maler, Manna, and Pnueli [MMP91] introduce a duration operator δ, where for a temporal formula ϕ, the value of $\delta(\phi)$ at any state s in an execution of a program is the largest time duration ending in s for which ϕ has continuously held. If ϕ is false at state s, then the value of $\delta(\phi)$ at s is 0.

These latter approaches are straightforward extensions of conventional reasoning techniques. They do not involve any serious logical extensions since they make use of the existing mechanisms in program logics. The key observation regarding these approaches is that certain special operators are employed to record the time of some event in the past history of the system. No other reference to the system's past is needed. The work we describe here is along the lines of these latter approaches to real-time system behavior. We present a computational model that includes a notion of real time.

This model is embedded in the higher-order logic of PVS. One can reason about this model using temporal logic, TLA, or Unity, since any of these can easily be embedded within the PVS logic. However, we do not use any of these formalisms and work directly with an encoding of concurrent, real-time system behavior within the higher-order logic of PVS. Either of the *punch* or the *duration* operators could have been used in our verification. We, however, employ a new operator for reasoning about real-time behavior and illustrate its use with the examples of Fischer's mutual exclusion protocol [Lam87] and a railroad crossing controller. Like the *punch* operator above, this new operator, called *since*, operates on assertions. The value of $|P|$ (read "since P") at a given state in a program execution is the time that has elapsed since p last held. For any P, the value of $|P|$ in the initial state of the computation is arbitrarily set to some positive value (say 1) since there is no previous state where P held and the initial value of $Time$ is 0. If P is false in a state s, then the value of $|P|$ in the next state s' of the computation, is got by adding $Time(s') - Time(s)$ to the value of $|P|$ at s. If P is true at s, then the value of $|P|$ at the next state s' is just $Time(s') - Time(s)$. Note that if P is false in every state preceding a state s, then the value of $|P|$ at s is $Time(s) + 1$ so that there is no corresponding state in the computation preceding s. Proofs involving the punch or the duration operators can easily be recast in terms of the since operator, and vice-versa. Further properties of the *since* operator are illustrated in Section 4.2. The main claim in this paper is that it is feasible to undertake mechanical verification of real-time protocols using a simple computational model and a straightforward extension of existing techniques for concurrent program verification.

Chapter 2

Modeling Real-time Systems

Our goal in this chapter is to formalize the real-time behavior of systems in the higher-order logic of PVS. Similar formalizations can be carried out in other higher-order logics, such as those of the HOL [Gor88] and Nuprl [C$^+$86] systems, and with some additional difficulty in a first-order logic [BM79, Gol90]. We first describe the semantic framework for formalizing and reasoning about real-time systems and show how the various temporal operators can be defined within this semantic framework. Let `state` be an uninterpreted type. A program variable of type `T` is a function from `state` to `T`, *i.e.*, of type `[state -> T]`. A `trace` is a sequence of states; sequences are modelled as functions from `nat` to `state`. A program is a set of traces specified by means of an initialization predicate that constrains the set of initial states, and a collection of atomic actions, which are binary relations on states, such that each pair of adjacent states in any trace for the program satisfies one of the actions.

The above model of program behavior is called the *interleaving* model since it models concurrent execution by the set of interleavings of possibly concurrent program actions. In other words, *concurrency* has been redefined in terms of *nondeterministic*, *sequential* behavior so that a system with possibly concurrent atomic actions A and B is replaced by one where only either $A; B$ (*i.e.*, A followed by B) or $B; A$ is possible.

An *assertion* or a *state predicate* is defined as `[state -> bool]`, *i.e.*, the type of predicates on states. The *invariance* predicate `Inv` asserts that a given state predicate holds of every state of a given trace. Invariance properties are typically proved by induction over the states in any trace of the program, that is, p is shown to be an invariant by showing that for any trace σ satisfying the program, $p(\sigma(0))$ holds, and $p(\sigma(j)) \supset p(\sigma(j+1))$, for any natural number j.

`Time` is a special program variable representing real time and ranging over the nonnegative rational numbers.[1] A *behavior* is a sequence of states such that `Time` never decreases and always eventually increases beyond any rational number value. This latter requirement, called the non-Zeno constraint, ensures that an infinite number of actions cannot take place within a duration of finite length. Note that actions can take a nonnegative amount of time so that if $\sigma(j)$ and $\sigma(j+1)$ are a pair of adjacent states in a behavior satisfying some program action, then $\texttt{Time}(\sigma(j+1)) - \texttt{Time}(\sigma(j))$ is nonnegative. A program identifies a set of behaviors. A specification also identifies a set of behaviors so that a program is a kind of specification. A program satisfies a specification if the set of behaviors given by the program is a subset of the behaviors identified by the specification.

The above model captures the essential aspects of the models behind logics such as temporal logic [Pnu77], TLA [Lam90], or Unity [CM88]. There are a few small differences in the approach to modeling time. In the work of Abadi and Lamport [AL91], there is an explicit process that increments time so that ordinary actions themselves take no time, but time-increment actions are interleaved with ordinary actions. The approach of Maler, Manna, and Pnueli [MMP91] (based on that of Henzinger, Manna, and Pnueli [HMP91]) similarly does not permit time and state to both change in any single atomic action but to interleave them. They also associate with each program transition, lower and upper bounds on the time that a transition can be continuously enabled and not taken. The reasons for these restrictions are somewhat technical but we feel that they make the model excessively complicated and contribute little to the formalization. The lower and upper bounds on actions ought to be part of the program specification and not part of the computation model. Furthermore, the precise instant when a transition occurred should be capable of taking on any value between the time at the state prior to the transition and the time immediately following the transition.

One argument that could be made against the interleaving model in general is that it does not accurately capture truly concurrent behavior

[1] A domain other than the rational numbers can also be used for this purpose. Typically, all one needs is a linearly ordered domain, though it can be important whether the ordering is dense or discrete.

where several actions do occur simultaneously in real time. One should note that the interleaving model is not a model of execution, it is a model of observation. It is assumed that there is a single sequential observer and the properties of the system are proved relative to such an observer. Some work has to be done to correctly model concurrent behavior in terms of interleaving actions in order to accommodate this observational view. The benefit of such an observational view is that the proof rules for proving system properties are greatly simplified. The validity of using interleaving models for concurrent real-time behavior clearly requires more investigation.

2.1 Time in PVS

The PVS theory shown below formalizes the informal discussion above. The type `state` is declared as an *uninterpreted* type, *i.e.*, it has no definition. The theory uses the `sequences` theory from the theories that form the prelude for PVS. The type `time` is defined to contain the nonnegative rational numbers. The type `behavior` is defined to contain all and only those sequences of states that satisfy the nondecreasing constraint `nondec?`, and the non-Zeno constraint `nonzeno?`. Note that the type expression `(behavior?)` is the subtype of those elements of `trace` satisfying the predicate `behavior?`. The term `Inv(pp)(aa)` is true when the assertion `pp` holds at every state in the behavior `aa`, *i.e.*, `pp` is an *invariant* of the behavior `aa`. The term `Init(pp)(aa)` is true if and only if the initial state of `aa` satisfies `pp`.

---

time: THEORY
  BEGIN

  state: TYPE
  USING sequences[state]
  statepred: TYPE $=$ PRED[state]
  Action: TYPE $=$ PRED[[state, state]]
  pp: VAR statepred
  action: VAR Action
  time: TYPE $= \{x\text{: rational} \mid x \geq 0\}$
  Time: [state $\rightarrow$ time]
  seq, seq1, seq2: VAR sequence[state]
  i, j, k: VAR nat
  nondec?(seq): bool $= (\forall\, i, j\text{: } i > j \supset \text{Time}(\text{seq}(j)) \leq \text{Time}(\text{seq}(i)))$

nonzeno?(seq): bool = ($\forall$ (x: rational): ($\exists$ i: x < Time(seq(i))))

behavior?(seq): bool = nondec?(seq) $\wedge$ nonzeno?(seq)

behavior: TYPE = (behavior?)
aa, bb, cc: VAR behavior
Inv(pp)(aa): bool = ($\forall$ (n: nat): pp(aa(n)))

Init(pp)(aa): bool = pp(aa(0))

END time

2.2 Proving Properties of Real-Time Systems

We are interested in using PVS to prove the typical classes of properties such as safety (nothing bad will happen), liveness (something good will eventually happen), and time-bounded versions of such properties.[2] The statement of mutual exclusion: "no two distinct processes can simultaneously be in their critical section," is an example of a safety property. The assertion: "any process waiting to enter its critical section will eventually do so," is an example of a liveness property. Safety and liveness are respectively analogous to the notions of *partial correctness* and *termination* in the case sequential program correctness. In the untimed case, liveness properties are established by using fairness constraints to demonstrate that the program must progress towards a desired goal. Safety properties are typically demonstrated by

[2] A formal definition of safety and liveness properties has been given by Alpern and Schneider [AS85] where they show that any property of a trace can be expressed as a conjunction of a safety and a liveness property. A safety property of a trace is defined as one which if false is falsified by some finite prefix of the trace, *i.e.*, if a safety property $P(\sigma)$ is false for a trace σ, then there exists a prefix α of σ such that for any β, the assertion $P(\alpha;\beta)$ is false, where ';' denotes concatenation. A liveness property Q is defined as one that is true of a trace σ if and only if there is a suffix β of σ such that for any α, the assertion $P(\alpha;\beta)$ holds.

establishing an invariant by means of induction, as discussed above. Time-bounded versions of liveness properties are safety properties since they can be falsified by a finite prefix of a trace corresponding to the relevant time bound. We focus only on *invariance* properties in this report since these are adequate for the safety requirements we wish to prove.

For notational convenience, state predicates are written with references to state suppressed. The property that the value of the variable x in a state is at least 2 greater than than the value of variable y is stated as $x > y + 2$. An initialization assertion has the form $\mathbf{initially}\{P\}$. An invariant assertion on the state predicate P is stated as $\mathbf{invariant}\{P\}$. To prove that $\mathbf{invariant}\{P\}$ holds of a program with initialization predicate *init* and atomic actions S_i, we show that $init \supset p$ and that the Hoare assertion $\{P\}S_i\{P\}$ holds for each atomic action.

Some additional axioms about '*since* P' or $|P|$ are needed to capture real-time behavior. The first axiom asserts that the initial value $|P|$ for any state predicate P is 1. Not all behaviors satisfy this axiom, but those that do are called *rooted behaviors*. Note that the value of *Time* is initially 0.[3]

$$\mathbf{initially}\{|P| = 1\}, \quad \text{for all } P \quad \mathbf{(init)}$$

The second axiom asserts that if P is true in the precondition of an atomic action, then the value of $|P|$ in the postcondition is equal to the delay for the action.

$$\begin{array}{c} \{(r = Time) \wedge P\} \\ S \\ \{|P| = Time - r\} \end{array} \qquad \text{for all } P \quad \mathbf{(step1)}$$

Note that with the above axiom, the difference between the postcondition and precondition time is given by the value of $|\mathbf{true}|$.

The third axiom asserts that if P is false in the precondition of an action, then the postcondition value of $|P|$ is got by adding the delay for the action to the precondition value of $|P|$.

$$\begin{array}{c} \{(r = Time) \wedge (t = |P|) \wedge \neg P\} \\ S \\ \{|P| = t + (Time - r)\} \end{array} \qquad \text{for all } P \quad \mathbf{(step2)}$$

[3]The PVS formalization does not insist that the initial time of a rooted behavior be zero, but merely constrains the initial value of $|P|$ for any P to be one more than the initial value of *Time*. The effect of this on the formalization is negligible.

With the *since* operator and the above axioms, it can be shown that conventional techniques can be used to establish the correctness of programs that exhibit real-time behavior. We illustrate the use of the since operator with the example of Fischer's mutual exclusion protocol in the next chapter.

2.3 The Since Operator in PVS

The PVS theory `since` below formalizes the since operator. The axiom `since_ax` captures both **(step1)** and **(step2)**. Rooted behaviors are expressed by the subtype `behavior0` of `behavior`. The axiom **(init)** is embedded in the subtype constraint of `behavior0`. Note that every suffix of a behavior is a behavior whereas not every suffix of a rooted behavior remains a rooted behavior. The last few definitions are not used elsewhere in this report but are included to illustrate how temporal operators can be formalized in this approach. The type `TFmla` of temporal formulas is defined to contain the predicates on behaviors (in contrast to `statepred` which contains the predicates on states). Any element of `statepred` can be converted into a temporal formula constraining the initial state of a behavior. Similarly, any element of `action` can be seen as constraining the first two states of a behavior. The temporal $\Box$ operator given by `Box` below, holds of a temporal formula `fmla` and a behavior `aa` if every suffix of `aa` satisfies `fmla`.

```
since: THEORY
  BEGIN

  USING time
  i, j, k: VAR nat
  pp: VAR statepred
  action: VAR Action
  aa, bb, cc: VAR behavior
  since: [statepred → [state → time]]
  since_ax:
   AXIOM
     since(pp)(aa(i + 1))
        =
      (IF pp(aa(i)) THEN (Time(aa(i + 1)) − Time(aa(i)))
       ELSE since(pp)(aa(i)) + (Time(aa(i + 1)) − Time(aa(i)))
       ENDIF)
```

behavior0:
  TYPE = $\{$aa: behavior | ($\forall$ pp: since(pp)(aa(0)) = Time(aa(0)) + 1)$\}$
TFmla: TYPE = PRED[behavior]
tfmla(pp): TFmla = (λ aa: pp(aa(0)))

tfmla(action): TFmla = (λ aa: action(aa(0), aa(1)))

Box((fmla: TFmla)): TFmla = (λ aa: ($\forall$ (n: nat): fmla(suffix(aa, n))))

END since

The next two chapters illustrate the use of the above embedding of real-time state transition systems with the examples of a mutual exclusion protocol and a railroad crossing controller.

Chapter 3

A Verification Example: Fischer's Mutual Exclusion Protocol

We now discuss the informal use of the above formalization of state transition systems in verifying a simple protocol that exploits real time. This protocol is described by Lamport [Lam87] and attributed to Michael Fischer. We use the Unity notation to present the protocol before we describe its encoding in PVS.[1] An arbitrary number of processes are represented by the positive natural numbers of the type `posnat` below. To each process `i`, there is a program counter `PC(i)`. There is a program variable `x` which controls the entry into the critical section. The program counters are initially set to `init`, and the value of `x` is initially 0. In the protocols below, each process can take one of three actions labeled:

`Try(i)`: Takes process `i` from the `init` to the `try` state if `x` is `0`.

`Wait(i)`: Takes process `i` from the `try` state to the `wait` state while setting `x` to `i`. There is an upper bound of `hi` on the amount of time that a process spends in its `try` state, and a lower bound of `lo` on the amount of time a process spends in its `wait` state, where `hi` < `lo`.

`Cs(i)`: Takes process `i` from the `wait` state into its critical section `cs` provided `x` is equal to `i`.

[1] We are using only the notation and not the formalism. Also, unlike Unity, we are not placing any fairness constraints on the transitions.

This simplified version of the protocol omits any exit action from the critical section. We prove that this protocol guarantees that no two processes are simultaneously in their critical section.

Program `mutex`

declare `x : nat, PC: [posnat -> {init, try, wait, cs}]`
initially `x = 0` $\|$ $\langle$ $\|$ `i : posnat ::` `PC(i) = init` $\rangle$
assign

```
   ⟨[]i : posnat ::
       PC(i) := try              if x = 0 ∧ PC(i) = init
    [] x, PC(i) := i, wait       if PC(i) = try
    [] PC(i) := cs               if x = i ∧ PC(i) = wait
   ⟩
```

end {`mutex`}

Two timing invariants are associated with timing constraints on the actions. An invariant asserts that a property holds of each state of a behavior satisfying the program, so that an invariant written as $\{A \supset |p| \leq |q|\}$ is to be interpreted as asserting that for every behavior σ satisfying the program and every n, $A(\sigma(n))$ implies $|p|(\sigma(n)) \leq |q|(\sigma(n))$. In some of the proofs, we do indicate whether an expression is interpreted as part of the precondition or the postcondition of an action. Explicit references to state do appear in the PVS specifications. Note that the variables `i`, `j`, and `k` in the assertions below, range over positive natural numbers.

The first invariant associates an upper bound `hi` with the time that any process spends in its `try` state. It states that if a process is in its `try` state, then the time since it was last in its `init` state is at most `hi`.

Invariant Axiom 3.1

$$\{\forall \mathtt{i}{:}\, \mathtt{PC(i)} = \mathtt{try} \supset |\mathtt{PC(i)} = \mathtt{init}| \leq \mathtt{hi}\}.$$

The second timing invariant associates a lower bound `lo` with the amount of time that any process spends in its `wait` state. It asserts that if a process is in its critical section, then the time since it was last in its `try` state exceeds the time that it was last in its `wait` state by at least `lo`.

Invariant Axiom 3.2

$$\{\forall \mathtt{i}\colon \mathtt{PC(i)} = \mathtt{cs} \supset |\mathtt{PC(i)} = \mathtt{wait}| + \mathtt{lo} \leq |\mathtt{PC(i)} = \mathtt{try}|\}.$$

Note that we require

$$\mathtt{hi} < \mathtt{lo}. \tag{3.1}$$

We now informally argue that the `mutex` protocol guarantees each process mutually exclusive access to its critical section. We prove that the protocol satisfies a number of invariants that lead to the statement of mutual exclusion. Some of these invariants are proved directly by induction on the behaviors satisfying the `mutex` program, whereas others are derived consequences of previously proved invariants. The first invariant asserts that whenever the value of `x` is `i`, then the last time `x` was `0` was no later than the last time `PC(i) = try` was true.

Invariant 3.3

$$\{\forall \mathtt{i}\colon \mathtt{x} = \mathtt{i} \supset |\mathtt{PC(i)} = \mathtt{try}| \leq |\mathtt{x} = \mathtt{0}|\}.$$

Proof. The invariant is established by induction over a possible program trace. Initially, the antecedent is false. The action `Try(j)`, for any `j`, trivially preserves the invariant since it ensures that `x` is equal to `0` thus falsifying the antecedent. Stated as a Hoare formula, we have

$$\{\mathbf{true}\}\ \mathtt{Try(i)}\ \{\mathtt{x} = \mathtt{0}\}.$$

The action `Wait(j)`, for $\mathtt{i} \neq \mathtt{j}$, preserves the invariant by falsifying the antecedent. For the `Wait(i)` action, we get two cases. If $\mathtt{x} = \mathtt{0}$ holds in the precondition, we have by **(step1)** that

$$\{\mathtt{x} = \mathtt{0}\}\ \mathtt{Wait(i)}\ \{|\mathtt{PC(i)} = \mathtt{try}| = |\mathtt{x} = \mathtt{0}| = |\mathbf{true}|\},$$

and hence the conclusion. If $\mathtt{x} = \mathtt{0}$ is false in the precondition, we have by **(step1)** and **(step2)** that:

$$\begin{gathered}\{\mathtt{x} \neq \mathtt{0} \wedge X = |\mathtt{x} = \mathtt{0}|\} \\ \mathtt{Wait(i)} \\ \{|\mathtt{PC(i)} = \mathtt{try}| = |\mathbf{true}| \leq (X + |\mathbf{true}|) = |\mathtt{x} = \mathtt{0}|\}\end{gathered}$$

The action $\texttt{Cs(j)}$, for $\texttt{i} \neq \texttt{j}$ preserves the invariant by falsifying the antecedent since $\texttt{x} = \texttt{j}$ in this case. For the action $\texttt{Cs(i)}$, we have by **(step2)** that

$$\begin{array}{c} \{Y = |\texttt{PC(i)} = \texttt{try}| \leq |\texttt{x} = \texttt{0}| = X\} \\ \texttt{Cs(i)} \\ \{(Y + |\mathbf{true}|) = |\texttt{PC(i)} = \texttt{try}| \leq |\texttt{x} = \texttt{0}| = (X + |\mathbf{true}|)\} \end{array} .$$

■

The next invariant is another obvious consequence of the $\texttt{mutex}$ program. It asserts that when a process is in its $\texttt{try}$ state, it was last in its $\texttt{init}$ state no later than when $\texttt{x}$ was last equal to $\texttt{0}$.

Invariant 3.4

$$\{\forall \texttt{i} \colon \texttt{PC(i)} = \texttt{try} \supset |\texttt{x} = \texttt{0}| \leq |\texttt{PC(i)} = \texttt{init}|\}$$

Proof. By induction. Initially, the antecedent is false. For a $\texttt{Try(j)}$ action, for any $\texttt{j}$, by **(step1)**, we have

$$\{\mathbf{true}\}\ \texttt{Try(j)}\ \{|\texttt{x} = \texttt{0}| = |\mathbf{true}| \leq |\texttt{PC(i)} = \texttt{init}|\}.$$

In the case of a $\texttt{Wait(j)}$ action, for any $\texttt{j}$, we have two cases. If $\texttt{x} = \texttt{0}$ holds in the precondition then by **(step1)** and **(step2)**, we have

$$\begin{array}{c} \{\texttt{x} = \texttt{0} \wedge Y = |\texttt{PC(i)} = \texttt{init}|\} \\ \texttt{Wait(j)} \\ \{|\texttt{x} = \texttt{0}| = |\mathbf{true}| \leq (Y + |\mathbf{true}|) = |\texttt{PC(i)} = \texttt{init}|\} \end{array} .$$

Otherwise, by **(step2)** we have

$$\begin{array}{c} \{X = |\texttt{x} = \texttt{0}| \leq Y = |\texttt{PC(i)} = \texttt{init}|\} \\ \texttt{Wait(j)} \\ \{|\texttt{x} = \texttt{0}| = (X + |\mathbf{true}|) \leq (Y + |\mathbf{true}|) = |\texttt{PC(i)} = \texttt{init}|\} \end{array} .$$

The $\texttt{Cs(j)}$ action, for any $\texttt{j}$, similarly preserves the invariant by using **(step2)** to add the same delay to both sides of the precondition inequality yielding the postcondition invariant. ■

The next lemma is the main step in the proof. It asserts that when process $\texttt{i}$ is in its critical section and $\texttt{x}$ is equal to $\texttt{k}$, then no process $\texttt{j}$ is in its try state.

Invariant 3.5

$$\{\forall \mathtt{j}, \mathtt{i}\colon \mathtt{PC(i)} = \mathtt{cs} \wedge \mathtt{x} = \mathtt{i} \supset \mathtt{PC(j)} \neq \mathtt{try}\}.$$

Proof. Suppose $\mathtt{PC(j)} = \mathtt{try}$, then by Invariants 3.1 and 3.4, we have

$$|\mathtt{x} = \mathtt{0}| \leq |\mathtt{PC(j)} = \mathtt{init}| \leq \mathtt{hi}.$$

By Invariants 3.3 and 3.2, this yields

$$\mathtt{lo} \leq |\mathtt{PC(i)} = \mathtt{try}| \leq |\mathtt{x} = \mathtt{0}| \leq \mathtt{hi}$$

since $|\mathtt{PC(i)} = \mathtt{wait}| \geq 0$. Hence, by the Inequality (3.1), we get a contradiction. ∎

As a consequence of the above invariant, if process `i` is in its critical section and `x` is `i`, then no other process can change the value of `x`.

Invariant 3.6

$$\{\forall \mathtt{i}\colon \mathtt{PC(i)} = \mathtt{cs} \supset \mathtt{x} = \mathtt{i}\}.$$

Proof. By induction. Initially, the antecedent is false. The `Try` action is the only action that can falsify the invariant, but by Invariant 3.5, $(\forall \mathtt{j}\colon \mathtt{PC(j)} \neq \mathtt{try})$, so that either $\mathtt{PC(i)} \neq \mathtt{cs}$ is true and the invariant trivially holds, or no `Try` action is enabled. ∎

Invariant 3.7

$$\{\forall \mathtt{i}, \mathtt{j}\colon \mathtt{PC(i)} = \mathtt{cs} \wedge \mathtt{PC(j)} = \mathtt{cs} \supset \mathtt{i} = \mathtt{j}\}$$

Proof. Follows trivially from Invariant 3.6. ∎

The verification of the above proof of mutual exclusion was carried out using PVS. Since it is a fairly simple example, it does not seriously stress the formalism we have described or the verification system. The example of the railroad crossing is somewhat more difficult.

3.1 Verifying the Mutual Exclusion Protocol in PVS

The theory `mutex` below describes the mutual exclusion protocol in terms of behaviors and checks the above invariants to be true. It is worth noting the correspondence between the informal description above and the formalization in PVS. The actions `Try`, `Wait`, and `Cs` in the above description are defined below as `Check`, `Ask`, and `Doublecheck`, respectively. The definitions of `Check`, `Ask`, and `Doublecheck` actions are slightly different from their definitions above. The definitions below use existential quantification to express the nondeterministic selection of the process in the mutex program above. The mutex `program` is defined as a predicate on behaviors asserting that the initial condition is satisfied and every two adjacent pairs of states in the behavior satisfy one of the `Check`, `Ask`, or `Doublecheck` actions. The Invariant Axioms 3.1 and 3.2 correspond to the axioms `TI1` and `TI2`, respectively. The Invariants 3.3, 3.5, and 3.6 correspond to the Lemmas `Inv1`, `Inv2_2`, and `Inv3`, respectively. Lemma `Inv1_2` differs slightly from Invariant 3.4 in that it additionally applies the timing constraint `TI1`. Invariant 3.6 corresponds to the theorem `safety` below. Note that the Inequality 3.1 is assumed as an antecedent in `safety` and is not written as a separate axiom.

---

mutex: THEORY
  BEGIN

  USING time, since, since_thms
  process: TYPE = posnat
  i, j, k: VAR process
  aa, bb, cc: VAR behavior0
  s, s_0, s_1: VAR state
  pp, qq, rr: VAR statepred
  program_counter: TYPE = {init, try, wait, cs}
  x: [state → nat]
  PC: [state → [process → program_counter]]
  Check(s_0, s_1):
    bool
      =
      ($\exists$ i:
        PC(s_0)(i) = init
          $\land$ $x(s_0) = 0 \land x(s_1) = x(s_0) \land$ PC(s_1) = PC(s_0) WITH [(i) := try])

$\mathrm{Ask}(s_0, s_1)$:
  bool
    =
    ($\exists$ i:
      $\mathrm{PC}(s_0)(i) = \mathrm{try} \wedge x(s_1) = i \wedge \mathrm{PC}(s_1) = \mathrm{PC}(s_0)$ WITH $[(i) := \mathrm{wait}]$)

$\mathrm{Doublecheck}(s_0, s_1)$:
  bool
    =
    ($\exists$ i:
      $\mathrm{PC}(s_0)(i) = \mathrm{wait}$
        $\wedge\ x(s_0) = i \wedge x(s_1) = x(s_0) \wedge \mathrm{PC}(s_1) = \mathrm{PC}(s_0)$ WITH $[(i) := \mathrm{cs}]$)

$\mathrm{Stutter}(s_0, s_1)$: bool $= (\forall\, i\colon \mathrm{PC}(s_1)(i) = \mathrm{PC}(s_0)(i)) \wedge x(s_1) = x(s_0)$

m, n: VAR nat
program(aa):
  bool
    =
    $(\mathrm{Init}(\lambda\, s\colon x(s) = 0 \wedge (\forall\, i\colon \mathrm{PC}(s)(i) = \mathrm{init}))(\mathrm{aa})$
      $\wedge$
      $(\forall\, n$:
        $\mathrm{Check}(\mathrm{aa}(n),\ \mathrm{aa}(n+1))$
          $\vee\ \mathrm{Ask}(\mathrm{aa}(n),\ \mathrm{aa}(n+1))$
           $\vee\ \mathrm{Doublecheck}(\mathrm{aa}(n),\ \mathrm{aa}(n+1)) \vee \mathrm{Stutter}(\mathrm{aa}(n),\ \mathrm{aa}(n+1))))$

hi, lo: time
TI1:
  AXIOM
   $\mathrm{Inv}(\lambda\, s$:
        $(\forall\, i$:
          $\mathrm{PC}(s)(i) = \mathrm{try} \supset \mathrm{since}(\lambda\, s_0\colon \mathrm{PC}(s_0)(i) = \mathrm{init})(s) \le \mathrm{hi}))(\mathrm{aa})$

TI2:
  AXIOM
   $\mathrm{Inv}(\lambda\, s$:
        $(\forall\, i$:
          $\mathrm{PC}(s)(i) = \mathrm{cs}$
            $\supset \mathrm{since}(\lambda\, s_0\colon \mathrm{PC}(s_0)(i) = \mathrm{wait})(s) + \mathrm{lo}$
             $\le \mathrm{since}(\lambda\, s_0\colon \mathrm{PC}(s_0)(i) = \mathrm{try})(s)))(\mathrm{aa})$

Inv1:
  LEMMA

program(aa)
$\supset$
Inv(λ s:
($\forall$ i:
$x(s) = i$
$\supset$ since(λ s_0: PC$(s_0)(i)$ = try)(s)
$\leq$ since(λ s_0: $x(s_0) = 0)(s)$))(aa)

Inv2:
LEMMA
program(aa)
$\supset$
Inv(λ s:
($\forall$ i:
PC$(s)(i)$ = try
$\supset$ since(λ s_0: $x(s_0) = 0)(s)$
$\leq$ since(λ s_0: PC$(s_0)(i)$ = init)(s)))(aa)

Inv2_2:
LEMMA
program(aa) $\wedge$ hi $<$ lo
$\supset$
Inv(λ s: ($\forall$ i: PC$(s)(i)$ = cs $\wedge$ $x(s) = i$ $\supset$ ($\forall$ j: PC$(s)(j)$ $\neq$ try)))(aa)

Inv3:
LEMMA
program(aa) $\wedge$ hi $<$ lo $\supset$ Inv(λ s: ($\forall$ i: PC$(s)(i)$ = cs $\supset$ $x(s) = i$))(aa)

safety:
THEOREM
program(aa) $\wedge$ hi $<$ lo
$\supset$ Inv(λ s: ($\forall$ i, j: PC$(s)(i)$ = cs $\wedge$ PC$(s)(j)$ = cs $\supset$ $i = j$))(aa)

END mutex

The relevant PVS transcripts are in Appendix B. The PVS proofs roughly correspond to a detailed rendering of the informal argument in the previous section. There is some additional overhead for dealing with state in an explicit manner as compared with the implicit treatment of state in the informal proofs. The extra work is in unfolding the definition of `Inv` in each application of a lemma whose conclusion is an invariant assertion. There is also some overhead for the fact that the time domain has been extended with an ∞ element since the arithmetic decision procedures of PVS cannot be directly applied to this domain. Some of this overhead has been reduced by defining suitable PVS proof strategies that can encapsulate the pattern of inferences involved. It seems likely that the overhead can be considerably reduced, and furthermore, that many of the invariance arguments can be carried out in a fully automated manner.

Chapter 4

Verifying the Safety and Utility of a Railroad Crossing Controller

We consider a railroad crossing system consisting of a gate, a controller, and an arbitrary number of trains. Relative to the crossing, any train is either `safe`, `approaching`, or `crossing`. A train goes from being `safe` to `approaching`, then to `crossing` and back to being `safe`. The controller senses when a train starts approaching and sets a signal to `lower` within D time units. Within G time units after the signal is set to `lower`, the gate is either down or starts moving down. Once a gate starts moving down, it is down within L time units. The gate then starts moving up only when no train is either approaching or in the crossing. No bounds are placed on the time it takes for the gate to start moving up or to be up once it has started moving up. The main correctness criterion for the system is that when a train is in the crossing, the gate must be down. To ensure this, we must assume that a train cannot go from `safe` to `crossing` within $D + G + L$ time units. Let `Approaching` define a state predicate that holds if there is some train that is approaching in the given state. Similarly, `Crossing` is a state predicate that holds if some train is crossing, and the state predicate `Safe` is defined to hold when no train is `approaching` or `crossing`.

The railroad crossing system can be written as the following Unity program:

```
Program RR

   declare train : [nat -> {safe, approaching, crossing}],
           signal : {lower, raise},
           gate : {up, down, moving_up, moving_down}
   initially ⟨∀ p:  train(p)  =  safe⟩
   assign
    ⟨[]p :
       train(p) := approaching, if train(p) = safe
     [] train(p) := crossing,    if train(p) = approaching
     [] train(p) := safe,        if train(p) = crossing
    ⟩
     [] signal := lower,         if Approaching ∧ signal ≠ lower
     [] signal := raise,         if Safe ∧ signal = lower
     [] gate := moving_down,     if signal = lower ∧ gate ≠ down
     [] gate := down,            if gate = moving_down
     [] gate := moving_up,       if signal = raise ∧ gate ≠ up
     [] gate := up,              if gate = moving_up
end {RR}
```

The first three guarded assignment statements represent the transitions of each train as it goes from being a safe distance from the crossing to approaching the crossing, then to actually enter the crossing, and finally exiting the crossing so that is once again safe. In the formalization above, there are an arbitrary number of trains and there are no constraints on how they behave relative to one another. When a train starts to approach the crossing, it triggers a sensor that causes the signal to be set to `lower` in the fourth transition. In each of the two signal transitions, the second guard is not really needed. The fifth transition raises the signal when there is no train either approaching or in the crossing. The remaining transitions deal with the gate. If the signal is `lower` and the gate is not already `down`, then the state of the gate is set to `moving_down`. The gate eventually goes from `moving_down` to `down`. If the signal is `raise` and the gate is not already `up`, then the state of the gate is set to `moving_up`. From `moving_up`, the gate eventually goes to being `up`.

Certain timing constraints are associated with the actions of the trains, the signal, and the gate. These constraints are expressed as invariants. The first of these expresses the constraint that within D time units of when any train starts approaching (*i.e.*, when the value of `Approaching` goes from false to true), the signal goes to `lower`.

Invariant Axiom 4.1

$$\{\texttt{Approaching} \wedge \texttt{signal} \neq \texttt{lower} \supset |\neg\texttt{Approaching}| \leq D\}.$$

The second timing invariant similarly bounds the delay between the time when the signal is set to `lower` and the gate goes to `moving_down` by G time units. It asserts that if the signal is lower, then either the gate is `down`, or it is `moving_down`, or it is fewer than G time units since the signal was not at `lower`.

Invariant Axiom 4.2

$$\begin{array}{l} \{\texttt{signal} = \texttt{lower} \supset \\ \qquad \texttt{gate} = \texttt{down} \\ \quad \vee \;\; \texttt{gate} = \texttt{moving\_down} \\ \quad \vee \;\; |\texttt{signal} \neq \texttt{lower}| \leq G\}. \end{array}$$

The third timing invariant bounds the delay between when the gate starts `moving_down` and when it is `down` by L time units.

Invariant Axiom 4.3

$$\{\texttt{gate} = \texttt{moving\_down} \supset |\texttt{gate} \neq \texttt{moving\_down}| \leq L\}.$$

The pattern in the above statements should be clear by now. To state that x is an upper bound on the time that the system can spend in a state satisfying p, we write $p \supset |\neg p| \leq x$. The last timing invariant places an upper bound on the time that a train can spend in its `approaching` state. It asserts that if a train is in its `crossing` state, then the time since the train was last `safe` exceeds that time since it was last `approaching` by over $D + G + L$ time units.

Invariant Axiom 4.4

$$\begin{aligned} \{\forall \texttt{Tr}\colon \texttt{train(Tr)} = \texttt{crossing} \supset \\ &|\texttt{train(Tr)} = \texttt{approaching}| + D + G + L \\ < \quad &|\texttt{train(Tr)} = \texttt{safe}|\}. \end{aligned}$$

4.1 The Railroad Crossing in PVS

The program describing the trains, the controller, and the gate is expressed in PVS with the three train actions (labeled `tr_action`), the two controller actions (`cont_action`), and the four `gate` actions. The Invariant Axioms 4.1, 4.2, 4.3, and 4.4 are represented by `TI1`, `TI2`, `TI3`, and `TI4`.

```
railroad: THEORY
  BEGIN

  USING time, since, since_thms
  USING time
  Train: TYPE
  Tr, Tr1, Tr2, Tr3: VAR Train
  Tr_state: TYPE  = {safe, approaching, crossing}
  train: [Train → [state → Tr_state]]
  s, s0, s1: VAR state
  Approaching(s): bool  = (∃ Tr: (train(Tr)(s)  = approaching))

  Crossing(s): bool  = (∃ Tr: (train(Tr)(s)  = crossing))

  Gate: TYPE  = {up, down, moving_up, moving_down}
  gate: [state → Gate]
  Signal: TYPE  = {lower, raise}
  signal: [state → Signal]
  tr_action1:
   Action
     =
     (λ s0, s1:
       (∃ Tr:
         train(Tr)(s0)  = safe
           ∧ train(Tr)(s1)  = approaching
             ∧ (∀ Tr1: Tr1  ≠ Tr  ⊃ train(Tr1)(s1)  = train(Tr1)(s0)))
         ∧ signal(s1)  = signal(s0)  ∧ gate(s1)  = gate(s0))
```

tr_action2:
 Action
 =
 (λ s_0, s_1:
 ($\exists$ Tr:
 train(Tr)(s_0) = approaching
 $\wedge$ train(Tr)(s_1) = crossing
 $\wedge$ ($\forall$ Tr1: Tr1 $\neq$ Tr $\supset$ train(Tr1)(s_1) = train(Tr1)(s_0)))
 $\wedge$ signal(s_1) = signal(s_0) $\wedge$ gate(s_1) = gate(s_0))

tr_action3:
 Action
 =
 (λ s_0, s_1:
 ($\exists$ Tr:
 train(Tr)(s_0) = crossing
 $\wedge$ train(Tr)(s_1) = safe
 $\wedge$ ($\forall$ Tr1: Tr1 $\neq$ Tr $\supset$ train(Tr1)(s_1) = train(Tr1)(s_0)))
 $\wedge$ signal(s_1) = signal(s_0) $\wedge$ gate(s_1) = gate(s_0))

cont_action1:
 Action
 =
 (λ s_0, s_1:
 Approaching(s_0)
 $\wedge$ signal(s_0) $\neq$ lower
 $\wedge$ signal(s_1) = lower
 $\wedge$ ($\forall$ Tr: train(Tr)(s_1) = train(Tr)(s_0)) $\wedge$ gate(s_1) = gate(s_0))

cont_action2:
 Action
 =
 (λ s_0, s_1:
 $\neg$ Approaching(s_0)
 $\wedge$ $\neg$ Crossing(s_0)
 $\wedge$ signal(s_0) = lower
 $\wedge$ signal(s_1) = raise
 $\wedge$ ($\forall$ Tr: train(Tr)(s_1) = train(Tr)(s_0))
 $\wedge$ gate(s_1) = gate(s_0))

gate_action1:
 Action
 =

```
    (λ s_0, s_1:
       signal(s_0) = lower
         ∧ gate(s_0) ≠ down
           ∧ gate(s_1) = moving_down
             ∧ signal(s_1) = signal(s_0)
               ∧ (∀ Tr: train(Tr)(s_1) = train(Tr)(s_0)))

gate_action2:
  Action
    =
    (λ s_0, s_1:
       gate(s_0) = moving_down
         ∧ gate(s_1) = down
           ∧ signal(s_1) = signal(s_0)
             ∧ (∀ Tr: train(Tr)(s_1) = train(Tr)(s_0)))

gate_action3:
  Action
    =
    (λ s_0, s_1:
       signal(s_0) = raise
         ∧ gate(s_0) ≠ up
           ∧ gate(s_1) = moving_up
             ∧ signal(s_1) = signal(s_0)
               ∧ (∀ Tr: train(Tr)(s_1) = train(Tr)(s_0)))

gate_action4:
  Action
    =
    (λ s_0, s_1:
       gate(s_0) = moving_up
         ∧ gate(s_1) = up
           ∧ signal(s_1) = signal(s_0)
             ∧ (∀ Tr: train(Tr)(s_1) = train(Tr)(s_0)))

stutter:
  Action
    =
    (λ s_0, s_1:
       gate(s_1) = gate(s_0)
         ∧ signal(s_1) = signal(s_0) ∧ (∀ Tr: train(Tr)(s_1) = train(Tr)(s_0)))

i, j, k, m, n: VAR nat
pp, qq, rr: VAR statepred
```

aa, bb, cc: VAR behavior
program_init(s): bool = ($\forall$ Tr: train(Tr)(s) = safe)

program_transition(aa):
bool
=
($\forall$ i:
tr_action1(aa(i), aa(i + 1))
$\vee$ tr_action2(aa(i), aa(i + 1))
$\vee$ tr_action3(aa(i), aa(i + 1))
$\vee$ cont_action1(aa(i), aa(i + 1))
$\vee$ cont_action2(aa(i), aa(i + 1))
$\vee$ gate_action1(aa(i), aa(i + 1))
$\vee$ gate_action2(aa(i), aa(i + 1))
$\vee$ gate_action3(aa(i), aa(i + 1))
$\vee$ gate_action4(aa(i), aa(i + 1))
$\vee$ stutter(aa(i), aa(i + 1)))

program(aa): bool = (program_init(aa(0)) $\wedge$ program_transition(aa))

non_negatives: TYPE = $\{x$: rational $\mid x \geq 0\}$
D, G, L: non_negatives
TI1:
AXIOM
Inv(λ s:
Approaching(s) $\wedge$ signal(s) $\neq$ lower
$\supset$ since(λ s_0: $\neg$ Approaching(s_0))(s) $\leq$ D)(aa)

TI2:
AXIOM
Inv(λ s:
signal(s) = lower
$\supset$ gate(s) = down
$\vee$ gate(s) = moving_down
$\vee$ since((λ s_0: signal(s_0) $\neq$ lower))(s) $\leq$ G)(aa)

TI3:
AXIOM
Inv(λ s:
gate(s) = moving_down
$\supset$ since((λ s_0: gate(s_0) $\neq$ moving_down))(s) $\leq$ L)(aa)

TI4:
AXIOM

```
    Inv(λ s:
            (∀ Tr:
              train(Tr)(s) = crossing
                 ⊃ since(λ s0: train(Tr)(s0) = approaching)(s) + (D + G + L)
                   < since(λ s0: train(Tr)(s0) = safe)(s)))(aa)

  Unsafe(s): bool = (Approaching(s) ∨ Crossing(s))

  Safe(s): bool = ¬ Unsafe(s)

  R, M, U: non_negatives
  TI5:
    AXIOM Inv(λ s: signal(s) = lower ∧ Safe(s) ⊃ since(Unsafe)(s) ≤ R)(aa)

  TI6:
    AXIOM
      Inv(λ s:
            signal(s) = raise
               ⊃
             (gate(s) = up
                ∨ gate(s) = moving_up
                  ∨ since(λ s0: signal(s0) = lower)(s) ≤ M))(aa)

  TI7:
    AXIOM
      Inv(λ s:
            gate(s) = moving_up
               ⊃ since(λ s0: gate(s0) ≠ moving_up)(s) ≤ U)(aa)

  END railroad
```

4.2 Safety Proof for the Railroad Crossing

Before we proceed to the proof of safety for the railroad crossing controller, we note some important lemmas regarding invariants and the behavior of the *since* operator but omit their proofs. These lemmas are proved to be true over rooted behaviors. These lemmas are contained in the theory `since_thms`. The variable P and Q in the statements below range over state predicates.

Lemma 4.2.1

$$\textbf{invariant}\{P \supset Q\} \Rightarrow \textbf{invariant}\{P\} \Rightarrow \textbf{invariant}\{Q\}.$$

Invariant 4.5

$$\{|(|P| \leq x)| \leq y \supset |P| \leq x + y\}.$$

Lemma 4.2.2

$$\textbf{invariant}\{P \supset Q\} \Rightarrow \textbf{invariant}\{|Q| \leq |P|\}.$$

Invariant 4.6

$$\{|P \vee Q| \leq \min(|P|, |Q|)\}.$$

Invariant 4.7

$$\{|P \vee Q| = |P| \vee |P \vee Q| = |Q|\}.$$

Invariant 4.8

$$\{|P \wedge Q| = |P| \vee |P \wedge \neg Q| = |P|\}.$$

Invariant 4.9

$$\{\max(|P|, |Q|) \leq |P \wedge Q|\}.$$

Invariant 4.10

$$\{|P| \leq |P \wedge Q|\}.$$

Invariant 4.11

$$\{|Q| \leq |P \wedge Q|\}.$$

The next stage in our exposition of the railroad crossing is the proof of safety, that is, whenever a train is in the crossing, the gate is down. The proof proceeds by proving various invariance properties of the railroad crossing system. Some of these invariance properties are proved directly from the program while the remaining are proved as consequences of already proved invariants.

The first invariant `Inv1` below is proved directly from the program. This invariant expresses the simple observation that after a train `Tr` makes the transition from `safe` to `approaching`, the time since the train was `safe` does not exceed the time since no train was approaching.

Invariant 4.12

$\{\forall \mathtt{Tr}\colon \mathtt{train(Tr)} = \mathtt{approaching} \supset |\mathtt{train(Tr)} = \mathtt{safe}| \leq |\neg\mathtt{Approaching}|\}.$

Proof. By induction on a behavior satisfying the given program. ∎

The second invariant is a consequence of Invariants 4.12 and 4.1. It asserts that for any train `Tr`, if `Tr` is approaching and the `signal` has not yet been set to `lower`, then the time since `Tr` was last safe is at most D.

Invariant 4.13

$$\begin{array}{ll} & \{\forall \mathtt{Tr}\colon \mathtt{train(Tr)} = \mathtt{approaching} \\ & \quad \wedge \;\; \mathtt{signal} \neq \mathtt{lower} \\ \supset & \qquad |\mathtt{train(Tr)} = \mathtt{safe}| \leq D\}. \end{array}$$

Proof. By Invariant Axiom 4.1 and the definition of `Approaching`, we get $|\neg\mathtt{Approaching}| \leq D$ under the given antecedents. This together with Invariant 4.12 yields that $|\mathtt{train(Tr)} = \mathtt{approaching}| \leq D$. ∎

The next invariant is somewhat harder to prove. It asserts that when a train `Tr` is approaching and the `signal` is set to `lower`, then either the `gate` is `down`, or `moving_down`, or the time since `Tr` was last in its `safe` state is at most $D + G$.

Invariant 4.14

$$\begin{array}{ll} & \{\forall \mathtt{Tr}\colon \mathtt{train(Tr)} = \mathtt{approaching} \\ & \quad \wedge \;\; \mathtt{signal} = \mathtt{lower} \\ \supset & \qquad \mathtt{gate} = \mathtt{down} \\ & \quad \vee \;\; \mathtt{gate} = \mathtt{moving\_down} \\ & \quad \vee \;\; |\mathtt{train(Tr)} = \mathtt{safe}| \leq D + G\}. \end{array}$$

Proof. We already have by Invariant Axiom 4.2 that when `signal = lower`, then either `gate = down`, or `gate = moving_down`, or

$$|\texttt{signal} \neq \texttt{lower}| \leq G.$$

We therefore need only concentrate on the latter case. We have by Invariant 4.8 that either

$$|\texttt{signal} \neq \texttt{lower}| = |\texttt{signal} \neq \texttt{lower} \wedge \texttt{Approaching}|$$

or

$$|\texttt{signal} \neq \texttt{lower}| = |\texttt{signal} \neq \texttt{lower} \wedge \neg\texttt{Approaching}|.$$

In the latter case, we have by `Inv1` that

$$\begin{array}{cl}
 & \texttt{train(Tr)} = \texttt{safe} \\
\leq & \{\text{Invariant 4.12}\} \\
 & |\neg\texttt{Approaching}| \\
\leq & \{\text{Invariant 4.9}\} \\
 & |\texttt{signal} \neq \texttt{lower} \wedge \neg\texttt{Approaching}| \\
= & \{\text{Assumption}\} \\
 & |\texttt{signal} \neq \texttt{lower}| \\
\leq & \{\text{Invariant 4.2}\} \\
 & G \\
\leq & \{D \geq 0\} \\
 & D + G.
\end{array}$$

Hence the conclusion.

In the former case, we have by Invariant Axiom 4.1 and Lemma 4.2.2 that

$$\begin{array}{cl}
 & |(|\neg\texttt{Approaching}| \leq D)| \\
\leq & \{\text{By Invariant Axiom 4.1 and Lemma 4.2.2}\} \\
 & |\texttt{signal} \neq \texttt{lower} \wedge \texttt{Approaching}| \\
\leq & \{\text{By Assumption}\} \\
 & G.
\end{array}$$

By Invariant 4.12 and Invariant 4.5, we therefore have the desired result

$$|\texttt{train(Tr)} = \texttt{safe}| \leq D + G.$$

$\blacksquare$

The next invariant follows directly from the program. It asserts that if the signal is currently at `lower` and the gate is not `down`, then the gate was last `down` at or prior to when the signal was last different from `lower`.

Invariant 4.15

$$\begin{array}{ll} \{ & \mathtt{signal} = \mathtt{lower} \\ \wedge & \mathtt{gate} \neq \mathtt{down} \\ \supset & |\mathtt{signal} \neq \mathtt{lower}| \leq |\mathtt{gate} = \mathtt{down}|\}. \end{array}$$

The following invariant is also easily derived from the program. It asserts that the only way that a train can make the transition from not `approaching` to `approaching` is by making the transition from `safe` to `approaching`.

Invariant 4.16

$$\begin{array}{l} \{\forall \mathtt{Tr}\colon \mathtt{train(Tr)} = \mathtt{approaching} \supset \\ \quad |\mathtt{train(Tr)} = \mathtt{safe}| = |\mathtt{train(Tr)} \neq \mathtt{approaching}|\} \end{array}$$

The next invariant is the main step in proving the safety of the railroad crossing controller. It asserts that when a train `Tr` is approaching, the `signal` is set to `lower` but the gate is still `moving_down`, then the time since the train `Tr` was last safe does not exceed $D + G + L$.

Invariant 4.17

$$\begin{array}{ll} \{\forall \mathtt{Tr}\colon & \\ & \mathtt{train(Tr)} = \mathtt{approaching} \\ \wedge & \mathtt{signal} = \mathtt{lower} \\ \wedge & \mathtt{gate} = \mathtt{moving\_down} \\ \supset & |\mathtt{train(Tr)} = \mathtt{safe}| \leq D + G + L\} \end{array}$$

Proof. The proof is similar to that of Invariant 4.14. By Invariant Axiom 4.3, we have $|\mathtt{gate} \neq \mathtt{moving\_down}| \leq L$. So by Invariant 4.8, we get two cases according to

$$|\mathtt{gate} \neq \mathtt{moving\_down} \wedge \mathtt{train(tr)} = \mathtt{approaching}| \leq L$$

or

$$|\texttt{gate} \neq \texttt{moving\_down} \wedge \texttt{train(tr)} \neq \texttt{approaching}| \leq L.$$

Since in the latter case, the desired conclusion easily follows from Invariant 4.16 and 4.5, we concentrate on the former case. Let A label the formula $\texttt{gate} \neq \texttt{moving\_down} \wedge \texttt{train(tr)} = \texttt{approaching}$.

Again we apply Invariant 4.8 to get two cases according to whether we have either

$$|A \wedge \texttt{signal} = \texttt{lower}| \leq L$$

or

$$|A \wedge \texttt{signal} \neq \texttt{lower}| \leq L.$$

The latter case easily yields the desired conclusion by Invariant 4.13 and Invariant 4.5, so we concentrate on the former case. Let B label the formula $A \wedge \texttt{signal} = \texttt{lower}$.

We once again apply Invariant 4.8 to get two cases according to whether we additionally have either

$$|B \wedge \texttt{gate} = \texttt{down}| \leq L$$

or

$$|B \wedge \texttt{gate} \neq \texttt{down}| \leq L.$$

The latter case easily follows from Invariant 4.14 and 4.5.

In the former case, we have by Invariant 4.15 that

$$|\texttt{signal} \neq \texttt{lower}| \leq L.$$

From this, an argument similar to the one in the proof of Invariant 4.14 yields the desired conclusion. ■

It seems puzzling that the proof of Invariant 4.17 should be so complicated since it ought to be possible prove the invariant by simply adding up the train-signal delay D, the signal-gate delay G, and the delay L associated with the lowering of the gate. However, in order to add up the delays in this manner, we have to assume that the intermediate states exist and this need not always be the case. Consider the proof of Invariant 4.14. Having derived that $|\texttt{signal} \neq \texttt{lower}| \leq G$, we cannot assume that there will be a state in the past where $\texttt{signal} \neq \texttt{lower}$ and $\texttt{train(Tr)} = \texttt{approaching}$ since it is possible that when train `Tr` is first approaching, the signal is already set to `lower` due to a previous train. In the case of Invariant 4.17, not only could the signal already be set to lower, but the gate could already be moving down thanks to a previous train. The kind of case analysis involved in these proofs is not only necessary but is also easily overlooked in an informal argument.

The final invariant establishes the safety of railroad crossing by proving that if there is a train in the crossing, the gate is down.

Invariant 4.18

$$\{\texttt{Crossing} \supset \texttt{gate} = \texttt{down} \wedge \texttt{signal} = \texttt{lower}\}.$$

Proof. The proof is by induction over the states in a behavior satisfying the program `RR`. The main step is to demonstrate that when `Crossing` goes from false to true, that is, in the precondition of that action where a train `Tr` goes from `approaching`to `crossing`, the time since that train was `safe` is more than $D + G + L$ by Invariant Axiom 4.4. Then by Invariant 4.17, the gate must be down. ■

4.3 Safety Proof in PVS

The Invariants 4.12, 4.14, 4.15, 4.16, and 4.17 in the informal proof above correspond to the Lemmas `Inv1`, `Inv3`, `Inv4_1`, `Inv4_3`, and `Inv4`, in the theory below. Invariant 4.18 corresponds to Lemma `Safety` below.

```
railroad_pr: THEORY
  BEGIN

  USING time, railroad, since, since_thms
  Tr, Tr1, Tr2, Tr3: VAR Train
  s, s0, s1: VAR state
  aa, bb, cc: VAR behavior0
  pp, qq, rr: VAR statepred
  rt, rt1, rt2, rt3: VAR time
  i, j, k, m, n: VAR nat
  Inv1:
   LEMMA
    program(aa)
      ⊃
```

```
    Inv(λ s:
            (∀ Tr:
              train(Tr)(s) = approaching
                  ⊃ since(λ s0: train(Tr)(s0) = safe)(s)
                    ≤ since(λ s0: ¬ Approaching(s0))(s)))(aa)

Inv2:
  LEMMA
    program(aa)
        ⊃
    Inv(λ s:
            (∀ Tr:
              train(Tr)(s) = approaching ∧ signal(s) ≠ lower
                  ⊃ since(λ s0: train(Tr)(s0) = safe)(s) ≤ D))(aa)

Inv3:
  LEMMA
    program(aa)
        ⊃
    Inv(λ s:
            (∀ Tr:
              train(Tr)(s) = approaching ∧ signal(s) = lower
                  ⊃
                (gate(s) = down
                     ∨ gate(s) = moving_down
                       ∨ since(λ s0: train(Tr)(s0) = safe)(s)
                         ≤ (D + G))))(aa)

Inv4_1:
  LEMMA
    program(aa)
        ⊃
    Inv(λ s:
            signal(s) = lower ∧ gate(s) ≠ down
                ⊃ since(λ s0: signal(s0) ≠ lower)(s)
                  ≤ since(λ s0: gate(s0) = down)(s))(aa)

Inv4_3:
  LEMMA
    program(aa)
        ⊃
    Inv(λ s:
            (∀ Tr:
              train(Tr)(s) = approaching
```

$\supset$ since(λ s_0: train(Tr)(s_0) = safe)(s)
= since(λ s_0: train(Tr)(s_0) $\neq$ approaching)(s)))(aa)

Inv4:
LEMMA
program(aa)
$\supset$
Inv(λ s:
($\forall$ Tr:
train(Tr)(s) = approaching
$\wedge$ signal(s) = lower $\wedge$ gate(s) = moving_down
$\supset$ since(λ s_0: train(Tr)(s_0) = safe)(s)
$\leq$ (D + G + L)))(aa)

Safety_pred(s): bool = (Crossing(s) $\supset$ gate(s) = down $\wedge$ signal(s) = lower)

Safety: LEMMA program(aa) $\supset$ Inv(Safety_pred)(aa)

END railroad_pr

The relevant PVS output is in Appendix C.

4.4 Utility Proof of Railroad Crossing

While the above specification of the railroad crossing does constrain the gate to be down when a train is in the crossing, it does not require the gate to ever be raised. In order to show that the crossing is actually useful (and not just safe), we have to further demonstrate that the gate is eventually raised if no train has been approaching or crossing for a certain amount of time. We define the state predicate *Unsafe* to be true if some train is either crossing or approaching. Its negation, the predicate *Safe* is true when no train is either approaching or crossing. In order to demonstrate utility, we introduce three new non-negative time bounds R, M, and U. The quantity R bounds the time taken for the signal to go from `lower` to `raise` once

Safe becomes true. The quantity M bounds the time taken by the gate to react to the change in the value of signal from `lower` to `raise`. The quantity U bounds the time taken for the gate to eventually be up once it starts moving up. These timing assumptions are captured by the following invariant axioms.

Invariant Axiom 4.19

$$\{\texttt{signal} = \texttt{lower} \wedge \text{Safe} \supset |\text{Unsafe}| \leq R\}.$$

Invariant Axiom 4.20

$$\begin{array}{ll} \{\texttt{signal} = \texttt{raise} \supset & \\ & \texttt{gate} = \texttt{up} \\ \vee & \texttt{gate} = \texttt{moving\_up} \\ \vee & |\texttt{signal} = \texttt{lower}| \leq M\}. \end{array}$$

Invariant Axiom 4.21

$$\{\texttt{gate} = \texttt{moving\_up} \supset |\texttt{gate} \neq \texttt{moving\_up}| \leq U\}.$$

The proof of the utility property employs a lemma asserting the invariant that when the gate is not up, the time since signal was last equal to `lower` does not exceed the time since the gate was last in the up position.

Invariant 4.22

$$\{\texttt{gate} \neq \texttt{up} \supset |\texttt{signal} = \texttt{lower}| \leq |\texttt{gate} = \texttt{up}|\}.$$

Proof. The proof is by a straightforward induction on the states in a behavior satisfying the program `RR`. ∎

Invariant 4.23

$$\{|\texttt{signal} = \texttt{lower}| \leq X \supset |\text{Unsafe}| \leq R + X\}.$$

Proof. Applying Invariant 4.8 to the antecedent, it follows that either

$$|\texttt{signal} = \texttt{lower} \wedge \text{Unsafe}| \leq X$$

holds trivially yielding the desired conclusion, or

$$|\texttt{signal} = \texttt{lower} \wedge \text{Safe}| \leq X$$

is true. In the latter case, we have by Lemma 4.2.2 and Invariant 4.22 that

$$||\mathit{Unsafe}| \leq R| \leq X.$$

The desired conclusion follows easily by applying Invariant 4.5. ∎

The statement of utility asserts that if the gate is not up and the crossing is safe (*i.e.*, *Safe* is true), then the time since the crossing was last unsafe does not exceed the sum $R + M + U$ of the delays. Contrapositively stated, if the crossing has been in a *Safe* state for longer than $R + M + U$ time units, then the gate must be in the up position.

Invariant 4.24

$$\{\texttt{gate} \neq \texttt{up} \wedge \text{Safe} \supset$$
$$|\text{Unsafe}| \leq R + M + U\}.$$

Proof. When `signal = lower`, we have by Invariant Axiom 4.19 that $|\mathit{Unsafe}| \leq R$, and the theorem is easily proved.

When `signal = raise`, we have by Invariant Axiom 4.20 that either `gate = up` or `gate = moving_up` or $|\texttt{signal} = \texttt{lower}| \leq M$. The first possibility trivially yields the theorem by falsifying its antecedent, so we deal with the remaining two cases.

Case 1: Here `gate = moving_up` holds so that by Invariant Axiom 4.21, we have $|\texttt{gate} \neq \texttt{moving\_up}| \leq U$. By 4.8, either

$$|\texttt{gate} \neq \texttt{moving\_up} \wedge \texttt{signal} = \texttt{lower}| \leq U,$$

and the theorem follows from Invariants 4.9 and 4.23, or

$$|\texttt{gate} \neq \texttt{moving\_up} \wedge \texttt{signal} = \texttt{raise}| \leq U.$$

In the latter case, by Invariant Axiom 4.20 and Lemma 4.2.2, either $|\texttt{gate} = \texttt{up}| \leq U$ or $||\texttt{signal} = \texttt{lower} \leq M| \leq U$ holds. In the former case, we can apply Invariant 4.22 to get $|\texttt{signal} = \texttt{lower}| \leq U$ from which the theorem follows by Invariant 4.23. In the case when $||\texttt{signal} = \texttt{lower} \leq M| \leq U$, Invariant 4.23 easily yields the theorem.

Case 2: In this case, $|\texttt{signal} = \texttt{lower}| \leq M$ holds and Invariant 4.23 yields $|\mathit{Unsafe}| \leq R + M$, and hence the theorem. ∎

Chapter 5

Conclusions and Future Work

We have shown how nontrivial real-time protocols can be formalized and verified within the higher-order logic of PVS. The approach we have adopted in formalizing real-time state transition systems works equally well for systems where real time is irrelevant. We illustrated our approach with two examples: Fischer's real-time mutual exclusion protocol and a real-time railroad crossing controller. The key safety properties of these two systems were proved using the PVS interactive proof checker. While these proofs involve some extra overhead associated with the explicit treatment of state and behavior, they are largely straightforward. The `mutex` proofs were first checked in less than a day and the railroad crossing proofs took less than a week. Our work leaves a number of significant open issues:

1. How accurate is our model of concurrent real-time systems?
2. How does our approach relate to other similar work?
3. Is it possible to further mechanize the proof process?

We are using an interleaving model where concurrent actions are modeled by means of their sequential, nondeterministic interleaving. This means that if two actions A and B started and ended concurrently taking 1 time unit of real time, then their interleaving as either $A; B$ or $B; A$ is not going to be accurate with respect to the time taken by the individual actions since one of A or B will have to take place instantaneously so that the total time for both actions is still 1 time unit. Since interleaving is not a model of execution but a model of observation, this inaccuracy is acceptable. The system could have been observed so that action A was first observed to be completed, and instantaneously following that, B was observed to be completed. Note that in our approach we do not require that there be delays associated with each atomic action. The delay corresponding to a state transition is actually due to the delay in the observation and is not to be regarded as the time taken by the action. However, we do have to be careful to accurately model concurrent behavior in terms of interleaving.

It is not straightforward to relate our approach to real time to other similar approaches. While the various operators like *punch*, *duration*, and *since* are not directly inter-definable, proofs involving these operators can be easily translated from one notation to another. We have presented a useful collection of laws regarding the *since* operator. A similar collection of laws can be derived for these other operators.

An earlier formalization of the *since* operator required an ∞ element to be introduced to capture the initial value of $|p|$ for any p. This proved to be a significant obstacle to mechanization since it made it difficult to directly apply the PVS decision procedures for inequality reasoning. The present version eliminates the ∞ element so that the proofs are considerably more automated. The other obstacle to mechanization was posed by the explicit use of state, but we believe that this obstacle can be overcome by defining suitable PVS proof strategies. Some of the invariants we used in our proofs (*e.g.*, Invariant 4.16) are trivial consequences of the corresponding state machine description. It should be easy to extract these from a diagrammatic presentation of the state machine.

As illustrated by the examples in this report, proofs involving real time and discrete state transitions can be quite subtle. The task of verifying real-time protocols clearly requires mechanized assistance. We have presented an approach to the mechanized verification of real-time systems by encoding such systems in the higher-order logic of PVS, and using the PVS proof checker to construct proofs. We believe that our approach will easily scale up to even more complex systems.

Acknowledgments. Constance Heitmeyer of the Naval Research Laboratory and Ralph Jeffords of Locus, Inc. were instrumental in suggesting the examples and providing a number of comments and suggestions. John Rushby (SRI) provided advice on aspects of real-time systems. Sam Owre

(SRI) assisted with the use of PVS. Discussions with Jay Misra (University of Texas at Austin) and Jens Skakkebæk were extremely helpful.

Appendix A

Formalizing Real-Time Systems in PVS

A.1 Time and State in PVS

```
time : THEORY

BEGIN

  state: TYPE
  USING  sequences[state]

  statepred: TYPE = PRED[state]  %assertions
  Action: TYPE =  PRED[[state, state]]

  pp: VAR statepred
  action: VAR Action
  time : TYPE = {x : rational| x >= 0}

  Time: [state -> time]

  seq, seq1, seq2: VAR sequence[state]

  i, j,k: VAR nat

  nondec?(seq): bool =
```

```
    (FORALL i, j: i>j IMPLIES Time(seq(j)) <= Time(seq(i)))

  nonzeno?(seq): bool =
    (FORALL (x: rational): (EXISTS i: x < Time(seq(i))))

  behavior?(seq): bool = nondec?(seq) AND nonzeno?(seq)

  behavior : TYPE = (behavior?)

  aa, bb, cc: VAR behavior

  Inv(pp)(aa) : bool =
   (FORALL (n : nat) : pp(aa(n)))

  Init(pp)(aa) : bool = pp(aa(0))

END time
```

A.1.1 Type Correctness Conditions

```
% Existence TCC generated for time
  % proved
Time_TCC1: OBLIGATION (EXISTS (x1: rational): x1 >= 0)
```

A.2 The Since Operator in PVS

```
since   : THEORY
  BEGIN
  USING time

  i, j,k: VAR nat
  pp: VAR statepred
  action: VAR Action
  aa, bb, cc: VAR behavior

  since: [statepred -> [ state-> time]]

  since_ax: AXIOM
    since(pp)(aa(i+1)) = (IF pp(aa(i))
                           THEN (Time(aa(i+1)) - Time(aa(i)))
                           ELSE since(pp)(aa(i)) + (Time(aa(i+1)) - Time(aa(i)))
                          ENDIF)

  behavior0: TYPE =
      {aa: behavior | (FORALL pp: since(pp)(aa(0)) = Time(aa(0)) + 1)}


  TFmla: TYPE = PRED[behavior]

  tfmla(pp): TFmla =
       (LAMBDA aa: pp(aa(0)))

  tfmla(action):  TFmla =
    (LAMBDA aa: action(aa(0), aa(1)))

  Box((fmla: TFmla)): TFmla =
    (LAMBDA aa:
         (FORALL (n: nat): fmla(suffix(aa, n))))

END since
```

A.2.1 Type Correctness Conditions

```
% Subtype TCC generated for suffix(aa, n)
  % proved
Box_TCC1: OBLIGATION (FORALL aa, (n: nat): behavior?(suffix[state](aa, n)))
```

A.3 Properties of the Since Operator

```
since_thms: THEORY

BEGIN
USING time, since

  i, j,k: VAR nat
  pp, qq, rr: VAR statepred
  aa, bb, cc: VAR behavior0
  rt, rt1, rt2, rt3: VAR time
  s, s0, s1 : VAR state


  Inv_implies: LEMMA
    Inv(LAMBDA s: pp(s) IMPLIES qq(s))(aa)
    IMPLIES Inv(pp)(aa) IMPLIES Inv(qq)(aa)

  since_since: LEMMA
    Inv(LAMBDA s: since((LAMBDA s0: since(pp)(s0) <= rt1))(s) <= rt2
                  IMPLIES since(pp)(s) <= rt1 + rt2)(aa)


  since_implies: LEMMA
    Inv(LAMBDA s: pp(s) IMPLIES qq(s))(aa) IMPLIES
    Inv(LAMBDA s: since(qq)(s) <= since(pp)(s))(aa)

  min(rt1, rt2): time =
   (IF rt1 <= rt2 THEN rt1 ELSE rt2 ENDIF)

  max(rt1, rt2): time =
   (IF rt1 <= rt2 THEN rt2 ELSE rt1 ENDIF)

  max_ax:  LEMMA rt1<= max(rt1, rt2)  AND rt2<= max(rt1, rt2)

  min_ax: LEMMA min(rt1, rt2) <= rt1 AND min(rt1, rt2)<=rt2

  since_or: LEMMA
    Inv(LAMBDA s: since(LAMBDA s0: pp(s0) OR qq(s0))(s)
                 <= min(since(pp)(s), since(qq)(s)))(aa)

  since_or2: LEMMA
```

```
    Inv(LAMBDA s: since(LAMBDA s0: pp(s0) OR qq(s0))(s)
                = since(pp)(s)
             OR since(LAMBDA s0: pp(s0) OR qq(s0))(s)
                = since(qq)(s))(aa)

  since_ex_middle: LEMMA
    Inv(LAMBDA s: since(LAMBDA s0: pp(s0) AND qq(s0))(s) = since(pp)(s)
            OR  since(LAMBDA s0: pp(s0) AND NOT qq(s0))(s) = since(pp)(s))(aa)

  since_and: LEMMA
    Inv(LAMBDA s: max(since(pp)(s), since(qq)(s))
               <= since(LAMBDA s0: pp(s0) AND qq(s0))(s))(aa)


  since_and1: LEMMA Inv(LAMBDA s: since(pp)(s) <=
                        since(LAMBDA s0: pp(s0) AND qq(s0))(s))(aa)


  since_and2: LEMMA Inv(LAMBDA s: since(qq)(s) <=
                 since(LAMBDA s0: pp(s0) AND qq(s0))(s))(aa)



END since_thms
```

Appendix B

Fischer's Mutual Exclusion Protocol in PVS

```
mutex   : THEORY

  BEGIN
  USING time, since, since_thms

  process : TYPE = posnat
  i, j, k: VAR process

  aa, bb, cc : VAR behavior0

  s, s0, s1 : VAR state

  pp, qq, rr: VAR statepred

  program_counter : TYPE = {init, try, wait, cs}

  x : [state -> nat]
  PC: [state -> [process -> program_counter]]

  Check(s0, s1): bool =
   (EXISTS i: PC(s0)(i) = init AND x(s0) = 0 AND x(s1) = x(s0) AND
             PC(s1) = PC(s0) WITH [(i) := try])

  Ask(s0, s1): bool =
```

```
   (EXISTS i: PC(s0)(i) = try  AND x(s1) = i AND
              PC(s1) = PC(s0) WITH [(i) := wait])

  Doublecheck(s0, s1) :bool =
   (EXISTS i: PC(s0)(i) = wait AND x(s0) = i AND x(s1) = x(s0) AND
              PC(s1) = PC(s0) WITH [(i) := cs])

  m, n : VAR nat

  program(aa): bool =
   (Init(LAM s: x(s) = 0 AND (FORALL i: PC(s)(i) = init))(aa)
  AND
   (FORALL n:
     Check(aa(n), aa(n+1)) OR
     Ask(aa(n), aa(n+1)) OR
     Doublecheck(aa(n), aa(n+1))))

  hi, lo: time

  TI1: AXIOM
    Inv(LAM s: (FORALL i: PC(s)(i) = try IMPLIES
                     since(LAM s0: PC(s0)(i) = init)(s) <= hi))(aa)

  TI2: AXIOM
    Inv(LAM s: (FORALL i: PC(s)(i) = cs IMPLIES
                    since(LAM s0: PC(s0)(i) = wait)(s) + lo <=
                    since(LAM s0: PC(s0)(i) = try)(s)))(aa)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  Inv1: LEMMA
    program(aa) IMPLIES
    Inv(LAM s: (FORALL i: x(s) = i  IMPLIES
               since(LAM s0: PC(s0)(i) = try)(s) <=
               since(LAM s0: x(s0) = 0)(s)))(aa)

  Inv2: LEMMA
    program(aa) IMPLIES
     Inv(LAM s: (FORALL i: PC(s)(i) = try IMPLIES
                    since(LAM s0: x(s0) = 0)(s) <=
                    since(LAM s0: PC(s0)(i) = init)(s)))(aa)
```

```
Inv2_2: LEMMA
  program(aa) AND hi < lo IMPLIES
  Inv(LAM s: (FORALL i: PC(s)(i) = cs AND x(s) = i IMPLIES
               (FORALL j: PC(s)(j)/=try)))(aa)

Inv3: LEMMA
  program(aa) AND hi < lo IMPLIES
  Inv(LAM s: (FORALL i: PC(s)(i) = cs IMPLIES x(s) = i))(aa)


safety: THEOREM
  program(aa) AND hi < lo IMPLIES
   Inv(LAM s: (FORALL i, j: PC(s)(i) = cs AND PC(s)(j) = cs
               IMPLIES i = j))(aa)



END mutex
```

B.1 Type Correctness Conditions

```
% Subtype TCC generated for PC(s0) WITH [(i) := try]
  % proved
Check_TCC1: OBLIGATION
    (FORALL i, s0, s1, (x1: nat):
       PC(s0)(i) = init AND x(s0) = 0 AND x(s1) = x(s0)
         IMPLIES (FORALL (x1: nat): x1 > 0 IFF x1 > 0 OR x1 = i))

  % Subtype TCC generated for PC(s0) WITH [(i) := wait]
    % proved
  Ask_TCC1: OBLIGATION
      (FORALL i, s0, s1, (x1: nat):
         PC(s0)(i) = try AND x(s1) = i
           IMPLIES (FORALL (x1: nat): x1 > 0 IFF x1 > 0 OR x1 = i))

    % Subtype TCC generated for PC(s0) WITH [(i) := cs]
```

```
  % proved
Doublecheck_TCC1: OBLIGATION
    (FORALL i, s0, s1, (x1: nat):
       PC(s0)(i) = wait AND x(s0) = i AND x(s1) = x(s0)
         IMPLIES (FORALL (x1: nat): x1 > 0 IFF x1 > 0 OR x1 = i))
```

B.2 Proof Chain Analysis

```
safety has been PROVED.

The proof chain for safety is COMPLETE.

safety depends on the following proved theorems:
   mutex.Inv1
   nat_induction.nat_induction
   mutex.Inv2_2
   time.Time_TCC1
   mutex.Inv2
   mutex.Inv3

safety depends on the following axioms:
   mutex.TI1
   mutex.TI2
   since.since_ax

safety depends on the following definitions:
   time.behavior?
   time.nondec?
   time.Inv
   mutex.Check
   mutex.Doublecheck
   mutex.Ask
   mutex.program
   time.Init
   time.nonzeno?
   mutex.Stutter
```

Appendix C

A Railroad Crossing Controller in PVS

C.1 The Railroad Crossing

```
railroad
                : THEORY

  BEGIN

  USING time, since, since_thms


  USING time
  Train : TYPE
  Tr, Tr1, Tr2, Tr3 : VAR Train

  Tr_state : TYPE = {safe, approaching, crossing}
  train : [Train -> [state -> Tr_state]]

  s, s0, s1 : VAR state

  Approaching(s): bool = (EXISTS Tr : (train(Tr)(s) = approaching))
  Crossing(s): bool = (EXISTS Tr : (train(Tr)(s) = crossing))

  Gate : TYPE = {up, down, moving_up, moving_down}
```

```
gate : [state -> Gate]

Signal : TYPE = {lower, raise}
signal : [state -> Signal]

tr_action1 : Action =
 (LAMBDA s0, s1 :
    (EXISTS Tr : train(Tr)(s0) = safe AND train(Tr)(s1) = approaching AND
             (FORALL Tr1: Tr1/=Tr IMPLIES train(Tr1)(s1) = train(Tr1)(s0)))
  AND signal(s1) = signal(s0) AND
      gate(s1) = gate(s0))

tr_action2 : Action =
  (LAMBDA s0, s1 :
    (EXISTS Tr : train(Tr)(s0) = approaching AND
                 train(Tr)(s1) = crossing AND
             (FORALL Tr1: Tr1/=Tr IMPLIES train(Tr1)(s1) = train(Tr1)(s0)))
  AND signal(s1) = signal(s0) AND
      gate(s1) = gate(s0))


tr_action3 : Action =
  (LAMBDA s0, s1 :
    (EXISTS Tr : train(Tr)(s0) = crossing AND
                 train(Tr)(s1) = safe AND
             (FORALL Tr1: Tr1/=Tr IMPLIES train(Tr1)(s1) = train(Tr1)(s0)))
  AND signal(s1) = signal(s0) AND
      gate(s1) = gate(s0))


cont_action1 : Action =
  (LAMBDA s0, s1 :
    Approaching(s0) AND signal(s0)/=lower AND signal(s1) = lower
    AND (FORALL Tr : train(Tr)(s1) = train(Tr)(s0))
    AND gate(s1) = gate(s0))

cont_action2 : Action =
 (LAMBDA s0, s1 :
    NOT Approaching(s0) AND NOT Crossing(s0) AND
    signal(s0) = lower AND signal(s1) = raise AND
    (FORALL Tr : train(Tr)(s1) = train(Tr)(s0))
    AND gate(s1) = gate(s0))
```

```
  gate_action1 : Action =
   (LAMBDA s0, s1 : signal(s0) = lower AND gate(s0)/=down AND
         gate(s1) = moving_down AND
         signal(s1) = signal(s0) AND
         (FORALL Tr : train(Tr)(s1) = train(Tr)(s0)))

  gate_action2 : Action =
   (LAMBDA s0, s1 :  gate(s0) = moving_down AND
         gate(s1) = down AND
         signal(s1) = signal(s0) AND
         (FORALL Tr : train(Tr)(s1) = train(Tr)(s0)))


gate_action3 : Action =
   (LAMBDA s0, s1 : signal(s0) = raise AND gate(s0) /= up AND
         gate(s1) = moving_up AND
         signal(s1) = signal(s0) AND
         (FORALL Tr : train(Tr)(s1) = train(Tr)(s0)))


gate_action4 : Action =
   (LAMBDA s0, s1 : gate(s0)= moving_up AND
         gate(s1) = up AND
         signal(s1) = signal(s0) AND
         (FORALL Tr : train(Tr)(s1) = train(Tr)(s0)))

stutter: Action =
   (LAMBDA s0, s1: gate(s1) = gate(s0) AND
      signal(s1) = signal(s0) AND
      (FORALL Tr : train(Tr)(s1) = train(Tr)(s0)))

  i, j, k, m, n: VAR nat
  pp, qq, rr: VAR statepred
  aa, bb, cc : VAR behavior

program_init(s): bool =
  (FORALL Tr: train(Tr)(s) = safe)


program_transition(aa): bool =
 (FORALL i: tr_action1(aa(i), aa(i+1)) OR
             tr_action2(aa(i), aa(i+1)) OR
             tr_action3(aa(i), aa(i+1)) OR
             cont_action1(aa(i), aa(i+1)) OR
```

```
             cont_action2(aa(i), aa(i+1)) OR
             gate_action1(aa(i), aa(i+1)) OR
             gate_action2(aa(i), aa(i+1)) OR
             gate_action3(aa(i), aa(i+1)) OR
             gate_action4(aa(i), aa(i+1)) OR
             stutter(aa(i), aa(i+1)))

program(aa): bool =
  (program_init(aa(0)) AND  program_transition(aa))




non_negatives : TYPE = {x : rational | x>=0}

D, G, L : non_negatives

TI1: AXIOM Inv(LAMBDA s: Approaching(s) AND signal(s)/=lower
                IMPLIES since(LAMBDA s0: NOT Approaching(s0))(s) <=
                         D)(aa)

TI2: AXIOM Inv(LAMBDA s: signal(s) = lower IMPLIES
                gate(s) = down OR gate(s) = moving_down OR
                since((LAMBDA s0: signal(s0) /= lower))(s) <= G)(aa)

TI3: AXIOM Inv(LAMBDA s: gate(s)=moving_down IMPLIES
                since((LAMBDA s0: gate(s0) /= moving_down))(s) <= L)(aa)

TI4: AXIOM Inv(LAMBDA s:
               (FORALL Tr: train(Tr)(s) = crossing
                   IMPLIES
                      since(LAMBDA s0: train(Tr)(s0) = approaching)(s) + (D+G+L)
                    < since(LAMBDA s0: train(Tr)(s0)=safe)(s)))(aa)


Unsafe(s): bool = (Approaching(s) OR Crossing(s))

Safe(s) : bool = NOT Unsafe(s)

R, M, U : non_negatives

TI5: AXIOM Inv(LAMBDA s: signal(s) = lower AND Safe(s) IMPLIES since(Unsafe)(s) <= R)(aa)

TI6:  AXIOM Inv(LAMBDA s: signal(s) = raise
```

```
               IMPLIES (gate(s) = up OR gate(s) = moving_up
                       OR since(LAMBDA s0: signal(s0)  = lower)(s) <= M))(aa)

TI7:  AXIOM Inv(LAMBDA s: gate(s) = moving_up IMPLIES
                 since(LAMBDA s0: gate(s0)/= moving_up)(s) <= U)(aa)


 END railroad
```

C.1.1 Type Correctness Conditions

```
% Existence TCC generated for non_negatives
  % proved
D_TCC1: OBLIGATION (EXISTS (x1: rational): x1 >= 0)
```

C.2 Proof of Safety

```
railroad_pr : THEORY

  BEGIN
 USING time, railroad, since, since_thms

   Tr, Tr1, Tr2, Tr3 : VAR Train

  s, s0, s1 : VAR state

  aa, bb, cc : VAR behavior0

  pp, qq, rr: VAR statepred

  rt, rt1, rt2, rt3: VAR time

i, j, k, m, n: VAR nat


Inv1: LEMMA
  program(aa) IMPLIES
  Inv(LAMBDA s: (FORALL Tr: train(Tr)(s)=approaching IMPLIES
                      since(LAMBDA s0: train(Tr)(s0)=safe)(s)<=
                      since(LAMBDA s0: NOT Approaching(s0))(s)))(aa)

Inv2: LEMMA
  program(aa) IMPLIES
  Inv(LAMBDA s: (FORALL Tr: train(Tr)(s) = approaching
                  AND signal(s)/=lower IMPLIES
                      since(LAMBDA s0: train(Tr)(s0)=safe)(s)<= D))(aa)

Inv3: LEMMA
  program(aa) IMPLIES
  Inv(LAMBDA s: (FORALL Tr: train(Tr)(s) = approaching
                AND signal(s) = lower
                IMPLIES (gate(s) = down OR
                         gate(s) = moving_down OR
                         since(LAMBDA s0: train(Tr)(s0) = safe)(s) <=
(D+G))))(aa)

Inv4_1: LEMMA
```

```
   program(aa) IMPLIES
   Inv(LAMBDA s: signal(s)=lower  AND gate(s) /=down
          IMPLIES since(LAMBDA s0: signal(s0)/=lower)(s)
                <= since(LAMBDA s0: gate(s0)=down)(s) )(aa)

Inv4_3: LEMMA
  program(aa) IMPLIES
  Inv(LAMBDA s: (FORALL Tr: train(Tr)(s) = approaching
            IMPLIES since(LAMBDA s0: train(Tr)(s0) = safe)(s) =
                    since(LAMBDA s0: train(Tr)(s0) /= approaching)(s)))(aa)

Inv4: LEMMA
   program(aa) IMPLIES
   Inv(LAMBDA s: (FORALL Tr: train(Tr)(s) = approaching
                AND signal(s) = lower AND gate(s) = moving_down
                IMPLIES  since(LAMBDA s0: train(Tr)(s0) = safe)(s) <=
(D+G+L)))(aa)



Safety_pred(s) : bool =
   (Crossing(s) IMPLIES gate(s) = down AND signal(s) = lower)


Safety: LEMMA
  program(aa) IMPLIES
  Inv(Safety_pred)(aa)

 END railroad_pr
```

C.3 Proof of Utility

```
utility         : THEORY

  BEGIN

  USING railroad, since, since_thms
  aa, bb, cc: VAR behavior0
  s, s1, s0: VAR state

   since_up: LEMMA program(aa) IMPLIES
                 Inv(LAMBDA s: gate(s) /= up IMPLIES
                     since(LAMBDA (s0 : state): signal(s0) = lower)(s) <=
                     since(LAMBDA (s0 : state): gate(s0) = up)(s))(aa)

    X : VAR non_negatives
    lower_unsafe: LEMMA  program(aa) IMPLIES
              Inv(LAMBDA s: since(LAMBDA s0: signal(s0) = lower)(s) <= X
```

C.3.1 Proof Chain Analysis

C.3.1.1 Safety

```
Safety has been PROVED.
The proof chain for Safety is COMPLETE.

Safety depends on the following proved theorems:
   railroad_pr.Inv3
   railroad_pr.Inv1
   since_thms.since_since
   since_thms.since_ex_middle
   railroad_pr.Inv4_3
   time.Time_TCC1
   railroad_pr.Inv4
   railroad_pr.Inv2
   railroad_pr.Inv4_1
   nat_induction.nat_induction
   railroad.D_TCC1
   since_thms.since_implies
   since_thms.since_and2
   since_thms.since_and
   since_thms.max_ax

Safety depends on the following axioms:
   railroad.TI3
   since.since_ax
   railroad.TI1
   railroad.TI2
   railroad.TI4

Safety depends on the following definitions:
   railroad.gate_action2
   railroad.cont_action1
   railroad.program_init
   since_thms.max
   railroad.tr_action1
   railroad.tr_action3
   railroad.Approaching
   notequal./=
   railroad.gate_action1
   railroad.gate_action4
```

```
   railroad_pr.Safety_pred
   time.nondec?
   railroad.gate_action3
   railroad.cont_action2
   numbers.<=
   time.Inv
   railroad.tr_action2
   railroad.Crossing
   railroad.program
   railroad.program_transition
   time.behavior?
   railroad.stutter
   time.nonzeno?
```

C.3.1.2 Utility

```
utility has been PROVED.

The proof chain for utility is COMPLETE.

utility depends on the following proved theorems:
   utility.lower_unsafe
   time.Time_TCC1
   nat_induction.nat_induction
   since_thms.since_and
   since_thms.since_ex_middle
   since_thms.since_or2
   utility.since_up
   since_thms.since_and2
   since_thms.since_implies
   since_thms.since_since

utility depends on the following axioms:
   railroad.Signal_inclusive
   railroad.TI5
   railroad.TI7
   since.since_ax
   railroad.TI6
```

```
utility depends on the following definitions:
   numbers.<=
   railroad.gate_action3
   railroad.stutter
   railroad.Crossing
   railroad.cont_action2
   railroad.Safe
   time.nonzeno?
   railroad.gate_action1
   railroad.cont_action1
   time.Inv
   railroad.tr_action3
   railroad.program_transition
   railroad.tr_action1
   time.behavior?
   railroad.Approaching
   railroad.tr_action2
   since_thms.max
   time.nondec?
   railroad.gate_action2
   railroad.gate_action4
   railroad.program
   railroad.Unsafe
   railroad.program_init
```

Bibliography

[AH89] R. Alur and T. A. Henzinger. A really temporal logic. In *30th IEEE Symp. on Foundations of Computer Science*, 1989.

[AH91] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74—106. Springer Verlag, 1991.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 1—27. Springer Verlag, 1991.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[CHR92] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CM92] J. A. Carruth and J. Misra. Proof of a real-time mutual-exclusion algorithm. Notes on UNITY: 32-92, 1992.

[dBHdRR91] J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors. *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[Eme90] E. A. Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 997—1072. MIT Press and Elsevier, 1990.

[EMSS91] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 136—145. Springer Verlag, 1991.

[Gol90] D. Goldschlag. Mechanizing unity. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North-Holland, 1990.

[Gor88] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, MA, 1988.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In J. C. Mitchell, editor, *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 402–413, 1990.

[HMP91] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226—251. Springer Verlag, 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[JM86] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[Lam90] Leslie Lamport. The temporal logic of actions. Technical Report 57, DEC Systems Research Center, Palo Alto, CA, April 1990. A substantially modified version is available dated January 1991.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 447—484. Springer Verlag, 1991.

[Ost90] Jonathan S. Ostroff. A logic for real-time discrete event processes. *IEEE Control Systems Magazine*, 10(4):95–102, June 1990.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.

[SBM91] F. B. Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 618—639. Springer Verlag, 1991.

[SRRC92] J. U. Skakkebæk, A. P. Ravn, H. Rischel, and Zhou Chaochen. Specification of embedded, real-time systems. In *Proceedings of 1992 Euromicro Workshop on Real-Time Systems*. IEEE Computer Society Press, 1992.

[Stu90] Douglas A. Stuart. Implementing a verifier for real-time systems. In *Real Time Systems Symposium*, pages 62–71, Lake Buena Vista, FL, December 1990. IEEE Computer Society.