Verification of Real-Time Systems Using PVS*

N. Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025
Phone: +1 (415)859-5272
shankar@csl.sri.com

Abstract. 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 proposing a simple model for real-time computation and introducing a new operator for reasoning about absolute time. This model is formalized and mechanized within the higher-order logic of PVS. The interactive proof checker of PVS is used to develop the proofs of two illustrative examples: Fischer's real-time mutual exclusion protocol and a railroad crossing controller.

1 Introduction

Time is used in several ways in computing to ensure, for instance, that tasks are scheduled in a timely manner, deadlines are met, processes are synchronized, and race conditions are avoided. Since real-time systems are often used in critical contexts, it is important to rigorously demonstrate that certain crucial requirements are satisfied by the system. We extend existing verification frameworks for concurrent programs such as temporal logic [Pnu77], the temporal logic of actions (TLA) [Lam90], and Unity [CM88] to handle real-time behavior. We introduce a new operator whose value at any state in a computation for a given condition corresponds to the time that has elapsed since the condition last held in the computation. With this new operator, conventional reasoning techniques can be applied to demonstrate the real-time behavior of programs. Our approach to the verification real-time systems is similar to some existing proposals but, to our knowledge, none of these proposals has yet been employed in mechanized verification. Also, the model of computation we employ is simple in contrast to previous models of real-time computation. We have mechanically verified some

* 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. Connie Heitmeyer (NRL) suggested the railroad crossing example. Sam Owre (SRI) assisted with the use of PVS. The helpful comments of John Rushby (SRI), Jayadev Misra (University of Texas at Austin), Ralph Jeffords (Locus, Inc.), Jens Skakkebæk (Technical University of Denmark), and the anonymous referees are gratefully acknowledged.

simple real-time protocols using PVS [ORS92], a general-purpose specification and verification environment based on higher-order logic.

There are numerous real-time extensions to propositional temporal logics that can be applied to the verification of finite-state systems with real-time constraints. A large class of problems can be handled in this manner. Alur and Henzinger [AH91] survey the above variants of temporal logic from the point of view of expressiveness and decidability. Real-time logic (RTL) [JM86] is an extension of the traditional approach to formalizing time with an occurrence function that records the time when an event occurs for the i'th time. The Duration Calculus (DC) [CHR92] is a very expressive interval temporal logic that can be used to reason about time-varying quantities. Skakkebæk, Ravn, Rischel, and Chaochen [SRRC92] illustrate the use of the duration calculus for specifying and verifying a realistic railroad crossing controller.

We are interested here in an approach to real-time that extends more conventional approaches to the verification of state transition systems. In this regard, Lamport's Temporal Logic of Actions (TLA) [Lam90] and Chandy and Misra's Unity logic [CM88] are both elegant logics dealing with state transition systems. Unity is a simplified temporal logic (with no nesting of temporal operators) 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. 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 but could also be applied to temporal logic, TLA, or Unity. Either of the *punch* or the *duration* operators could have been used in our verification, but we 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. Proofs involving the punch or the duration operators can easily be recast in terms of the since operator, and vice-versa. The *since* operator is inspired by, and is somewhat a generalization of, the counter variables used by Abadi and Lamport [AL91], so that $|P|$ provides a counter that measures elapsed time for any given P without requiring explicit counters to be introduced. The main claim of this paper, however, 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.

2 Modeling and Proving Properties of Real-time Systems

We now state the computational model that we use to describe real-time systems. Intuitively, a *state* is taken to be a mapping of program variables to values. A *trace* is defined to be a infinite sequence of *states*. Each *program variable* maps a given state to the value of the variable in that state. *Time* is a special program variable whose value is not modified by a program. For our purpose, the value of *Time* ranges over the non-negative rational numbers. A *behavior* is a trace where the value of *Time* is non-decreasing and eventually increases above any bound (non-Zeno[2]) [AL91]. A *rooted behavior* is a behavior where the initial value of *Time* is 0. A program identifies a set of rooted behaviors. A specification also identifies a set of behaviors so that a program is also 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.

A *state predicate* is a predicate on states. A program is typically given in terms of an initialization state predicate and a set of *atomic actions*. Each atomic action is a binary relation between states. In any behavior satisfying a given program, the initial state must satisfy the initialization predicate and each pair of adjacent states must satisfy one of the atomic actions of the program. Specifications are often stated in terms of *invariance* assertions: a state predicate P is invariant over a behavior if it holds of each state in the behavior. To show that a program satisfies an invariant, it is typical to use induction over the states of an arbitrary behavior satisfying the program.

There are a few small differences here with respect to previous approaches 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,

[2] The non-Zeno constraint is not used in any of the proofs in this paper.

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 interleaves time and state changes. 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 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 computational model.

We concentrate here on the verification of invariance properties. Time-bounded versions of certain liveness properties can also be expressed as invariance properties. 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 two greater than than the value of variable y is stated as $x > y + 2$. An initialization assertion has the form **initially**$\{P\}$. An invariant assertion on the state predicate P is stated as **invariant**$\{P\}$. To prove that **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 S_i.

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. Any positive initial value of $|P|$ would be fine (as long as it is the same for every state predicate P) since this guarantees for example that the value of |**false**| at any state is always greater than the value of *Time* at that state. This makes it clear that there is never a previous state where **false** held since such a state would have a negative *Time* value.

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

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.

$$\{r = Time \land P\} \ S \ \{|P| = Time - r\}, \text{for all } P. \ \textbf{(step1)}$$

Note that the value of $|P|$ in the postcondition does not depend on whether P is true or false in the postcondition state; it only depends on the prior part of the computation. With the above axiom, the difference between the postcondition and precondition times (*i.e.*, the *delay* for the action) is equal to the value of |**true**| in the postcondition state.

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 $|P|$.

$$\{r = Time \land t = |P| \land \neg P\} \ S \ \{|P| = t + (Time - r)\}, \text{for all } P. \ \textbf{(step2)}$$

With the *since* operator and the above axioms, conventional techniques can be used to establish the correctness of programs that exhibit real-time behavior.

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

3 An 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 informally present the protocol but we are not directly using the Unity logic. Also, unlike Unity, we are not placing any fairness constraints on the transitions. 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. This simplified version of the protocol omits any exit action from the critical section or a recovery action upon failure to enter the critical section. We prove that this protocol guarantees that no two processes are simultaneously in their critical section. 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`.

```
Program mutex

   declare  x : nat, PC: [posnat -> {init, try, wait, cs}]
   initially  x = 0 || ⟨ ||  i : posnat ::  PC(i) = init ⟩
   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.[3] Note that the variables `i`, `j`, and `k` in the assertions below range over positive

[3] These constraints are presented as axioms in this proof but they are more appropriately viewed as a part of the program.

natural numbers. The first axiom associates an upper bound hi with the time that any process spends in its try state by bounding the time that has elapsed since the process was last in its previous init state.

Axiom 3.1

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

The second timing axiom associates a lower bound lo with the amount of time separating the try state and a subsequent cs state of any process. The axiom as actually stated is more complicated than necessary but has the effect of asserting that the try state must occur at least lo time units prior to any cs state of a process.

Axiom 3.2

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

Note that we require

$$\mathtt{hi} < \mathtt{lo}. \tag{1}$$

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 PC(i) = try was last observed to be true no earlier than when x was last observed to be 0. This invariant is obviously true since the Try(i) action sets the value of x to i, where i is positive.

Lemma 3.3

$$\textbf{invariant}\{\forall \mathtt{i}{:}\, \mathtt{x} = \mathtt{i} \supset |\mathtt{PC(i)} = \mathtt{try}| \leq |\mathtt{x} = \mathtt{0}|\}.$$

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

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

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

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

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

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

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

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

■

The next invariant is another obvious consequence of the **mutex** program. It asserts that if a process is in its **try** state, then **x** was equal to 0 more recently than when the process was last in its **init** state.

Lemma 3.4

$$\mathbf{invariant}\{\forall \mathtt{i}{:}\, \mathtt{PC(i)} = \mathbf{try} \supset |\mathtt{x} = 0| \leq |\mathtt{PC(i)} = \mathbf{init}|\}$$

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

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

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

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

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

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

The `Cs(j)` action, for any `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 and does not mention time in its statement. It asserts that when process **i** is in its critical section and **x** is equal to **k**, then no process **j** is in its **try** state.

Lemma 3.5

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

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

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

By Invariants 3.3 and 3.2, this yields

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

since $|\mathtt{PC(i)} = \mathtt{wait}| \geq 0$. Hence, by the Inequality (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.

Lemma 3.6

$$\textbf{invariant}\{\forall \mathtt{i}{:}\, \mathtt{PC(i)} = \mathtt{cs} \supset \mathtt{x} = \mathtt{i}\}.$$

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, $(\forall \mathtt{j}{:}\, \mathtt{PC(j)} \neq \mathtt{try})$, so that either $\mathtt{PC(i)} \neq \mathtt{cs}$ is true and the invariant trivially holds, or no Try action is enabled. ∎

Lemma 3.7

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

Proof. Follows trivially from Invariant 3.6. ∎

4 Verifying the Safety of a Railroad Crossing Controller

We next 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 a delay of 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 without the timing constraints can be written as the Unity program shown below.

```
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}
```

As with the mutex example, the timing constraints associated with the trains, controller, and gate are expressed as invariants. We omit these here but the full details appear in a technical report [Sha92].

The key safety requirement can be stated as the invariant asserting that if there is a train in the crossing, the gate is down.

Lemma 4.1

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

An additional *utility* requirement has also been proved. It asserts that if the crossing has been *safe* (*i.e.*, no train has either been approaching or crossing) for a certain number of time units, then the gate is up.

We omit the details of the proof of safety for the railroad crossing. It is significantly more complicated than the proof of the mutual exclusion protocol. The proof employs various important laws regarding invariants and the *since* operator which are listed below. The variable P and Q in the statements below range over state predicates.

$$\textbf{invariant}\{|(|P| \le x)| \le y \supset |P| \le x + y\}.$$
$$\textbf{invariant}\{P \supset Q\} \Rightarrow \textbf{invariant}\{|Q| \le |P|\}.$$
$$\textbf{invariant}\{|P \vee Q| = |P| \vee |P \vee Q| = |Q|\}.$$
$$\textbf{invariant}\{|P \wedge Q| = |P| \vee |P \wedge \neg Q| = |P|\}.$$
$$\textbf{invariant}\{|P| \le |P \wedge Q|\}.$$

5 Verification Using PVS

We have described our approach to the formalization of real-time behavior and illustrated it by sketching informal correctness proofs. These proofs have been mechanically verified using the PVS specification/verification system [ORS92]. PVS consists of a specification language based on higher-order logic and an interactive proof checker that uses powerful arithmetic decision procedures. The higher-order logic underlying PVS employs a rich type system but only a small part of the expressiveness of the type system is used for the real-time examples above. A computation trace is a sequence of states, where the type state is an undefined base type. A sequence of type T is a function from the built-in type nat of natural numbers to T. A program variable of type T is just a function from the type state to type T. The type of non-negative rational numbers can be defined in PVS as a *predicate subtype* of the built-in type rational. The program variable Time has the type shown below.

```
time : TYPE = {x : rational| x >= 0}
Time: [state -> time]
```

The notion of behavior can be captured by the following PVS declarations.

```
seq: 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?)
```

The type (behavior?) signifies the subtype of sequences of state satisfying the behavior? predicate. The type statepred is the type of predicates on state. The two axioms **(step1)** and **(step2)** (in Section 2) are captured by the following axiom defining since where aa ranges over behavior. The **(init)** axiom is not needed since it is implicit in the definition of the type of rooted behaviors.

```
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)
```

The notion of invariance is defined for an assertion pp and a behavior aa as below.

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

An atomic program action is expressed as a binary predicate which relates a precondition and a postcondition state, as illustrated by the predicate `Check` which corresponds to the `Try` action encountered in Section refMutex.

```
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])
```

The predicate `program` is defined to hold of a behavior if and only if the initial condition holds of the initial state of the behavior and every adjacent pair of states satisfies one of the atomic actions. The mutual exclusion property is stated below, where the variable `s` ranges over `state`.

```
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)
```

The proof of the mutual exclusion example follows the outline given in Section 3. The verification using PVS makes moderately heavy use of induction and the arithmetic decision procedures. Since the PVS proof makes explicit use of state, there is some overhead work in unwinding definitions of functions such as `Inv` in order to bring the expressions into a form that the decision procedures can handle. The first attempt at verifying mutual exclusion protocol took a few hours of effort, whereas the first attempt at verifying railroad crossing controller took nearly a week.

6 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 have been proved using the PVS interactive proof checker. Once a reasonable informal outline of the proof has been obtained, the mechanical verification is largely straightforward since PVS employs decision procedures for equalities and arithmetic inequalities.

As formalized above, these systems are not finite-state systems. We allow arbitrarily many processes in the mutual exclusion protocol and arbitrarily many trains in the case of the railroad crossing. Our future efforts will be directed towards making the mechanical verification of real-time systems more systematic, automatic, and compositional. We believe that with such improvements, mechanical verification based on interactive theorem proving can be competitive with model-checking in terms of human effort on similar finite-state systems.

References

[AH91] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In de Bakker et al. [dBHdRR91], pages 74—106.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In de Bakker et al. [dBHdRR91], pages 1—27.

[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*, REX Workshop, Mook, The Netherlands, June 1991. Springer Verlag.

[HMP91] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In de Bakker et al. [dBHdRR91], pages 226—251.

[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.

[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 de Bakker et al. [dBHdRR91], pages 447—484.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer Verlag.

[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 de Bakker et al. [dBHdRR91], pages 618—639.

[Sha92] N. Shankar. Mechanized verification of real-time systems using PVS. Technical Report SRI-CSL-12, SRI International Computer Science Laboratory, Menlo Park, CA, 1992.

[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.