

Mechanical Verification of Timed Automata: A Case Study*

Presented at RTAS '96, Boston, MA, June 10-13, 1996

Myla Archer and Constance Heitmeyer

Code 5546, Naval Research Laboratory, Washington, DC 20375

{archer, heitmeyer}@itd.nrl.navy.mil

Abstract

This paper reports the results of a case study on the feasibility of developing and applying mechanical methods, based on the proof system PVS, to prove propositions about real-time systems specified in the Lynch-Vaandrager timed automata model. In using automated provers to prove propositions about systems described by a specific mathematical model, both the proofs and the proof process can be simplified by exploiting the special properties of the mathematical model. This paper presents the PVS specification of three theories that underlie the timed automata model, a template for specifying timed automata models in PVS and an example of its instantiation, and both hand proofs and the corresponding PVS proofs of two propositions. It concludes with a discussion of our experience in applying PVS to specify and reason about real-time systems modeled as timed automata.

1 Introduction

Researchers have proposed many innovative formal methods for developing real-time systems [7]. Such methods are intended to give system developers and customers greater confidence that real-time systems satisfy their requirements, especially their critical requirements. However, applying formal methods to practical systems raises a number of challenges:

1. How can the artifacts produced in applying formal methods (e.g., formal descriptions, formal proofs) be made understandable to the developers?
2. To what extent can software developers use the formal methods, including formal proof methods?
3. What kinds of tools can aid developers in applying formal methods?

The purpose of this paper is to describe the results of a case study in which these issues were investigated. In particular, we are interested in how a mechanical proof system can support formal reasoning about real-time systems using a specific mathematical model. By validating human proofs of timing properties, such a system can increase confidence that a given specification satisfies critical properties of interest.

In the case study, we applied the mechanical proof system PVS [20, 21] to a solution of the Generalized Railroad Crossing (GRC) problem [8, 5, 6]. The solution is based on the Lynch-Vaandrager timed automata

model [17, 16] and uses invariant and simulation mapping techniques. Our approach, which should generalize to proving properties about real-time systems specified in any model, was to develop a template containing a set of common theories and a common structure useful in constructing timed automata models and proving properties about them. To specify a particular timed automata model and its properties, the user fills in the template. The user then may use the proof system to verify that the model satisfies the properties. This approach simplifies both the specification process and the proof process because users can reason in a specialized domain, the timed automata model; they need not master the base logic and the user interface of the full automatic proof system.

Like other approaches to formal reasoning about real-time systems, such as SMV [18, 3], HYTECH [9], and COSPAN [10], our approach is based on a formal automata model. Moreover, like these other approaches, our methods can be used to prove properties of particular automata and, like COSPAN, to prove simulations between automata. However, our approach is different from other approaches in two major ways. First, the properties we prove are expressed in a standard logic with universal and existential quantification. This is in contrast to most other approaches, where the properties to be proved are expressed either in a temporal logic, such as CTL or ICTL, or in terms of automata. Second, unlike other automata-based methods, the generation of proofs in our method is not completely automatic. Rather, our method supports the checking of human-developed proofs of the properties based on deductive reasoning. By this means, and by providing templates for developing specifications, we largely eliminate the need for ingenuity in expressing a problem using the special notations and special logics of a verification system.

Requiring some interaction with an automatic theorem prover does demand a higher level of sophistication from the user. But by supporting reasoning about automata at a high level of abstraction, we make it possible to prove more powerful results than can be done with tools requiring more concrete descriptions of automata and avoid the state explosion problem inherent in other automata-based approaches.

In our approach, each mechanically generated proof closely follows a corresponding English language proof. Such a proof is more likely to be understandable and convincing to developers familiar with the specialized timed automata domain and comfortable with English

*This work is funded by the Office of Naval Research.

language proofs. Our study identified proof techniques, such as induction, that were most useful in proofs about timed automata models. We designed PVS strategies that automatically do the standard parts of proofs having a standard structure. A major goal was to develop PVS versions of hand proofs that could be understood and, in some cases, even produced using appropriate tools, by domain experts who are able to understand hand proofs but who are not PVS experts.

In Section 2, the paper reviews the GRC benchmark, the timed automata model, and PVS. Section 3 presents three theories that underlie the timed automata model and gives their representation in PVS. One of these theories, the theory **machine**, contains as a theorem the induction principle used to prove state invariants in the timed automata model. Section 4 presents a template for defining timed automata models in PVS and an example of how the template can be instantiated to specify the *Trains* component of the timed automata solution of the GRC. Section 5 presents a hand proof and the corresponding PVS proof of the induction principle given in the theory **machine**. To illustrate how our approach can be used to check a complex proof, Section 5 presents the hand proof of the Safety Property (see the GRC problem statement below) along with the corresponding PVS proof. Sections 6, 7 and 8 present major results of our case study, a discussion of related work, and some early conclusions. A fuller version of this paper can be found in [1].

2 Background

2.1 The Generalized Railroad Crossing

The purpose of the GRC problem is to provide a benchmark for comparing different real-time formalisms. Although it is a “toy” problem, the different specifications and solutions of the GRC benchmark provide many insights into the strengths and weaknesses of different formal approaches for representing and reasoning about real-time systems. The problem statement is as follows:

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R , i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R . To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I . The i th occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the i th entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited, as have any trains that entered the crossing after τ_i).

Given two constants ξ_1 and ξ_2 , $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

Safety Property: $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (Gate is down during all occupancy intervals.)

Utility Property: $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ (Gate is up when no train is in I .)

2.2 The Timed Automata Model

The formal model used in [5, 6] to specify the GRC problem and to develop and verify a solution represents both the computer system and its environment as *timed automata*, according to the definitions of Lynch and Vaandrager [17, 16]. A timed automaton is a very general automaton, i.e., a labeled transition system. It need not be finite-state: for example, the state can contain real-valued information such as the current time or the position of a train or crossing gate. This makes timed automata suitable for modeling not only computer systems but also real-world entities such as trains and gates. The timed automata model describes a system as a set of timed automata, interacting by means of common actions. In solving the GRC problem using timed automata, separate timed automata represent the trains, the gate, and the computer system; the common actions are sensors reporting the arrival of trains and actuators controlling the raising and lowering of the gate. Below, we define the special case of timed automata, based on the definitions in [5, 6], which we used in our case study.

Timed Automata. A *timed automaton* A consists of five components:

- $states(A)$ is a (finite or infinite) set of states.
- $start(A) \subseteq states(A)$ is a nonempty (finite or infinite) set of start states.
- A mapping now from $states(A)$ to $R^{\geq 0}$, the non-negative real numbers.
- $acts(A)$ is a set of actions (or events), which include special *time-passage* actions $\nu(\Delta t)$, where Δt is a positive real number, and *non-time-passage* actions, classified as *input* and *output* actions.
- $steps(A) : states(A) \times acts(A) \rightarrow states(A)$ is a partial function that defines the possible steps (i.e., transitions).

This is a restricted definition that requires $steps(A)$ to be a function. The most general definition of timed automata permits $steps(A)$ to be an arbitrary relation. Straightforward modifications to our approach would handle the general case.

Timed Executions and Reachability. A *trajectory* is either a single state or a continuous series of states connected by time passage events. A *timed execution fragment* is a finite or infinite alternating sequence $\alpha = w_0 \pi_1 w_1 \pi_2 w_2 \dots$, where each w_j is a trajectory and each π_j is a non-time-passage action that “connects” the final state s of the preceding trajectory w_{j-1} with the initial state s' of the following trajectory w_j . A *timed execution* is a timed execution fragment in which the

initial state of the first trajectory is a start state. A state of a timed automaton is defined to be *reachable* if it is the final state of the final trajectory in some finite timed execution of the automaton.

A timed execution is *admissible* if the total amount of time-passage is infinity. We use the notation $atexecs(A)$ to represent the set of admissible timed executions of timed automaton A . The notion of admissible timed executions is important in expressing the Utility Property (and other properties defined over time intervals rather than time points) and in defining simulation relations between timed automata.

MMT Automata. An *MMT automaton* [19, 14, 13] is a special case of the general Lynch-Vaandrager timed automata model, whose states can be represented as having a “basic” part representing the state of an underlying *I/O automaton* [15], a current time component *now*, and *first* and *last* components that define lower and upper time bounds on each action.

Invariants and Simulation Mappings. An *invariant* of a timed automaton is any property that is true of all reachable states, or equivalently, any set of states that contains all the reachable states. A *simulation mapping* [17, 16, 13] relates the states of one timed automaton A to the states of another timed automaton B in such a way that the actions and their timings in admissible timed executions correspond. The existence of a simulation mapping from A to B implies that each visible behavior of automaton A is contained in the set of visible behaviors of automaton B . Proofs of both state invariants and simulation mappings have a standard structure with a base case involving start states and a case for each possible action.

2.3 PVS

The following description of PVS is taken from [23]:

PVS (Prototype Verification System) [21] is an environment for specification and verification that has been developed at SRI International’s Computer Science Laboratory. In comparison to other widely used verification systems, such as HOL [4] and the Boyer-Moore prover [2], the distinguishing characteristic of PVS is that it supports both a highly expressive specification language and a very effective interactive theorem prover in which most of the low-level proof steps are automated. The system consists of a specification language, a parser, a type checker, and an interactive proof checker. The PVS specification language is based on higher-order logic with a richly expressive type system so that a number of semantic errors in specification can be caught by the type checker. The PVS prover consists of a powerful collection of inference steps that can be used to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, au-

tomatic rewriting, and BDD-based boolean simplification.

A major goal of our study was to evaluate PVS as a basis for suitable theorem proving support for establishing properties of specifications in our specialized domain. Our experience with PVS is summarized in Section 8.

3 Underlying Theories

Our approach to specifying timed automata in PVS is to use a template that defines a set of underlying theories and provides a standard framework and standard names and definitions for each specification. The standard framework can be defined in more than one way. In Section 6, we discuss the tradeoffs in selecting a framework. Below, we introduce three underlying theories shared by all timed automata: the theory **machine**, which contains as a theorem the induction principle upon which we base our specialized induction strategies; the theory **states**, which defines the components of states; and the theory **time_thy**, which uses the extended non-negative real numbers to represent time values.¹

```

machine [ states, actions: TYPE,
         enabled: [actions,states -> bool],
         trans: [actions,states -> states],
         start: [states -> bool] ] : THEORY

BEGIN
s,s1: VAR states
a: VAR actions
n,n1: VAR nat

Inv: VAR [states -> bool];

reachable_hidden(s,n): RECURSIVE bool =
  IF n = 0 THEN start(s)
  ELSE (EXISTS a, s1: reachable_hidden(s1,n - 1)
        & enabled(a,s1) & s = trans(a,s1))
  ENDIF
MEASURE n

reachable(s): bool = (EXISTS n: reachable_hidden(s,n))
base(Inv) : bool = (FORALL s: start(s) => Inv(s))
inductstep(Inv) : bool = (FORALL s, a:
  reachable(s) & Inv(s) & enabled(a,s) => Inv(trans(a,s)))
inductthm(Inv): bool =
  base(Inv) & inductstep(Inv)
  => (FORALL s: reachable(s) => Inv(s))

machine_induct: THEOREM (FORALL Inv: inductthm(Inv))

END machine

```

Figure 1. The Theory machine.

3.1 The Theory machine

Figure 1 shows the PVS specification of the theory **machine**. This theory, which defines the meaning of mathematical induction in the context of the timed

¹An additional theory, *atexecs*, which we do not need for the examples in Section 5, defines $atexecs(A)$, the admissible timed traces of automaton A .

automata model, is the core of our general PVS strategy for performing the standard steps of state invariant proofs. It is also of interest because Section 5 uses the proof of the induction principle as an example of how a hand proof can be translated into PVS. The theory consists of the induction principle along with the definitions needed for its statement. Most of these definitions are straightforward.

The theory has the five parameters needed to define a timed automaton: *states*, the automaton’s states; *actions*, its input alphabet; *start*, its start states; *enabled*, the guards on state transitions; and *trans*, the automaton’s transition function. The two parameters *states* and *actions* are simply type parameters. The actual parameters in an instantiation of the template are the *states* and *actions* types (i.e., the sets of possible values of *states* and *actions*) of some particular timed automaton. The parameter *start* is instantiated by a predicate on states true only for start states, and the parameter *enabled* by a predicate on actions and states true only when the action is enabled in the state. The parameter *trans* is instantiated by a function that maps an action and a state to a new state.

The body of the theory describes six predicates used to define the induction principle. The first predicate *Inv* represents an arbitrary predicate (i.e., an invariant) on states. The second predicate *reachable_hidden* is true of a state *s* and natural number *n* if *s* is reachable from a start state in *n* steps. The MEASURE clause of this definition permits PVS to verify during type checking that the predicate *reachable_hidden* is always well defined, i.e., that its (recursive) definition terminates on all arguments. The predicate *reachable* is true of a state *s* if *reachable_hidden* is true for *s* and some natural number *n*. (We have proved in PVS that this definition of reachability is equivalent to the definition given in Section 2.2.) The next two predicates define the two parts of the induction principle: *base*, which states that the given invariant holds for the base case, and *inductstep*, which states that the invariant is preserved by every enabled action on a reachable state. Finally, the predicate *inductthm* on invariants states that an invariant is true if it holds in the base case and is preserved in the induction step.

3.2 The Theory states

Figure 2 gives the PVS specification of the very simple theory **states**. The main purpose of this theory is to define a standard record structure and standard temporal information for the states of an automaton. The theory has four parameters. The first three, *actions*, *MMTstates*, and *time*, are type parameters. The fourth parameter *fin_pred* is a predicate that is true if its argument, a time value, is finite.

The body of the theory contains a single statement defining the record structure of a state. The theory requires that a state contain a basic component, a time component, and components *first* and *last* representing time restrictions on specified actions. In PVS, the symbols “[# ··· #]” are record brackets. The *basic* component contains all of the nontimed information in the state along with any special absolute time markers. The *now* component is an element of type *time* satisfying the predicate *fin_pred* (that is, *now* is finite).

```

states [ actions, MMTstates, time : TYPE,
         fin_pred : [time -> bool] ] : THEORY
BEGIN
  states: TYPE = [# basic: MMTstates, now: (fin_pred),
                 first, last: [actions -> time] #]
END states

```

Figure 2. The Theory states.

The *first* and *last* components specify the upper and lower time bounds on each action.²

Both the theory *machine* and the theory *states* have parameters that are functions. The ability to define a theory with function parameters and to define states with components that are functions exists because PVS has a higher-order logic. In general, using a higher-order logic facilitates the creation of template specifications. Section 8 describes other advantages of a higher-order logic.

3.3 The Theory time_thy

Figure 3 gives the PVS specification of the data type *time* and the theory **time_thy**. In a timed automaton, each state has an associated time in $R^{\geq 0}$. However, in the time bounds associated with actions, infinity is allowed as a time value to represent the case when no final deadline on an action exists. Thus, to represent time in our template, we require the union type, $R^{\geq 0} \cup \{\infty\}$.

Like many other strongly typed languages, the PVS specification language represents union types using abstract data type definitions reminiscent of traditional algebraic specifications. In PVS, these definitions consist of a line for each constructor which specifies the constructor name, names and types for each argument (if any) to the constructor, and a predicate that recognizes elements of the data type built using the constructor.³ We thus define the type *time* as a PVS data type. (Later, we define another part of our template, the type *actions*, as a PVS data type; its definition is similarly understood.)

The data type *time* has two constructors. The first constructor, *fintime*, has a non-negative real parameter *dur* and the recognizer *fintime?*, and the second constructor, *infinity*, has no parameters and the recognizer *infintime?*. The PVS prover recognizes the following assertions as true:

$$\begin{aligned}
 dur(fintime(x)) &= x \quad (\text{for any } x \in R^{\geq 0}) \\
 fintime?(fintime(x)) & \quad (\text{for any } x \in R^{\geq 0}) \\
 infintime?(infinity) &
 \end{aligned}$$

The theory **time_thy** contains the definitions of the standard arithmetic operators and predicates for time

²Although the type *states* is designed to make it easy to express an MMT automaton as a timed automaton, it is general enough for any timed automaton.

³When processing a **datatype** declaration, the PVS type-checker generates individual declarations for all the constructors, their arguments, and their recognizers, together with axioms defining their relationships, an induction axiom, etc.

```

time: DATATYPE
BEGIN
  ftime(dur:{r:real|r>=0}): ftime?
  infinity: inftime?
END time

time_thy: THEORY
BEGIN
  IMPORTING time
  zero: time = ftime(0);
  <= (t1,t2:time):bool =
    IF ftime?(t1) & ftime?(t2) THEN dur(t1) <= dur(t2)
    ELSE inftime?(t2) ENDIF;
  >= (t1,t2:time):bool =
    IF ftime?(t1) & ftime?(t2) THEN dur(t1) >= dur(t2)
    ELSE inftime?(t1) ENDIF;
  < (t1,t2:time):bool =
    IF ftime?(t1) & ftime?(t2) THEN dur(t1) < dur(t2)
    ELSE NOT(inftime?(t1)) & inftime?(t2) ENDIF;
  > (t1,t2:time):bool =
    IF ftime?(t1) & ftime?(t2) THEN dur(t1) > dur(t2)
    ELSE NOT(inftime?(t2)) & inftime?(t1) ENDIF;
  + (t1,t2:time):time =
    IF ftime?(t1) & ftime?(t2)
    THEN ftime(dur(t1) + dur(t2)) ELSE infinity ENDIF;
  - (t1:time, t2:(ftime?)):time =
    IF ftime?(t1) & dur(t1) >= dur(t2)
    THEN ftime(dur(t1) - dur(t2)) ELSE infinity ENDIF;
END time_thy

```

Figure 3. Theory `time_thy` and Data Type `time`.

values. Note that we have exploited the support PVS provides for overloading names.

4 A Timed Automata Model Template

4.1 What the Template Looks Like

Figure 4 shows one template we have developed for defining a timed automata model in PVS. The template imports appropriate instantiations of the fixed theories `time_thy`, `states`, and `machine`. The theory `time_thy` appears first in the template because it has no parameters. The two remaining theories, `states` and `machine`, appear later in the template because their parameters must first be defined. The template is instantiated by filling in the missing parts and adding any desired auxiliary declarations and definitions. The missing parts are represented in Figure 4 by the symbol “< . . . >”.

Before the theory `states` can be imported, two of its parameters, `actions` and `MMTstates`, must be defined. The type `actions` is defined as a data type with one constructor, the time passage action `nu`, which is an action associated with every timed automata model. The corresponding parameter extractor, called `timeof`, is declared as an element of type `time` that satisfies the predicate `ftime?`. The symbol “< . . . >” is a placeholder for the other (non-time-passage) actions associated with a given timed automaton. The type of the *basic* component of an element of type `states` is `MMTstates`. The symbol “< . . . >” that follows “MMT-

```

<timed-automaton name>: THEORY
BEGIN
  IMPORTING time_thy
  actions : DATATYPE
  BEGIN
    nu(timeof:(ftime?)): nu?
    <...>
  END actions;
  MMTstates: TYPE = <...>
  IMPORTING states[actions,MMTstates,time,ftime?]
  OKstate? (s:states): bool = <...> ;
  enabled_general (a:actions, s:states):bool =
    now(s) >= first(s)(a) & now(s) <= last(s)(a);
  enabled_specific (a:actions, s:states):bool =
    CASES a OF
      nu(delta_t): (delta_t > zero & <...>),
      <...>
    ENDCASES;
  trans (a:actions, s:states):states =
    CASES a OF
      nu(delta_t): s WITH [now := now(s)+delta_t],
      <...>
    ENDCASES;
  enabled (a:actions, s:states):bool =
    enabled_general(a,s) & enabled_specific(a,s)
    & OKstate?(trans(a,s));
  start (s:states):bool = (now(s) = zero) & <...> ;
  IMPORTING machine[states, actions, enabled, trans, start]
END <timed-automaton name>

```

Figure 4. A Timed Automata Model Template.

`states: TYPE =`” is a place holder for the nondefault part of the state of the timed automaton, typically a record structure. Once `actions` and `MMTstates` are defined, the type `states` can be defined by importing the appropriate instance of the theory `states`.

One proceeds in a similar fashion before importing the theory `machine`. The definition of the predicate `enabled` divides naturally into three parts. The first part, `enabled_general`, is the same for all timed automata; it defines the time bounds associated with actions. In particular, if the automaton is in state `s`, the time `now(s)` that action `a` can occur is bounded below by `first(s)(a)` and above by `last(s)(a)`. The second part, called `enabled_specific`, restricts the time passage action `nu` to positive values and provides place holders for other restrictions on when actions are enabled in a given timed automaton. The third part is defined by the predicate `OKstate?` on states, which provides an optional mechanism for enforcing a state invariant by fiat. In the transition function `trans`, the definition of “`nu(delta_t)`” is the same for all timed automata: as expressed by the `WITH` construct, the effect of a time passage action is simply to update the `now` component of the state. The remaining action cases for a particular timed automaton must be supplied. Finally, the predicate `start(s)` must enforce the requirement `now(s) = zero`.

We introduce an additional convention in our timed

automaton template to make our proof strategies simpler: State invariants are assigned names of the form $Inv_<name>$, and the associated state invariant lemma (or theorem) is called $lemma_<name>$ (or $theorem_<name>$). The PVS proof of the Safety Property in Section 5 uses this convention.

4.2 Instantiating the Template

To illustrate an instantiation of the template, we use the template to specify in PVS the timed automaton *Trains*, a component of the timed automata solution of the GRC problem. Before presenting the PVS specification, we present the original specification of *Trains*, extracted from [6]. The timed automaton *Trains* has no input actions, three output actions, $enterR(r)$, $enterI(r)$, and $exit(r)$, for each train r , and the time passage action $\nu(\Delta t)$. The basic component of each train's state is the *status* component, which simply describes where the train is. Each train's state also includes a current time component *now*, and *first* and *last* components for each action, giving the earliest and latest times at which an action can occur once enabled.

The state transitions of *Trains* are described by specifying the “Precondition” under which each action can occur and the “Effect” of each action. s denotes the state before the event occurs, and s' the state afterwards. The transition function contains conditions that enforce the bound assumptions; that is, an event cannot happen before its *first* time, and time cannot pass beyond any *last* time. In the *Trains* specification, only the state components *now* and $first(enterI(r))$ and $last(enterI(r))$ for each r contain nontrivial information, so the other cases are ignored. Note that the time that $enterI(r)$ occurs is always no sooner than ϵ_1 and no later than ϵ_2 after the train r entered the region R . The states and transition of the timed automaton *Trains* are shown in Figure 5.

State:	
<i>now</i> , a nonnegative real, initially 0	
for each train r :	
$r.status \in \{not_here, P, I\}$, initially <i>not_here</i>	
$first(enterI(r))$, a nonnegative real, initially 0	
$last(enterI(r))$, a nonnegative real or ∞ , initially ∞	
Transitions:	
<i>enterR(r)</i>	<i>enterI(r)</i>
Precondition:	Precondition:
$s.r.status = P$	$s.r.status = P$
Effect:	$s.now \geq s.first(enterI(r))$
$s'.r.status = P$	Effect:
$s'.first(enterI(r)) = now + \epsilon_1$	$s'.r.status = I$
$s'.last(enterI(r)) = now + \epsilon_2$	$s'.first(enterI(r)) = 0$
	$s'.last(enterI(r)) = \infty$
<i>exit(r)</i>	$\nu(\Delta t)$
Precondition:	Precondition:
$s.r.status = I$	for all r ,
Effect:	$s.now + \Delta t \leq s.last(enterI(r))$
$s'.r.status = not_here$	Effect:
	$s'.now = s.now + \Delta t$

Figure 5. States and Transitions of *Trains*.

Figure 6 uses our template to specify the *Trains* automaton in PVS. In addition to the time passage ac-

```

trains: THEORY
BEGIN
  IMPORTING time_thy
  delta_t: VAR (fintime?);      eps_1, eps_2: (fintime?);
  train: TYPE;                  r: VAR train;
  actions : DATATYPE
  BEGIN
    nu(timeof:(fintime?):) nu?
    enterR(Rtrainof:train): enterR?
    enterI(Itrainof:train): enterI?
    exit(Etrainof:train): exit?
  END actions;
  a: VAR actions;              status: TYPE = {not_here,P,I};
  MMTstates: TYPE = [train -> status];
  IMPORTING states[actions,MMTstates,time,fintime?]
  status(r:train, s:states):status = basic(s)(r);
  OKstate? (s:states): bool = true ;
  enabled_general (a:actions, s:states):bool =
    now(s) >= first(s)(a) & now(s) <= last(s)(a);
  enabled_specific (a:actions, s:states):bool =
  CASES a OF
    nu(delta_t): (delta_t > zero &
      (FORALL r: now(s) + delta_t <= last(s)(enterI(r))),
    enterR(r): status(r,s) = not_here,
    enterI(r): status(r,s) = P & first(s)(a) <= now(s),
    exit(r): status(r,s) = I,
  ENDCASES
  trans (a:actions, s:states):states =
  CASES a OF
    nu(delta_t): s WITH [now := now(s)+delta_t],
    enterR(r): (# basic := basic(s) WITH [r := P], now := now(s),
      first := first(s) WITH [(enterI(r)) := now(s)+eps_1],
      last := last(s) WITH [(enterI(r)) := now(s)+eps_2] #),
    enterI(r): (# basic := basic(s) WITH [r := I], now := now(s),
      first := first(s) WITH [(enterI(r)) := zero],
      last := last(s) WITH [(enterI(r)) := infinity] #),
    exit(r): s WITH [basic := basic(s) WITH [r := not_here]]
  ENDCASES;
  enabled (a:actions, s:states):bool =
  enabled_general(a,s) & enabled_specific(a,s) & OKstate?(trans(a,s));
  start (s:states):bool =
  (s = (# basic := (LAMBDA r: not_here), now := zero,
    first := (LAMBDA a: zero), last := (LAMBDA a: infinity) #));
  IMPORTING machine[states, actions, enabled, trans, start]
END trains

```

Figure 6. Instantiating the Template for *Trains*.

tion ν , the instantiation contains the three output actions, $enterR(r)$, $enterI(r)$, and $exit(r)$, for each train r . The *basic* component of each train's state, which has type *status*, has the value *not_here*, P , or I . The predicate *enabled_specific* captures the “Preconditions” and the function *trans* captures the “Effects” shown in the above specification. Note the lower and upper bounds, ϵ_1 and ϵ_2 on the action $enterI$. Also note the initialization of the start states with the basic component set to *not_here*, the *now* component to zero (thus fulfilling the template requirement), and the *first* and *last* components to zero and infinity, respectively. Our template instantiation also includes some auxiliary declarations, such as the types *trains* and *status* needed to define the type *MMTstates*, and the function $status(r, s)$, which retrieves the value of the *status* component of train r in state s .

5 Two Examples of Proofs

To illustrate the correspondence between a hand proof and a PVS proof, this section presents exam-

Step 1. We wish to prove the following formula—call it (*):

$$\begin{aligned} & \forall Inv : states \rightarrow bool. \\ & (\forall s : states. start(s) \Rightarrow Inv(s) \wedge \\ & \quad \forall s : states, a : actions : ((reachable(s) \wedge Inv(s) \\ & \quad \quad \wedge enabled(a, s)) \\ & \quad \quad \Rightarrow Inv(trans(a, s)))) \\ & \Rightarrow \forall s : states. reachable(s) \Rightarrow Inv(s) \end{aligned}$$

Step 2. Let Inv_1 be any state invariant. Claim: the body of (*) holds for Inv_1 .

Step 3. To prove the claim, suppose that

$$(\alpha) \forall s : states. start(s) \Rightarrow Inv_1(s)$$

and

$$(\beta) \forall s : states, a : actions : ((reachable(s) \wedge Inv_1(s) \wedge enabled(a, s)) \Rightarrow Inv_1(trans(a, s))).$$

Step 4. Then, let s_1 be any state. We will show that

$$reachable(s_1) \Rightarrow Inv_1(s_1).$$

Step 5. Thus, suppose $reachable(s_1)$.

Step 6. Now, $reachable(s_1)$ means that s_1 can be reached from a start state in n steps, for some $n \geq 0$.

Step 7. We will use induction on n .

Step 7.1. If $n = 0$, then $start(s_1)$; hence by (α) , $Inv_1(s_1)$ holds.

Step 7.2. If $n > 0$, then $s_1 = trans(a_0, s_0)$ for some state s_0 reachable in $n-1$ steps from a start state and some action a_0 for which $enabled(a_0, s_0)$ is true. By inductive hypothesis, $Inv_1(s_0)$ holds. By (β) applied to a_0 and s_0 , $Inv_1(trans(a_0, s_0))$ holds; i.e., $Inv_1(s_1)$ holds.

QED.

Figure 7. Induction Principle Hand Proof.

ple hand proofs and corresponding PVS proofs of two results. The first hand proof is a proof of the induction principle presented in Section 3.1. The second is a proof of the Safety Property taken from the technical report [5].

5.1 Proof of the Induction Principle

The first hand proof establishes an essential component of the support we provide for developing PVS proofs for timed automata, namely, the induction principle. This example illustrates how a very detailed hand proof can be translated almost directly into a PVS proof. At the same time, it illustrates the need to bring additional knowledge to the prover at points where the hand proof implicitly appeals to human knowledge and experience.

Figure 7 gives our detailed hand proof of the induction principle, while Figure 8 presents our best PVS approximation to that proof. A systematic method for translating much of the hand proof to the PVS proof maps short proof steps to particular PVS rules or strategies. For example, to appeal to a definition, use EXPAND; to say “let ...” or “choose ...”, use SKOLEM; to apply a quantified formula or to establish one by providing an instance, use INST; to do straightforward simplification and propositional reasoning, use GROUND; and to set up an induction, use INDUCT. Together with a few uses of DELETE to simplify the current proof goal and one use of SIMPLIFY to simplify an assertion, the set of translations above is sufficient to handle nearly everything in our hand proof.

Step 1.	(EXPAND "inductthm")
Step 2.	(SKOLEM 1 "Inv_1")
Step 3.	(FLATTEN)
Step 4.	(SKOLEM 1 "s_1")
Step 5.	(FLATTEN)
Step 6.	(EXPAND "reachable")
Show induction result is sufficient.	(CASE "(FORALL(n): (FORALL(s): (reachable_hidden(s,n) => Inv_1(s))))")
Let n_0 be such that reachable_hidden(s_1,n_0).	(("1" (DELETE -2 -3) (SKOLEM -2 "n_0"))
Induction result applied to s_1 and n_0 finishes the proof.	(INST -1 "n_0") (INST -1 "s_1") (GROUND))
Step 7.	("2" (INDUCT "n"))
Begin Step 7.1.	(("1" (DELETE -2 -3 2)
If n = 0 then start(s_1).	(EXPAND "reachable_hidden")
By (α) , $Inv_1(s_1)$ holds.	(EXPAND "base") (PROPAX))
Begin Step 7.2.	("2"
Let j_1 ≥ 0.	(DELETE -1 -3 2)
Suppose ind. hyp. for j_1.	(SKOLEM 1 "j_1") (FLATTEN)
Choose s_01, and suppose reachable in j_1+1 steps.	(SKOLEM 1 "s_01") (FLATTEN)
Then s_01 is reached from s via a for some s reachable in j_1 steps.	(EXPAND "reachable_hidden" -2) (SIMPLIFY)
Let a = a_0 and s = s_0.	(SKOLEM -2 ("a_0" "s_0"))
By inductive hypothesis, $Inv_1(s_0)$ holds.	(INST -1 "s_0") (GROUND)
By (β) applied to a_0 and s_0, $Inv_1(trans(a_0, s_0))$ holds,	(EXPAND "inductstep") (INST -5 "s_0" "a_0") (GROUND)
because s_0 is reachable in j_1 steps.	(EXPAND "reachable") (INST 1 "j_1"))))

Figure 8. PVS Proof of the Induction Principle.

The correspondence between the steps in the hand proof and the PVS steps is more easily understood from the actual user interaction with PVS. Figure 9 shows the contents of the PVS proof buffer during the first few steps of the proof in Figure 8. The current goal at each step is represented as a *sequent*, with a line dividing a list of hypotheses from a list of conclusions. At each step, the object is to show that at least one of the conclusions follows from the hypotheses. The sequents in Figure 9 all have only one conclusion.

The parts of the hand proof in Figure 7 that require the help of a knowledgeable human when translating to the PVS proof in Figure 8 are those associated with induction: first, the specification of exactly what to prove by induction; second, establishing that this inductive assertion is enough to obtain the proof; and finally, replacement of the state s_1 in the induction step Step 7.2 by an arbitrary state reachable in the same number of steps.

To fully understand the correspondence between the proofs in Figures 7 and 8, one needs to run PVS. For example, although the specification of **machine** makes clear that “inductstep” corresponds to hypothesis (β) , to apply (β) to a_0 and s_0 , one needs to know that its

```

machine_induct :
|------
{1} (FORALL Inv: inductthm(Inv))
Rule? (EXPAND "inductthm")
Expanding the definition of inductthm, this simplifies to:
machine_induct :
|------
{1} (FORALL Inv: base(Inv) & inductstep(Inv)
=> (FORALL (s: states): reachable(s) => Inv(s)))
Rule? (SKOLEM 1 "Inv_1")
For the top quantifier in 1, we introduce Skolem constants: Inv_1,
this simplifies to:
machine_induct :
|------
{1} base(Inv_1) & inductstep(Inv_1)
=> (FORALL (s: states): reachable(s) => Inv_1(s))
Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent, this simplifies to:
machine_induct :
{-1} base(Inv_1)
{-2} inductstep(Inv_1)
|------
{1} (FORALL (s: states): reachable(s) => Inv_1(s))
Rule? (SKOLEM 1 "s_1")
For the top quantifier in 1, we introduce Skolem constants: s_1,
this simplifies to:
machine_induct :
[-1] base(Inv_1)
[-2] inductstep(Inv_1)
|------
{1} reachable(s_1) => Inv_1(s_1)
Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent, this simplifies to:
machine_induct :
[-1] base(Inv_1)
[-2] inductstep(Inv_1)
{-3} reachable(s_1)
|------
{1} Inv_1(s_1)
Rule? (EXPAND "reachable")
Expanding the definition of reachable, this simplifies to:
machine_induct :
[-1] base(Inv_1)
[-2] inductstep(Inv_1)
{-3} (EXISTS (n: nat): reachable_hidden(s_1, n))
|------
[1] Inv_1(s_1)

```

Figure 9. Contents of the PVS Proof Buffer.

assertion number is -5 . The ability to tag assertions or identify them by content would reduce this problem.

In contrast to our detailed PVS proof, we show in Figure 10 a more conventional PVS proof of the induction principle which relies heavily on the workhorse strategy GRIND.⁴ In this proof, one must also supply the inductive assertion. In addition, one must determine when to tell GRIND not to reduce quantified

⁴The GRIND strategy in PVS approximates an automatic theorem prover. It expands definitions and forms, applies rewrites, invokes propositional and arithmetic decision procedures, and does automatic skolemization and instantiation. Instantiation is done by best guess and can be incorrect. To provide more control of instantiation and other features, GRIND has options that can be selected by supplying arguments.

```

("""
(GRIND :IF-MATCH NIL)
(CASE "(FORALL (n): (FORALL (s):
(reachable_hidden(s,n) IMPLIES Inv!1(s))))")
(("1" (GRIND))
("2"
(INDUCT "n")
(("1" (GRIND))
("2"
(GRIND :IF-MATCH NIL)
(APPLY (THEN (INST -8 "s!1" "a!1") (INST -2 "s!1"))
"At this point, it is evident that we have the inductive hypo-
thesis for n = j!1 and the hypothesis (beta) to work with, and
need to establish Inv!1(perform(a!1,s!1)). So, we instan-
tiate the latter with s!1 and a!1, and the former with s!1!1.")
(GRIND))))))

```

Figure 10. PVS Proof Using GRIND.

formulae (the effect of the “:IF-MATCH NIL” argument), and help PVS decide how to use the inductive hypothesis and assumption (β). One must also analyze the current goal after a call to GRIND terminates to recognize what help is needed.

We refer to these two styles of PVS proofs as small step and large step proofs. One can view a hand proof as a proof plan for a PVS proof. With a small step proof, one can more easily determine what point has been reached in a proof plan and what step one wishes to take next. With a large step proof, especially one using generic large steps based on GRIND, it is harder to control the position in the proof plan. In fact, in some cases, this position may not be well defined, since GRIND may perform steps from the plan out of order. With experience, a PVS user can often predict the result of a large step, but even so must rely on interaction with PVS to see just what piece of information from the plan should be provided to PVS next.

In our experience, both styles of proof benefit, in terms of speed of construction with minimal backtracking, from the existence of a proof plan. We note that if the automatic-instantiation feature of GRIND had been somewhat more powerful, the only proof information PVS would have required in the large step proof is the inductive assertion, and the reason why the resulting PVS proof worked would be impossible to discern.⁵ The degree to which we find the resulting PVS proof convincing, in the sense that the theorem is true for the right reasons, is certainly greater with the small step proof, although some of these reasons were supplied to PVS in the large step proof.

On the other hand, for theorems with complex proofs, or for theorems with proofs having a standard structure, mimicking all the micro-steps of the PVS proof is unnecessarily tedious and repetitive. In our specialized domain, we have been able to define reusable PVS strategies that allow the user to follow a proof plan reasonably closely without most of the tedium of providing the micro-steps. Large step proofs using GRIND typically execute several times as slowly

⁵If one uses GRIND\$ in place of GRIND, PVS will save the small steps that GRIND has followed. However, understanding these steps is very difficult.

as short step proofs. Because our strategies are specialized for timed automata, they yield an efficiency comparable to that of short step proofs.

5.2 Proof of the Safety Property

Our second example of a hand proof translated into a PVS proof is a proof with a standard structure: namely, the proof by induction of a state invariant. The particular state invariant is the Safety Property for the timed automaton *SystImpl*, which is stated and proved as Lemma 6.3 in [5, 6]. Figure 11 shows the hand proof and the corresponding PVS proof.

The PVS proof uses the induction strategy `AUTO_PROOF_UNIV_SYSTIMPL` to set up the induction, potentially producing subgoals for the base case and each possible action. Subgoals deemed sufficiently “trivial” are proved automatically, and only the nontrivial subgoals are displayed. As can be seen, in the hand proof, the action cases for *enterI(r)* and *raise* are the nontrivial cases. The PVS proof of the *enterI(r)* case is obtained as in the hand proof by invoking the precondition, doing a case split, applying the indicated lemmas appropriately, and asking for a little simplification and linear arithmetic. The PVS proof of the *raise* case is translated analogously. An extra case *up* is generated in the PVS proof, but is handled by invoking the precondition, a step considered obvious in the hand proof.

In our general experience with proofs of state invariants, we have noticed that an “extra” case waved away as obvious in the hand proof occasionally turns up in the PVS proof. Appealing to one of a short list of standard facts about timed automata typically proves these cases. In proving the Safety Property, the standard fact needed is that the precondition must be satisfied or the case will not arise. Adding such standard facts to the induction strategies would eliminate the need to deal with most such “obvious” cases interactively, but at the expense of longer proof times.

A close look at the PVS proof of the Safety Property in Figure 11 reveals a few subtleties involving the choice of a particular version of a strategy or its arguments.⁶ However, all of these choices could be made automatically by an interface to PVS, given user input of the form “use induction” or “apply invariant lemma 6_1”. Thus, in proving the Safety Property, it is possible to shield the verifier from low-level interaction with PVS. Our experience so far with other state invariant proofs indicates that this is very often the case.

A third category of translations of hand proofs to PVS proofs contains translations of proofs with a more ad hoc proof structure than state invariant proofs. For more ad hoc proofs, our results so far suggest that one cannot disengage the verifier from low-level interaction with PVS to the extent that one can with the more structured state invariant proofs. However, one can identify repeated patterns of reasoning that occur because a result from a particular domain with much shared structure is being proved. Appropriate PVS strategies can frequently handle these repeated patterns. A menu of such strategies, specially tailored for

⁶The PVS proof commands are embedded in `APPLY` so that they can be accompanied by comments.

Lemma 6.3. *In all reachable states of SystImpl, if Trains.r.status = I for any r, then Gate.status = down.*

Proof: Use induction. The interesting cases are *enterI* and *raise*. Fix *r*.

1. *enterI(r)*

By the precondition, $s.Trains.r.status = P$.

If $s.Gate.status \in \{up, going-up\}$, then Lemma 6.1 implies that $s.Trains.first(enterI(r)) > now + \gamma_{down}$, so $s.Trains.first(enterI(r)) > now$. But, the precondition for *enterI(r)* is $s.Trains.first(enterI(r)) \leq now$. This means that it is impossible for this action to occur, a contradiction.

If $s.Gate.status = going-down$, then Lemma 6.2 implies that $s.Trains.first(enterI(r)) > s.Gate.last(down)$. By Lemma B.1, $s.Gate.status = going-down$ implies $s.Gate.last(down) \geq now$. This implies that $s.Trains.first(enterI(r)) > now$, which again means that it is impossible for this action to occur.

The only remaining case is $s.Gate.status = down$. This implies $s'.Gate.status = down$, which suffices.

2. *raise*

We need to show that the gate doesn't get raised when a train is in *I*. So suppose that $s.Trains.r.status = I$. The precondition of *raise* states that $\exists r: s.ComplImpl.r.sched-time \leq now + \gamma_{up} + \delta + \gamma_{down}$, which implies that, for all *r*, $s.ComplImpl.r.sched-time > now$. But Parts 1 and 3 of Lemma 5.1 imply that in this case, $s.Trains.r.status = P$, a contradiction.

```

Inv_6_3_A(s: states):bool =
  (FORALL (r: train): status(r,s) = I => gate_status(s) = fully_down);
(*** (APPLY (AUTO_PROOF_UNIV_SYSTIMPL "Inv_6_3_A")
  "Use induction. Fix r = r_2.")
  ("1" (APPLY (THEN (EXPAND "enabled_specific"))(SYSTIMPL_SIMP))
    "Case enterI(r_1). Invoke the precondition.")
    (CASE "gate_status(s_1) = fully_up OR gate_status(s_1) = going_up"
      ("1" (APPLY (THEN (APPLY_UNIV_INV_LEMMA "6_1" "r_1")
        (SYSTIMPL_SIMP))
        "Invoke the invariant lemma 6.1.")
      (APPLY (TIME_ETC_SIMP)
        "Derive contradiction with the precondition."))
      ("2" (APPLY (THEN (APPLY_UNIV_INV_LEMMA "6_2" "r_1")
        (SYSTIMPL_SIMP))
        "Invoke the invariant lemma 6.2.")
      (APPLY (THEN (APPLY_INV_LEMMA "B_1_1")
        (SYSTIMPL_SIMP))
        "Invoke invariant lemma B_1, part 1.")
      (APPLY (TIME_ETC_SIMP)
        "Derive contradiction with the precondition."))))
    ("2" (APPLY (THEN (EXPAND "enabled_specific"))(SYSTIMPL_SIMP)
      (INST 2 "r_2"))
      "Case raise. Invoke and specialize the precondition.")
      (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_1" "r_2")
        (SYSTIMPL_SIMP))
        "Invoke invariant lemma 5_1, part 1.")
      (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_3" "r_2")
        (SYSTIMPL_SIMP))
        "Invoke invariant lemma 5_1, part 3.")
      (APPLY (TIME_ETC_SIMP) "Derive contradiction."))
      ("3" (APPLY (THEN (EXPAND "enabled_specific"))(SYSTIMPL_SIMP))
        "Case up. Invoke the precondition."))))

```

Figure 11. Safety Property Hand/PVS Proofs.

timed automata models, can support translating hand proofs into PVS proofs made up of a combination of a limited set of small PVS steps combined with large standard steps, whose correspondence with the source hand proofs is much easier to see.

6 Summary of Results

Based on our experience to date, we discuss below our use of template specifications, how repeating patterns in proofs were detected and exploited, how best to interact with the theorem prover, and how real-time properties are expressed and proven in our approach.

6.1 Using Template Specifications

Using a template to create a formal specification of a particular mathematical model greatly reduces the required effort. This reduction comes from two sources. First, with the basic theories and lemmas already specified, the amount that remains to be specified for a particular model is significantly reduced. Second, the existence of conventions regarding names, types, and definitions of the missing parts eliminates many organizational decisions required in specifying a particular model: the specifier needs only to fill in the missing pieces.

Creating a template helped us to identify commonalities among instances of the timed automata model that are useful in proving properties. In our study, we identified induction structured over actions as an important principle that underlies many proofs about timed automata models. This principle can be used to prove state invariants about these models by invoking appropriately designed PVS strategies.

Templates can be enforced in different ways. Through an additional, top-level parameterized theory added to the framework in Section 4, we can require that all models include a time passage action and enforce other similar template conventions. The advantage of this approach is that it permits many additional generic lemmas to be proved without instantiating the template. An alternative is to enforce the template conventions through an interface that compiles user-provided information into a PVS specification of the proper form. Our experiments with template instantiations suggests that proofs run more efficiently when the second approach is used.

However, no matter how the template is enforced, the strong type system in PVS is very helpful in establishing a template discipline. In contrast to Lamport [11], we find strong typing more a help than a hindrance.

6.2 Patterns in Timed Automaton Proofs

In analyzing proofs in the timed automata domain, our approach has been to create small step proofs, optimize them for both efficiency and logical structure, and find patterns that can be translated into PVS strategies. We have found a variety of patterns. These patterns can be classified by whether it is possible to translate them into an appropriate strategy, whether the strategy can be written in PVS as it stands or requires enhancements to PVS, and whether the strategy requires instance-specific details to compile or choose. The classification of certain repeating patterns remains to be decided. For some patterns, we do not yet have a PVS strategy but can supply a heuristic: an example is the recurring argument in hand proofs that time cannot pass beyond a certain bound unless a certain type of event occurs. In following a hand proof, the need to turn to a heuristic typically arises when the hand proof does not supply enough detail.

6.3 Patterns in Using PVS

As indicated above, our approach to PVS proofs about timed automata is to follow a hand proof as closely as possible. For nontrivial theorems, a hand proof provides essential guidance in constructing the

automated proof, since it presents, in some organized fashion, the reasons why a theorem is believed to be true. These reasons generally correspond closely to the information that must be supplied to a theorem prover.

As illustrated in Section 5, very detailed proofs and routine proofs can be easily translated into PVS. A direct translation of a detailed hand proof to a PVS proof involves detailed human guidance, but most of this guidance is routine and could conceivably be mechanized. In translating a hand proof with a routine structure (e.g., an induction proof of a state invariant of an automaton with a standard structure), human guidance is mostly needed to provide the non-routine facts and case splits needed to complete proof branches generated by a strategy that performs the standard initial stages of the proof. Translating hand proofs that omit many details and have an ad hoc structure to PVS requires significant interactive guidance. However, this problem can be reduced by using model-specific strategies and heuristics. The model-specific strategies permit one to take larger steps in a proof and make it easier to track one's place in the hand proof. A preliminary exercise in developing model-specific strategies for timed automata and employing them in an ad hoc proof resulted in a more than 60% reduction in proof size (415 lines to 158 lines) with no penalty—in fact, a slight improvement—in the running time of the proof.

To keep track of the correspondence between a hand proof and a PVS proof, inserting comments in the PVS proof is very helpful, and for a proof of any length, it is essential. A combination of comments in the proof and a glossary of English meanings of PVS strategies can create confidence that the PVS proof succeeded for the right reasons.

6.4 Proving Real-Time Properties

In our approach, the real-time properties of a timed automaton are determined by the definitions of *enabled* and *trans*. Real-time properties that are state invariants are proved in PVS by induction. The specific stage at which reasoning about time occurs in each branch of the induction is typically a point at which a set of inequalities involving time values has been established by invoking the definitions of *enabled* and *trans* and by introducing previous state invariant lemmas. One must then prove that at least one inequality in a new set of inequalities holds. If time were simply represented by the non-negative real numbers, the decision procedures in PVS that do arithmetic would complete the proof in a single step. Because we include infinity in the set of possible time values, these decision procedures will not work directly. To handle this problem, we developed a strategy called `TIME_ETC_SIMP` that reduces *time* inequalities to inequalities involving non-negative real numbers and then invokes the PVS decision procedures for arithmetic.

Care must be taken in translating assertions involving time values from hand proofs into PVS. While “negative” time values can be used in hand proofs, they cannot be used in our PVS proofs, because our type *time* does not contain values corresponding to negative numbers. To handle this problem, we transform any equations or inequalities involving subtraction of time values so that they involve only addition, prior to doing

PVS proofs.

Other real-time properties of a timed automaton concern the relative timing of events during an admissible timed execution. Proofs of these properties often involve establishing the claim that if the automaton is in a certain state, then time cannot pass beyond a certain time bound unless a specified event occurs prior to the bound. As indicated above, we lack a specific strategy for this type of reasoning. However, we do have a heuristic that often works. With this heuristic, we prove by induction that if the required event does not occur between the current time *now(s)* and the time bound, then some component of the state involved in the precondition for time passage is not changed by subsequent events and that the precondition prevents a time passage event from crossing the bound. It is likely that a PVS strategy with a sufficient set of arguments can be developed to set up a proof based on this heuristic. We also envision an interactive interface that guides the user through an application of the strategy or of the heuristic directly.

7 Related Work

An effort closely related to ours uses the Larch Shared Language and the Larch Prover (LP) to prove state invariants and simulations for real time systems represented as timed automata [12]. In this approach, proofs are developed in LP that follow hand proofs, but proof strategies specialized to timed automata that can support this correspondence in more complex proofs and proofs of an ad hoc structure are not included. Whether such proof strategies can be developed in LP to the same extent as in PVS is an open question. Other efforts have used PVS in proving properties of real-time systems expressed in different formalisms. For example, a proof assistant that encodes the Duration Calculus in PVS and supports the development of Duration Calculus specifications and proofs of real-time properties is described in [22]. A second effort whose goal is to make formal specification and theorem proving in PVS more accessible to hardware design practitioners is described in [23].

8 Conclusions

A major goal of our research is to make the use of an automatic theorem prover feasible for software developers. Checking properties of specifications of real-time systems with a mechanical theorem prover can lead to the early discovery of inconsistencies and omissions in a design. We envision that such use of automatic provers can be made feasible by appropriate automated support. Parts of this support may be direct, e.g., through an appropriate interface to a system such as PVS that supports specification and automatic theorem proving. Other parts of it may be indirect, e.g., by way of a mechanism for arriving at formal specifications that are understandable to both the developer and a formal methods expert, and for creating mechanically checked proofs that also are understandable to both.

Our early results are encouraging. For real-time systems specified in the timed automata model, we have developed a template that can be instantiated in a straightforward manner. For understandable translations of hand proofs, we have identified PVS proof steps

that correspond to natural steps in hand proofs. We have been able to define specialized strategies in PVS that make the translation of hand proofs of state invariants into recognizably similar PVS proofs straightforward and also simple enough in many cases that developers themselves could create them through an appropriate interface to PVS. Such an interface would perform such services as choosing the appropriate instance of the induction strategy or the invariant lemma strategy and would also be useful to the formal methods expert in simplifying the proof effort. We have defined additional model-specific strategies that can be useful to the formal methods expert in translating more complex proofs of properties of designs into recognizable PVS equivalents.

Although PVS strategies such as GRIND reduce the necessary human interaction with the theorem prover in obtaining a proof, the reasoning in proofs obtained from these strategies is hard to follow. In contrast, we have found that human-understandable PVS proofs can be derived naturally and with an acceptable level of human interaction by applying a set of domain-specific strategies in the course of following a hand proof. Being specialized, these strategies result in proofs with a significantly shorter execution time than proofs based on GRIND. There is also an advantage in undertaking proofs using our methods and strategies when the proof does not succeed: it is much simpler to discover the reason that the proof does not succeed when one knows exactly the corresponding step in the hand proof.

Similar observations apply when we compare our methods to other automata-based formal approaches to reasoning about real-time systems. In particular, while the latter can be used to prove properties, they provide no feedback on why the properties are true. When a proof fails, a tool such as SMV can supply the trace of a counterexample. While this information is helpful, it is on the same low level as that used in software debugging. By contrast, the information provided by the failure of a mechanically checked hand proof is on a conceptual level, thus providing more direct information on where one's assumptions about a particular automaton specification are incorrect. Mechanically checked hand proofs have an additional advantage: they make it easier to predict the effects of changes in specifications on the properties of the specified automata. In addition, when these changes do not affect the validity of a property, checking the property can often be done by modifying the former proof only slightly, or not at all—as opposed to rerunning a time-consuming algorithm on the entire specification.

Our use of PVS as a basis for specification and proof support has been largely successful. Using decision procedures to handle the obvious low-level reasoning greatly facilitates the creation of the proofs. Moreover, the rich specification language of PVS supports both parameterized theories and higher-order constructs that allow functions and predicates to be used as record components and theory parameters. As a result, once one has identified common features to include in a template, expressing the template in PVS is largely straightforward and natural. The higher-order logic of PVS makes it possible to prove useful,

reusable high level theorems about arbitrary predicates and functions, such as our induction principle.

However, the current version of PVS does not always satisfy our needs. For example, it imposes some constraints that limit the directness with which one can express timed automaton specifications and translate steps from hand proofs. For example, we must use a single slot (which we label *basic*) in the state record type to hold all the specialized state components of an instantiation of our template, and we must sometimes use assertion numbers in the translation of hand proofs. Most or all of these problems can be solved by a combination of enhancements to PVS (e.g., the ability to recognize assertions by content) and an interface to PVS specialized for the timed automata model.

Acknowledgments

We wish to thank the anonymous reviewers for insightful comments and our colleagues Ramesh Bharadwaj and Ralph Jeffords for very helpful discussions.

References

- [1] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. Technical Report NRL/MR/5546-96-7845, NRL, Wash., DC, 1996.
- [2] R. Boyer and J Moore. *A Computational Logic*. Academic Press, 1979.
- [3] S. Campos, E. Clarke, and M. Minea. Analysis of real-time systems using symbolic techniques. In *Formal Methods for Real-Time Computing*, chapter 9. John Wiley & Sons, 1996.
- [4] M. J. C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
- [5] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-51, Lab. for Comp. Sci., MIT, Cambridge, MA, 1994. Also Technical Report 7619, NRL, Wash., DC 1994.
- [6] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, December 1994.
- [7] C. Heitmeyer and D. Mandrioli, editors. *Formal Methods for Real-Time Computing*. Number 5 in Trends in Software. John Wiley & Sons, 1996.
- [8] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc., 10th Intern. Workshop on Real-Time Operating Systems and Software*, May, 1993.
- [9] T. Henzinger and P. Ho. Hytech: The Cornell Hybrid Technology Tool. Technical report, Cornell University, 1995.
- [10] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: the Automata-Theoretic Approach*. Princeton University Press, 1994.
- [11] L. Lamport. Types are not harmless. Digital Systems Research Center, July 1995.
- [12] V. Luchangco, E. Söylemez, S. Garland, and N. Lynch. Verifying timing properties of concurrent algorithms. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII: Proc. of the 7th IFIP WG6.1 Intern. Conference on Formal Description Techniques (FORTE'94, Berne, Switzerland, October 1994)*, pages 259–273. Chapman and Hall, 1995.
- [13] N. Lynch. Simulation techniques for proving properties of real-time systems. In *REX Workshop '93*, volume 803 of *Lecture Notes in Computer Science*, pages 375–424, Mook, the Netherlands, 1994. Springer-Verlag.
- [14] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distrib. Comput.*, 6:121–139, 1992.
- [15] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.
- [16] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. To appear in *Information and Computation*.
- [17] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer-Verlag, 1991.
- [18] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [19] M. Merritt, F. Modugno, and M. R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, editors, *CONCUR'91: 2nd Intern. Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1991.
- [20] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
- [21] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.
- [22] J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems, Lecture Notes in Computer Science 863*. Springer-Verlag, 1994.
- [23] M. K. Srivas and S. P. Miller. Formal verification of a commercial microprocessor. Technical Report SRI-CSL-95-04, Computer Science Lab, SRI Intl., Menlo Park, CA, 1995.