Naval Research Laboratory

Washington, DC 20375-5320

Mechanical Verification of Timed Automata: A Case Study

MYLA ARCHER
CONSTANCE L. HEITMEYER

Center for High Assurance Computer Systems
Information Technology Division

December 31, 1998

DTIC QUALITY INSPECTED 2

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 31, 1998	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Mechanical Verification of Timed Automata: A Case Study	PE - 61153N15

6. AUTHOR(S)

Myla Archer and Constance Heitmeyer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Naval Research Laboratory Washington, DC 20375-5320	NRL/MR/5540--98-8180

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Office of Naval Research, Code 311, Ralph Wachter 800 North Quincy Street Arlington, VA 22217-5660	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited.	A

13. ABSTRACT (Maximum 200 words)

This report describes 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. Because both specifications and methods of reasoning about them tend to be repetitive, the use of a standard template for specifications, accompanied by repetitive, the use of a standard template for specifications, accompanied by standard shared theories and standard proof strategies or tactics, if often feasible. Presented are 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. Both hand proofs and the corresponding PVS proofs of two propositions are provided to illustrate how these can be made parallel at different degrees of granularity. Our experience in applying PVS to specify and reason about real-time systems modeled as timed automata is also discussed. The methods for reasoning about timed automata in PVS developed in the study have evolved into a system called TAME (Timed Automata Modeling Environment). A summary of recent developments regarding TAME is provided. A shorter version of the report was presented at the 1997 Real-Time Applications Symposium.

14. SUBJECT TERMS			15. NUMBER OF PAGES
Real-time systems	Automated deduction	Theorem proving	78
Verification	Formal methods	Mechanical verification	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

MECHANICAL VERIFICATION OF TIMED AUTOMATA: A CASE STUDY

1 Introduction

Researchers have proposed many innovative formal methods for developing real-time systems [14]. 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 report 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 [31, 33] to a solution of the Generalized Railroad Crossing (GRC) problem [16, 12, 13]. The solution is based on the Lynch-Vaandrager timed automata model [28, 27] 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, a common structure, and specialized proof strategies, 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. The techniques we have developed using this approach have become the basis for a tool TAME (Timed Automata Modeling Environment), which we are continuing to develop.

Like other approaches to formal reasoning about real-time systems, such as SMV [29, 8], HyTech [18], and COSPAN [19], 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 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 report 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. Section 9 gives an overview of recent developments regarding TAME.

Additional detail on the work done in the study is provided in the appendices. Appendix A presents the theory **atexecs** of admissible timed executions of timed automata, the fourth underlying theory for the timed automata model. Appendix B contains the full specifications of four example timed automata from [13, 12]. Appendix C presents the PVS strategies that were used in proofs of properties of the example timed automata. The full set of proofs of state invariants of the example timed automata is shown in Appendix D. An example ad hoc proof of a property of admissible timed executions of timed automata, together with a discussion of feasibility of better PVS strategies to support it and similar proofs, is presented in Appendix E. Appendix F exhibits an alternative timed automaton template with an example of its use.

A briefer version of this report can be found in [1].

2   Background

2.1   The Generalized Railroad Crossing Problem

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 ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith 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 [12, 13] 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 [28, 27]. 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 [12, 13], 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, which are *visible*, and *internal* actions.

- *steps(A)* : *states(A)* $\times$ *acts(A)* $\to$ *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 $\alpha = w_0 \pi_1 w_1 \pi_2 w_2 \cdots$, 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* [30, 25, 24] 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* [26], 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* [28, 27, 24] relates the states of one timed automaton A to the states of another timed automaton B with the same visible actions in such a way that the visible actions and their timings in any admissible timed execution of A correspond to those in some admissible timed execution of B. The existence of a simulation mapping from A to B thus 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 [35]:

> PVS (Prototype Verification System) [33] 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 [11] and the Boyer-Moore prover [7], 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, automatic rewriting, and BDD-based boolean simplification.

```
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**.

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 for Timed Automata

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]

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 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

---

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

```
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**.

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. Together, *enabled* and *trans* define the steps of the timed automaton.

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.[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 nonstandard absolute time markers. The *now* component is an element of type *time* satisfying the predicate *fin_pred* (that is, *now* is finite). The *first* and *last* components specify the upper and lower time bounds on each action.[3]

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, $R^{\geq 0} \cup \{\infty\}$.

---

[2] See Lemma *reachability* in the theory **opspec_atexecs_aux** in Appendix B.3.

[3] Although 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.

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

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

Figure 3: The Theory **time_thy** and the Data Type *time*.

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.[4] 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 *inftime?*. The PVS prover recognizes the following assertions as true:

$$dur(fintime(x)) = x \quad \text{(for any } x \in R^{\geq 0})$$
$$fintime?(fintime(x)) \quad \text{(for any } x \in R^{\geq 0})$$
$$inftime?(infinity)$$

The theory **time_thy** contains the definitions of the standard arithmetic operators and predicates for time values. Note that we have exploited the support PVS provides for overloading names.

4   A Template for Timed Automata Models

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

---

[4]When processing a **datatype** declaration, the PVS typechecker generates individual declarations for all the constructors, their arguments, and their recognizers, together with axioms defining their relationships, an induction axiom, etc.

```
<timed-automaton name>: THEORY
  BEGIN
   IMPORTING time_thy
   actions : DATATYPE
    BEGIN
       nu(timeof:(fintime?)): nu?
       <...>
    END actions;
   MMTstates: TYPE = <...>
   IMPORTING states[actions,MMTstates,time,fintime?]
   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 Template for Specifying Timed Automata Models.

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 "$< \ldots >$".[5]

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 *fintime?*. The symbol "$< \ldots >$" 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 "$< \ldots >$" that follows "MMTstates: 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, its time $now(s)$ allows action a to occur if it 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

---

[5]The form of the template to use, as well as that of the missing parts, depends on how adherence to the template conventions is enforced. Section 6 discusses this issue. An alternate template is presented in Appendix F.

of the state.[6] The remaining action cases for a particular timed automaton must be supplied. Finally, the partial declaration of the predicate *start(s)* indicates that it 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 [13]. 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 P. The states and transitions of the timed automaton *Trains* are shown in Figure 5.

State:
    now, a nonnegative real, initially 0
    for each train r:
        r.status ∈ *{not-here, P, I}*, initially *not-here*
        first(enterI(r)), a nonnegative real, initially 0
        last(enterI(r)), a nonnegative real or ∞, initially ∞

Transitions:

enterR(r)
    Precondition:
        s.r.status = P
    Effect:
        s'.r.status = P
        $s'.first(enterI(r)) = now + \varepsilon_1$
        $s'.last(enterI(r)) = now + \varepsilon_2$

enterI(r)
    Precondition:
        s.r.status = P
        s.now ≥ s.first(enterI(r))
    Effect:
        s'.r.status = I
        s'.first(enterI(r)) = 0
        s'.last(enterI(r)) = ∞

exit(r)
    Precondition:
        s.r.status = I
    Effect:
        s'.r.status = not-here

$\nu(\Delta t)$
    Precondition:
        for all r,
        s.now + Δt ≤ s.last(enterI(r))
    Effect:
        s'.now = s.now + Δt

Figure 5: States and Transitions of the Timed Automaton *Trains*.

---

[6]The template we present here thus restricts the time passage action more than does the model we described in Section 2.2. Sometimes, as in the railroad crossing solution with a continuous gate action described in [12], one wants to allow other components of the state besides the *now* component to change during time passage. An obvious modification of our template would permit our template to support the description of these more general timed automata.

```
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 to Specify *Trains*.

Figure 6 uses our template to specify the *Trains* automaton in PVS. In addition to the time passage action *nu*, 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*.[7] 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, *eps_1* and *eps_2*, on the action *enterI(r)* established by the action *enterR(r)*. 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

---

[7]In this simple example, we were able to define the type *MMTstates* of the automaton basic state as a simple function type, rather than the more typical record type used in *AxSpec*, *OpSpec*, and *SystImpl* (see Appendix B).

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 example 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 [12].

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 "suppose" the hypotheses of an implication being proved, use FLATTEN; to say "let $\cdots$" or "choose $\cdots$", use SKOLEM; to apply a quantified formula or to establish one by providing an instance,

---

Step 1. What one wants to prove is the following formula—call it (*):

$\forall Inv : states \rightarrow bool.$
$(\forall s : states. start(s) \Rightarrow Inv(s) \wedge$
$\quad \forall s : states, a : actions : (reachable(s) \wedge Inv(s) \wedge enabled(a, s)$
$\quad\quad\quad\quad\quad\quad \Rightarrow Inv(trans(a, s))))$
$\Rightarrow \forall s : states. reachable(s) \Rightarrow Inv(s)$

Step 2. Let Inv_1 be a particular state invariant. We will show that the body of (*) holds for Inv_1.

Step 3. So, suppose that

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

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

Step 4. Then, let s_1 be a particular 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 $start(s_1)$. In this case, 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: Hand Proof of the Induction Principle.

	(""
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))))")
	(("1"
Let n_0 be such that reachable_hidden(s_1,n_0).	(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))
	("2"
Step 7.	(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 $\geq$ 0. Suppose ind. hyp. for j_1.	(DELETE -1 -3 2) (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. Let a = a_0 and s = s_0.	(EXPAND "reachable_hidden" -2) (SIMPLIFY) (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.

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.[8]

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

[8]The PROPAX in Figure 8 is generated by PVS, and is not supplied by the user.

11

```
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.

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 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

12

```
(""
(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 "s1!1" "a!1") (INST -2 "s1!1"))
        "At this point, it is evident that we have the inductive hypothesis for n = j!1 and the
         hypothesis (beta) to work with, and need to establish Inv!1(perform(a!1,s1!1)).
         So, we instantiate the latter with s1!1 and a!1, and the former with s1!1.")
    (GRIND))))))
```

Figure 10: PVS Proof Using GRIND.

induction principle which relies heavily on the workhorse strategy GRIND.[9] In this proof, one must also supply the inductive assertion. In addition, one must determine when to tell GRIND not to reduce quantified 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 back-tracking, 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.[10] 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 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 [12, 13]. 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,

---

[9]The 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.

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

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* + γ_{down}, so *s.Trains.first(enterI(r))* > *now*. But, the precondition for *enterI(r)* is *s.Trains.first(enterI(r))* $\le$ *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)* $\ge$ *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* $\le$ *now* + γ_{up} + δ + γ_{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: Hand Proof and PVS Proof of the Safety Property.

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. The precondition is present among the hypotheses in name, but its definition must be expanded for it to be taken into account. One other standard fact has sometimes been required in handling the "obvious" cases of the

hand proofs we have translated from [12]: the uniqueness of actions. In particular, if $r\_1 \neq r\_2$, then $enterI(r\_1) \neq enterI(r\_2)$, and similarly for other actions with arguments. One can envision that certain relationships among constants used to define a timed automaton might also sometimes be considered too obvious to mention. Adding knowledge of 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. It would also obscure their application in the PVS proof in cases in which one wants to explicitly mention them in the hand proof for emphasis.

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.[11]

With respect to the choice of arguments, one notes first that when an invariant lemma is invoked, it is sometimes applied to r_1, and sometimes to r_2. It would seem that the choice of appropriate argument would have to be made by examining the current PVS goal. Although this could be done, the choice can be made on another basis: choose r_1 when one is clearly referring to the train index of the action case; choose r_2 when one is clearly referring to the train whose status is I (or, more generically, to the train mentioned explicitly in the body of the lemma). Second, we note that the precondition of *raise* is "specialized" when it is invoked in the *raise* branch of the proof. This is necessary because it contains a quantifier. The instantiation to give that quantifier is r_2, since the *raise* action is not indexed over trains. This instantiation can also be chosen based on matching the body of the quantified formula to the known fact that r_2 is the train whose status is I. Thus, these choices of argument can be made, if not automatically, at least from information provided by the user of a more abstract nature than the argument's explicit name.

With respect to versions of strategies, one question that arises with respect to the PVS proof in Figure 11 is how one chooses the particular induction, simplification, and lemma application strategies used. The choice of AUTO_PROOF_UNIV_SYSTIMPL and SYSTIMPL_SIMP can be made by the PVS user on knowing that one is proving a universally quantified invariant in the theory SystImpl. Whether to use APPLY_UNIV_INV_LEMMA or APPLY_INV_LEMMA is determined by whether or not the invariant lemma is universally quantified. We note that while the user can make these choices of strategy, all of them 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 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

---

[11]The PVS proof commands are embedded in APPLY so that they can be accompanied by comments.

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 can be exploited in designing specialized support for proving properties. The very discipline of creating a template helps one to identify high level common patterns. For example, 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. At a more detailed level, the particular specification structure and conventions enforced in a template can be taken advantage of in creating reusable proof strategies: for example, in our template, we know that we always want to expand the definition "start" of the start state in doing the base case of an induction proof. This advantage can be taken even farther by exploiting even lower level features of the template such as definition structure. More detail on the relation between our template and the strategies we have developed can be found in Appendix C.

Templates can be enforced in different ways, with tradeoffs involved. 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, define the types of *enabled-specific* and *trans*, define the significance of these three operators in relation to the admissible timed executions of a timed automaton, and enforce other similar template conventions. This approach has the advantage of permitting many generic lemmas that provide important support for proof strategies specialized for timed automata to be proved without instantiating the template. These generic lemmas can then be kept in a standard library, saving much of the processing time needed to load instantiations of the template into PVS. 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 suggest that proofs of properties 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 [20], we find strong typing more a help than a hindrance.

6.2   Repeated 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 can arise when the hand proof does not supply enough detail.

Many patterns that recur in small step proofs are of such a general nature that existing PVS strategies already handle them. A simple example is the pattern handled by the PVS strategy SKOSIMP, which corresponds to "Suppose that the hypotheses hold for some generic values; we will prove that the conclusion holds for these values." However, there are some patterns of a general nature that require domain specialized PVS strategies. An example is the repeated need to substitute names for values (as opposed to expressions—that is, semantically as opposed to syntactically), and, conversely, to retrieve information associated with names. Although this need in general will arise in nearly any domain, appropriate strategy support will require such domain-specific information as when two expressions represent the same value, what kind of information there exists to retrieve, and so on. We have made some progress in writing PVS strategies that partially meet these needs; see Appendix C. Ideal strategy support will require certain enhancements to PVS, such as the ability to recognize formulae by content.

Others patterns have appeared that are specific to timed automata. Examples are the repeated need to apply state invariant lemmas and to establish that a state is reachable. In Appendix C, we present strategies we have defined to help with each of these patterns. For applying state invariants, we have two strategies, one

16

for unquantified invariants and another for universally quantified invariants. The choice of which strategy to invoke is assumed to be made externally to PVS. In principle, the choice could be made internally by PVS if more access to the PVS data structures were provided. To show that a state in an admissible timed execution is reachable, we have a strategy that applies a lemma containing general reachability results. The user must supply instantiations that focus the general results on the neighborhood (in terms of the number of non-time steps that have occurred) of the state that is to be confirmed to be reachable. With appropriate enhancements to PVS, the appropriate instantiations could be inferred automatically from the current proof goal; also, after simplification of the results from application of the lemma, the irrelevant results could be deleted. This is a further example where a better PVS strategy is in principle feasible, but not yet expressible: helpful enhancements to PVS would again include the ability to recognize formulae by content, and also the ability to extract parts of formulae and to apply naming conventions to formulae.

There are other patterns specific to timed automata that require instantiation-specific knowledge before an appropriate PVS strategy can be constructed. An important example is in the setting up cases in an induction proof. Each case of a particular form is handled in a particular way. The information that must be supplied includes whether the case is the base case or an action case; if an action case, how the action is parameterized; and if an action case, whether the body of the corresponding case in the definition of trans is an IF_THEN_ELSE. All of the supplied information can be determined automatically from the instantiated template specification. Thus, it is possible, in principle, to compile such strategies from a template instantiation. It is conceivable that the necessary choices could be made by a single PVS strategy that accesses details of definitions; such a strategy would require access to the PVS data structures that currently has not been provided. There is clearly an efficiency tradeoff involved in the choice of solution; a strategy which itself must make choices will obviously be less efficient. Whether the reduction in efficiency is significant remains to be determined.

In proving state invariants, another interesting pattern arises. In particular, when the state invariant involves quantification, one often wishes to coordinate the simplification of the quantified formulae from the inductive hypothesis and conclusion (specifically, one wants to instantiate one with the skolem constant or constants from the other). This is typically the case when one is reasoning about the behavior of independent entities, such as the trains in the GRC benchmark. When this is the case, the standard part of the induction strategy can be extended to include this; this was in fact done in our proof of the Safety Property. The fact that a PVS strategy could be developed to do the coordination of skolemization and instantiation depended heavily on the predictability of the assertion numbers of the related quantified formulae. For invariants involving quantification in other forms, such as existential or embedded, it is harder to predict the related assertion numbers. In principle, it should be possible to extend the standard induction strategy to handle any particular additional coordination case analogously, and also to select the appropriate variant of the induction strategy to apply automatically from knowledge of the form of the invariant to be proved. Implementing such extensions would require enhancements to PVS.

Some repeating patterns that occur in proofs are of such a general nature that the best one can offer as a general solution is a heuristic. For these cases, one can hope to provide automated support that guides the user in applying the heuristic.

6.3   Repeated 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. Undertaking such a direct translation helps to clarify at which points in the proof crucial information that involves some human insight must be supplied to the prover: e.g., in our example proof of the induction principle, this crucial information involved the exact formulation and use of what needs to be proved using induction over natural numbers. 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. The need for human guidance can be further minimized in this case by taking advantage of template conventions to provide domain-specific strategies for recurring types of reasoning. For example, in proving the Safety Property in section 5, the strategy TIME_ETC_SIMP handles reasoning about extended time values.

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 domain-specific strategies and heuristics. The domain-specific strategies permit one to take larger steps in a proof and make it easier to track one's place in the hand proof. An example that has arisen in ad hoc proofs about timed automata is the need to confirm with minimum effort that a certain state is reachable, because one wants to apply a state invariant lemma to it. We have designed a strategy that simplifies this step greatly, and would be able to improve it further if certain enhancements are made to PVS. The specialized strategies that we have developed so far for ad hoc proofs have not reduced proof efficiency. A preliminary exercise in developing domain-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.

We have undertaken the translation into PVS of several hand proofs of properties of timed automata of an ad hoc structure, the longest being the one page hand proof of a result equivalent to the Utility Property in Section 2 for the timed automaton OpSpec. From this experience, we can make additional observations on the process of translating hand proofs into PVS.

The first observation is that although PVS has many built-in rules and strategies that allow one to closely mimic the steps of a detailed hand proof, there are some cases in which one cannot quite do this with PVS as it stands. This is not only due to the fact that one must sometimes take many steps in PVS to follow a step in the hand proof—a phenomenon that will become less of a problem as more specialized strategies are developed—but results from PVS sometimes forcing a slightly different structure on the proof by way of undesired case splits.

In general, case splits should be avoided unless they are natural occurrences in a human style proof, since when they are forced, the existing proof plan will need to be revised. A major example where one is forced in PVS to split a goal into separate subgoals, where conceptually this is not necessary, is as follows. When one has facts A and $A \Rightarrow B$ in the antecedent of the goal, a call to the PVS strategy ASSERT will, in most cases, reduce the second fact to simply B, *provided* the form of A is simple. For the case when A is more complex, a user-defined PVS strategy can be written that will, in most cases, accomplish the same thing. An exception in both cases is when the form of B is $B_1 \Rightarrow B_2$. In this case, one is forced into a case split. The difficulty is that some of the PVS rules and strategies are not exactly on target with the natural steps in a hand proof. Adding rules to PVS that provide finer control of subgoal manipulation should overcome this difficulty.

Even when case splits are part of the proof plan, they can cause the problem of losing track of one's place in a proof when using PVS. Planned case splits may be explicit in the hand proof, or implicit as the result of including an in-line lemma in the proof—that is, a lemma proved on the spot and then applied. After doing several case splits in a row and then discharging subgoals in the default order, upon returning to the subgoal or subgoals that correspond to the second or additional branches from the first case split, one can easily forget where they came from, and therefore, what one's approach to their proof will be. The ability to attach comments to related subgoals at least semi-automatically, based on user input, would greatly alleviate this problem.

The second observation is that a very common occurrence in the process of creating a machine-checked proof is the reappearance of subgoals that have already been proved in an earlier proof branch. In the hand proof, one can simply say "as shown earlier ..." but this will not work in PVS or most other automatic theorem proving systems. However, one of the advantages to starting from a hand proof is the ability to see

easily where some piece of information is used more than once in the proof. A careful restructuring of the hand proof prior to undertaking the PVS proof can eliminate much subgoal duplication in the PVS proof, particularly of subgoals corresponding to facts playing a major role in the proof. Our experience has shown that eliminating all duplication of subgoals is difficult and perhaps impossible, since some subgoals come from type correctness conditions implicitly needed for the application of lemmas in the course of the proof. And even though some repeated subgoals can be eliminated by restructuring the proof plan, there will be some proofs where this can make the reasoning in the proof more difficult to follow, since such restructuring usually involves the up front introduction of facts that will be used more than once whose role in the proof is not yet clear. Introducing these facts as separately proved lemmas is one possible solution, but not always ideal; assuming that one is checking hand proofs using PVS, one must ask why these facts were not introduced as separate lemmas in the hand proof. The answer is typically that they are too specialized to be worth including in a theory, being unlikely to be used outside the current proof. Thus, some mechanism in PVS for handling repeating subgoals would be a very welcome enhancement.

6.4   Expressing and 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. The proof is then completed using only reasoning about the inequalities. 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, as a result, 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 [22]. In this approach, proofs are developed in LP that follow hand proofs, but proof strategies specialized to timed automata that can support a close correspondence in the more complex induction or simulation 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 [34]. A second effort whose goal is to make formal specification and theorem proving in PVS more accessible to hardware design practitioners is described in [35].

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. At least one case has arisen where it would be helpful to have parametric polymorphism in the type system, as is the case in HOL.

The limitation on specifications is visible in the template instantiation of the timed automaton *Trains*; the *status* component of a state of *Trains* is represented by the *basic* component of the state of the PVS instantiation **trains**, rather than by a component named *status*. An auxiliary function definition is included in the PVS version to permit this *basic* component to be referred to using the name *status*. The natural

method of providing a template for the type *state* with slots for the standard parts involving time and timing is to define type *state* as a record type with standard components. In addition to the standard components, the state of any particular timed automaton may have an arbitrary number of specialized state components. Being unable to define a parameterized record type in PVS with a variable number of components, we are constrained to use a single slot, to which we give the standard name *basic*, to represent these additional specialized components.

In translating steps from hand proofs to PVS, there are cases in which one must choose the appropriate version of a PVS strategy—say, to invoke a state invariant lemma—for the current context. While this sometimes might be done using a single, parameterized strategy, we wish to relieve the user of providing (the often considerable and technical) information that has the potential to be supplied automatically. Both of these problems could be eliminated from the user's point of view by an appropriate interface to PVS. However, there are other limitations of PVS as it stands as support for translating hand proofs that adding an interface cannot eliminate, such as the need to refer to particular assertion numbers when applying proof rules (see Section 5.2). One outcome of our study is the identification of a number of features, such as the ability to name assertions or identify them by contents, that would remove most or all of these other limitations if added to PVS.

An example difficulty that affects both specification and proof is the problem of reasoning about extended non-negative time. In both mathematical specification and hand proof, one can allow a larger time value to be subtracted from a smaller one with a negative number as the result. Because we have had to define type *time* as an abstract data type in PVS, *time* values cannot easily be viewed as overlapping real number values and therefore sharing some arithmetic. In fact, the result of subtracting a larger *time* value from a smaller one is undefined.[12] To accomplish specifications and proofs equivalent to the originals in [12], we have had to rephrase any equalities involving subtraction as equalities involving only addition. (See the definitions related to the Utility Property in Appendix B.3.)

The lack of parametric polymorphism in the type system of PVS has led to the following minor frustration: in timed automata, it is known that the time transition action changes only the *now* component of a state. Thus, other state components are equal for the states at the endpoints of a time transition interval. One cannot state a general lemma to this effect in PVS because these components do not all have the same type. One must instead prove a separate invariance lemma for each individual state component. With a standardized naming structure for these lemmas, the fact that they are separate can be masked on the strategy level by designing the strategy to invoke the appropriate lemma when passed the name of a state component as an argument.

9   TAME: Recent Developments

Since the publication of [1], our system for supporting the methods developed in this study was given the name TAME [2]. Further developments regarding TAME have been reported in [5], [4], and [6]. TAME has now been applied with some success to multiple examples of timed and non-timed automata, including the boiler controller in [21] (see [5, 3]), a vehicle control system from [36], a timed version of Fischer's algorithm from [23], the group communication service in [10, 9], and several examples of SCR specifications (see [6]).[13]

For the boiler controller and vehicle control system, TAME was extended by expanding the template conventions to cover specifying nondeterministic transitions using Hilbert's "choice" operator ϵ, extending the set of common theories to include a theory **real_thy** containing facts about real numbers helpful in reasoning about real arithmetic, and adding a new strategy to the standard strategies to simplify reasoning about ϵ. A slightly modified version of TAME's template and strategies was developed for reasoning about SCR specifications: for this purpose, it has proved more useful to represent transitions using a relation rather than a function. Although TAME was developed for timed automata, it can also be used without modification for non-timed automata, the group communication service being an example. TAME was used in a somewhat different fashion in connection with this example: many of the proofs of state invariants were undertaken with no hand proof to follow, or at best an extremely sketchy hand proof. While this resulted in

---

[12]We could have simply permitted negative *time* values; however, doing so would have complicated several of our definitions, and, therefore, proofs involving reasoning about time. For example, we would have had to explicitly state that the value of *now* for any state is nonnegative.

[13]For more on SCR specifications, see [15, 17].

21

```
Inv_6_3_A(s: states):bool = ( FORALL (r: train): status(r,s) = I => gate_status(s) = fully_down );

(""
(AUTO_INDUCT)
(("1" ;;Case enterI(Itrainof_action)
 (APPLY_SPECIFIC_PRECOND)
 (SUPPOSE "gate_status(prestate)=fully_up OR gate_status(prestate)=going_up")
 (("1" ;;Suppose [gate_status(prestate)=fully_up OR gate_status(prestate)=going_up]
  (APPLY_INV_LEMMA "6_1" "Itrainof_action")
  (TRY_SIMP))
 ("2" ;;Suppose not [gate_status(prestate)=fully_up OR gate_status(prestate)=going_up]
  (APPLY_INV_LEMMA "6_2" "Itrainof_action")
  (APPLY_INV_LEMMA "B_1_1")
  (TRY_SIMP))))
("2" ;;Case raise
 (APPLY_SPECIFIC_PRECOND)
 (INST "specific-precondition" "r_theorem")
 (APPLY_INV_LEMMA "5_1_1" "r_theorem")
 (APPLY_INV_LEMMA "5_1_3" "r_theorem")
 (TRY_SIMP))
("3" ;;Case up
 (APPLY_SPECIFIC_PRECOND)
 (TRY_SIMP))))
```

Figure 12: Updated TAME Proof of the Safety Property

some extra backtracking in the search for a mechanical proof, many of the proofs of simple properties were obtained fairly quickly. Feedback from proofs that did not succeed was provided to the authors of [10, 9], and proved helpful in suggesting additional state invariants needed as lemmas, for indicating that the statements of proposed invariants needed revision, of for suggesting the additional guidance needed in mechanizing the proof. For one complex property that was accompanied by a detailed hand proof, TAME helped to reveal an important missing case not covered by that proof. It should be noted that many extra theories for the specialized data types used in the specification of the group communication service, and corresponding proof strategies for reasoning about these types, were used to support the application of TAME in this context. In fact, completing the checking of all the invariant lemmas in [10, 9] awaits fuller development of these special data type theories. Therefore, while TAME has proved very useful for this application, the use of TAME for this and similar examples entails more than the usual overhead. However, this overhead and more would be needed in any ad hoc approach to mechanizing proofs of properties of specifications that use complex data types.

Based on a preliminary version of PVS with some added features—the ability to generate automatic labels for formulae in a sequent, the ability to generate automatic comments that are displayed both interactively and in saved proofs, the ability to probe into the content of formulae, and a few more atomic proof steps—we have solved some of the problems noted in Sections 6.3 and 8. In particular, comments labeling the base case and induction cases of a state invariant induction proof are now generated automatically, as are comments showing the content of various facts applied in the proof such as preconditions, previous invariant lemmas, or suppositions. Strategies that help the user avoid unnecessary branching in mechanized proofs have been or are being developed. Uniform strategies for the induction step and the application of invariant lemmas now exist, and with the added PVS features plus some documentation of PVS internals, were implemented internally to PVS, without an external interface. This work is discussed in [6]. As an example of how TAME proofs of state invariants now typically appear using the improved TAME strategies, Figure 12 shows the most recent version of the PVS proof of the Safety Property in Figure 11.[14] It should now be possible to extend TAME with the strategies proposed in Appendix E and other proof steps useful in ad hoc proofs.

Future plans for TAME include developing user interface support outside of PVS. An external interface would include support for entering the application-specific parts of specifications into the TAME template, and support for automatic translation of automata specifications in other specification languages into TAME

---

[14]For clarity, comments generated by APPLY_SPECIFIC_PRECOND and APPLY_INV_LEMMA have been omitted.

form. (There is a preliminary implementation of the latter for SCR specifications.) The interface would also handle some processing of a specification externally to PVS—for example, the construction of application-specific strategies such as SYSTIMPL_SIMP. In addition, we expect the interface to provide help to the user in the form of simple access to lemmas from all relevant theories and descriptions of existing TAME strategies.

So far, no proof support has been developed for proofs of simulation of one automaton by another. While it is possible to provide a template with slots for two automata for this purpose, accompanied by appropriate proof strategies, a problem arises when one wishes to apply a lemma previously proved for one of the automata in the course of a proof: this automaton has been specified and reasoned about in a separate theory. When theory instantiations become available in PVS as planned [32], support for simulation proofs is feasible in a form we desire.

Acknowledgments

References

[1] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*. IEEE Computer Society Press, 1996.

[2] M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. In *Work-In-Progress Proc. 1996 IEEE Real-Time Systems Symp. (RTSS'96)*, pages 3–6, 1996.

[3] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. Technical report, NRL, Wash., DC, 1997. In preparation.

[4] Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 33–48. Springer-Verlag, 1997.

[5] Myla Archer and Constance Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.

[6] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. Submitted for publication.

[7] R. Boyer and J Moore. *A Computational Logic*. Academic Press, 1979.

[8] 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.

[9] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. Technical Memo MIT/LCS/TM-570, Lab. for Comp. Sci., Mass. Inst. of Tech., October, 1997.

[10] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *Proc. Sixteenth Ann. ACM Symp. on Principles of Distributed Computing (PODC'97)*, pages 53–62, Santa Barbara, CA, August 1997.

[11] M. J. C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[12] 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 TR 7619, NRL, Wash., DC 1994.

[13] 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.

[14] C. Heitmeyer and D. Mandrioli, editors. *Formal Methods for Real-Time Computing*. Number 5 in Trends in Software. John Wiley & Sons, 1996.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[16] 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.

[17] Constance Heitmeyer, James Kirby, and Bruce Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.

[18] T. Henzinger and P. Ho. Hytech: The Cornell Hybrid Technology Tool. Technical report, Cornell University, 1995.

[19] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: the Automata-Theoretic Approach.* Princeton University Press, 1994.

[20] L. Lamport. Types are not harmless. Digital Systems Research Center, July 1995.

[21] Gunter Leeb and Nancy Lynch. Proving safety properties of the Steam Boiler Controller: Formal methods for industrial applications: A case study. In Jean-Raymond Abrial, Egon Boerger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1996.

[22] 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.

[23] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Massachusetts Institute of Technology, June 1995.

[24] 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.

[25] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distrib. Comput.*, 6:121–139, 1992.

[26] 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.

[27] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. To appear in *Information and Computation*.

[28] 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.

[29] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[30] M. Merritt, F. Modugno, and M. R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, eds., *CONCUR'91: 2nd Intern. Conference on Concurrency Theory*, vol. 527 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1991.

[31] 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.

[32] John Rushby. Private communication. NRL, Jan. 1997.

[33] 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.

[34] 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, Lect. Notes in Comp. Sci. 863.* Springer-Verlag, 1994.

[35] 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.

[36] Henri B. Weinberg. Correctness of vehicle control systems: A case study. Master's thesis, Massachusetts Institute of Technology, February 1996.

A Appendix. The Theory atexecs: Admissible Timed Executions

Below is the specification of the (parameterized) theory **atexecs** that is one of the underlying theories of our template specification for timed automata. The major purpose of this specification is the definition of the type *atexecs*. This definition, which closely follows the description of admissible timed executions in [12, 13], represents *atexecs* as a complex predicate subtype of a record type with three components: an action sequence, a trajectory sequence, and a time sequence. The associated predicate restricts these three-sequence combinations to those whose first trajectory starts with a start state, whose trajectory time bounds connect up, whose trajectory end points are connected by the corresponding actions, and whose time sequence satisfies a "greatest lower bound" property that ensures that it is non-Zeno: that is, as the index of the time points approaches infinity, so does the indexed time.

The definitions after that of *atexecs* set up two examples of lemmas about admissible timed executions, *last_event* and *first_event*, that eventually will be used to support useful specialized strategies. For example, the strategies using these lemmas will allow one in a single PVS step to follow hand proof steps of the form "let π be the last (or first) event before (or after) state *s* that has property *P*".

We note that, as with the template definition of time passage events, this part of the template is also more restrictive than the model described in Section 2.2. First, we enforce the condition that the value of *now* in any admissible timed execution must approach infinity by requiring the non-time-passage events to be infinite in number and to include a first one after any fixed finite time. In the general model, an admissible timed execution might have only finitely many non-time-passage events, with time approaching infinity through successive time-passage events. This difference is not really significant, since one can always add a dummy do-nothing non-time-passage action at infinitely many future points in a "finite" admissible timed execution. The second difference in the model is that we have added an axiom *trajectory_unique* (see the theory **opspec_atexecs_aux** in Appendix B.3) whose effect is to ensure that there are no repeating states in an admissible timed execution. A later state that repeats an earlier state could result only from a series of actions occuring in zero time; otherwise, the later state would have a different time component. We do not believe that ignoring executions with such "loops" is inordinately restrictive. In particular, an unsafe state lasting zero time should be unimportant, and properties (such as the utility property for **opspec**) involving time intervals should be unaffected in practice. If a real reason to permit repeating states arises, we could add the concept of "state occurrence" on which to base some of our reasoning.

```
atexecs[states, actions: TYPE,
        start: [states -> bool],
        now: [states -> {r:real | r>=0}],
        step?: [[states,actions,states] -> bool],
        nu: [{r:real | r>0} -> actions]] : THEORY

   BEGIN

      future: TYPE = {r:real | r>=0};

      k,m,n1,n2: VAR nat;
      z,t1,t2: VAR future;
      a: VAR actions;

      time_action?(a):bool = (EXISTS (t:future): t > 0 & a = nu(t));

      interval(t1,t2)(z):bool = (t1 <= z & z <= t2);

      time_path: TYPE = [# ftime,length:future, path:[(interval(ftime,ftime+length))->states] #];

      ltime(w:time_path):future = ftime(w) + length(w);

      trajectory?(w:time_path): bool =
             (FORALL (z1,z2: (interval(ftime(w),ltime(w))))):
                        z1<z2 => step?(path(w)(z1),nu(z2-z1),path(w)(z2)))
          & (FORALL (z: (interval(ftime(w),ltime(w))))): now(path(w)(z)) = z);

      trajectory: TYPE = (trajectory?);
```

fstate(w:trajectory):states = path(w)(ftime(w));

lstate(w:trajectory):states = path(w)(ltime(w));

time_seq: TYPE = {t:[nat -> future] | t(0)=0 & (FORALL (n1,n2): n1<=n2 => t(n1)<=t(n2))};

is_glb(z:future,t:time_seq,k:nat):bool = (t(k) <= z & (FORALL (m): m>k => z < t(m)));

has_glb(z:future,t:time_seq):bool = (EXISTS (k): is_glb(z,t,k));

pos_nat: TYPE = {n:nat | n > 0};

non_time_action: TYPE = {a:actions | (NOT (time_action?(a)))};
action_seq: TYPE = [pos_nat -> non_time_action];

traj_seq: TYPE = {w:[nat -> trajectory] | (FORALL (k): ltime(w(k)) = ftime(w(k+1)))};

atexecs: TYPE = {alpha : [# pi: action_seq, w: traj_seq, t: time_seq #] |
                start(fstate(w(alpha)(0)))
        & (FORALL (k): t(alpha)(k)= ftime(w(alpha)(k)))
        & (FORALL (k): step?(lstate(w(alpha)(k)), pi(alpha)(k+1), fstate(w(alpha)(k+1))))
        & (FORALL (z): has_glb(z,t(alpha))) };

% The definitions and lemmas below are auxiliary to the main theory **atexecs**. They serve to
% illustrate one of the conveniences of a theorem proving system with a higher order logic: one can
% state such results as *last_event* and *first_event* that say that if there exists an event
% before (after) some state that satisfies some property Q, then there is a last (first) such event.

in_trajectory(w:trajectory)(s:states):bool =
        (EXISTS (t:future): t >= ftime(w) & t <= ltime(w) & path(w)(t) = s);

precedes(alpha:atexecs)(s1,s2:states):bool =
        (now(s1) <= now(s2))
     & (EXISTS (n1,n2):(in_trajectory(w(alpha)(n2))(s2) & in_trajectory(w(alpha)(n2))(s2)
                        & n1 <= n2));

precedes_state(alpha:atexecs)(n1:posnat,s2:states):bool =
        (t(alpha)(n1) <= now(s2))
     & (EXISTS (n2): (in_trajectory(w(alpha)(n2))(s2) & n1 <= n2));

precedes_event(alpha:atexecs)(s1:states,n2:posnat):bool =
        (now(s1) <= t(alpha)(n2))
     & (EXISTS (n1): (in_trajectory(w(alpha)(n1))(s1) & n1 <= n2-1));

state_event_prop: TYPE = [atexecs,states,posnat -> bool];
Q: state_event_prop;

last_event: LEMMA (FORALL (alpha:atexecs, s:states, P:state_event_prop):
        (LET Q = (LAMBDA(alpha:atexecs, s:states, n:pos_nat):
                        (precedes_state(alpha)(n,s) & P(alpha,s,n)))
         IN (FORALL (n:posnat): (Q(alpha,s,n) =>
            (EXISTS (m: posnat): m >= n & Q(alpha,s,m)
                        & (FORALL (k: posnat): k >= m & Q(alpha,s,k) => k = m)))))));

first_event: LEMMA (FORALL (alpha:atexecs, s:states, P:state_event_prop):
        (LET Q = (LAMBDA(alpha:atexecs, s:states, n:pos_nat):
                        (precedes_event(alpha)(s,n) & P(alpha,s,n)))
         IN (FORALL (n:posnat): (Q(alpha,s,n) =>
            (EXISTS (m: posnat): m <= n & Q(alpha,s,m)
                        & (FORALL (k: posnat): k <= m & Q(alpha,s,k) => k = m)))))));
END atexecs

B Appendix. Specifying the GRC Timed Automata Solution in PVS

The specification in Figure 6 shows how the definition of the timed automaton *Trains* from [12, 13] is represented in PVS. *Trains* is a component of each of the timed automata used in deriving a solution to the Generalized Railroad Crossing problem in [12, 13]. Figure 6 shows only the declarations needed to define the automaton *Trains*; the full theory of *Trains* also contains lemmas that have been proved about the automaton.

In general, when using the template shown in Figure 4, it has proved convenient to organize the full theory of a timed automaton into several PVS theories that group definitions and lemmas or theorems according to their significance, and to import these separate theories either directly or indirectly into a trivial top-level theory. For any given automaton $< timed\_automaton\_name >$, we name the subsidiary theories according to the following conventions:

1. $< timed\_automaton\_name >$_decls contains the definitions required to instantiate the template;

2. $< timed\_automaton\_name >$_unique_aux contains the lemmas that document the fact that parameterized actions with distinct arguments are distinct;

3. $< timed\_automaton\_name >$_invariants contains the state invariant definitions and corresponding state invariant lemmas for the state invariants of $< timed\_automaton\_name >$;

4. $< timed\_automaton\_name >$_atexecs_aux contains the standard definitions and "IMPORTING atexecs" declaration to define the admissible timed executions of $< timed\_automaton\_name >$;

5. $< timed\_automaton\_name >$_atexecs contains the lemmas, theorems, and any supporting definitions concerning properties of the admissible timed executions of $< timed\_automaton\_name >$;

6. $< timed\_automaton\_name >$_strat_aux contains the lemmas needed to support the specialized strategies designed for use in the ad hoc portions of proofs of properties of $< timed\_automaton\_name >$;

7. $< timed\_automaton\_name >$ is the trivial top-level theory of $< timed\_automaton\_name >$ that imports all the subsidiary theories.

The subsidiary theories having the _aux suffix have the potential of being generated automatically from the information in the $< timed\_automaton\_name >$_decls theory. For example,

(A) $< timed\_automaton\_name >$_unique_aux can be generated from the declaration of the *actions* datatype;

(B) the definitions of *Now*, *Nu*, and *step?*, as well as the "IMPORTING atexecs" clause in the theory axspec_atexecs_aux, are of a standard form, and are technically part of (an extended form of) the template; and

(C) the lemmas in $< timed\_automaton\_name >$_strat_aux are identical in form for all applications.

The theory $< timed\_automaton\_name >$_unique_aux in (A) contains a set of lemmas about the uniqueness of actions whose content is not part of the knowledge incorporated in existing PVS strategies, but which are provable in PVS.[15] In fact, the proofs of these lemmas could also be generated automatically.

The syntactic content of the theories in (B) and (C) is fixed, and can be considered an extension of the template. Note that the lemmas in the theory $< timed\_automaton\_name >$_strat_aux need to be proved at some point, in order to guarantee the soundness of proofs obtained using strategies that depend on the lemmas. These lemmas need to be proved in an environment in which type of *trans* is known. One method for providing such an environment is to import a theory into $< timed\_automaton\_name >$_strat_aux,

---

[15]The information that they contain is one example of the type of knowledge that is "obvious" to a human but not to PVS. Note that the truth of this information depends on the fact that there are no equations postulated among elements of the data type *actions*. PVS does not support the declaration of such equations, although other theorem proving systems, including LP, do allow them.

directly or indirectly, in which *trans* is declared (following our naming conventions, this theory should be < *timed_automaton_name* >_decls). This is the method used in the specifications in this Appendix. Note that it requires proving the lemmas anew for each timed automaton. Another method would be to define the lemmas within a theory to which *trans* is passed in as a parameter of known type. This second method is used in our second template, which is shown in Appendix F. Using this method, the lemmas need to be proved only once, at the top level.

The connections between the lemmas in the theory < *timed_automaton_name* >_strat_aux and our domain specific strategies are made explicit in Appendix C.

Below, we present four full PVS theories in the order that the corresponding timed automata are defined in [12, 13]: **trains**, the theory of *Trains*; **axspec**, the theory of *AxSpec*; **opspec**, the theory of *OpSpec*; and **systimpl**, the theory of *SystImpl*. Not every one of these timed automata required all of the subtheories listed above. For the timed automata *Trains* and *Systimpl*, we have only needed the theories described in (1), (2), (3), and (7). For *AxSpec*, only (1), (2), (4), (5), and (7) are needed. *OpSpec* requires all seven subtheories.

B.1 Appendix. The Full Theory of *Trains* in PVS

The specification in Figure 6 shows how the definition of the timed automaton *Trains* from [12, 13] is represented in PVS. The full theory **trains** of *Trains* also includes one invariant lemma: lemma_3_1. In accordance with our naming conventions, lemma_3_1 appears in the subsidiary theory **trains_invariants**.

---

trains_decls: 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): (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_decls

trains_unique_aux: THEORY

BEGIN

  IMPORTING **trains_decls**

  enterR_unique: LEMMA (FORALL (r1, r2: train): (enterR(r1) = enterR(r2) => r1 = r2));
  enterI_unique: LEMMA (FORALL (r1, r2: train): (enterI(r1) = enterI(r2) => r1 = r2));
  exit_unique: LEMMA (FORALL (r1, r2: train): (exit(r1) = exit(r2) => r1 = r2));
  nu_unique: LEMMA (FORALL (t1, t2: (fintime?)): (nu(t1) = nu(t2) => t1 = t2));

END trains_unique_aux

trains_invariants: THEORY

BEGIN

  IMPORTING **trains_unique_aux**

  Inv_3_1(s: states):bool =
      (FORALL (r: train): (status(r,s) = P =>
                  first(s)(enterI(r)) + eps_2 - eps_1 = last(s)(enterI(r))));

  lemma_3_1: LEMMA (FORALL (s: states): reachable(s) => Inv_3_1(s));

END trains_invariants

trains: THEORY

BEGIN

  IMPORTING **trains_invariants**

END trains

B.2 Appendix. Representing the Automaton *AxSpec* in PVS

Below, we present the theory **axspec**, which is the translation into PVS of the theory of the automaton *AxSpec* from [12, 13].

The automaton *AxSpec* includes axiomatic versions of the Safety and Utility Properties as part of its definition, so we wish to include these in the full corresponding PVS theory **axspec**. Since the Safety and Utility axioms restrict the admissible timed executions of *AxSpec*, they are defined in the subsidiary theory **axspec_atexecs**.

Note that the use of subtraction in the time expressions appearing in the inequalities involved in the definition of the Utility Property axiom has been avoided by following the convention of replacing the inequalities with equivalent ones involving only addition. Doing this results in reducing the number and complexity of the cases to be considered in PVS proofs relying on these inequalities. It also ensures that the inequalities have the same semantics as if they permitted negative values to result from subtractions, as is typically assumed in hand proofs involving values in $R^{\geq 0} \cup \{\infty\}$. We have included the original formulations of the definitions for comparison. This particular convention for fitting an automaton specification to our PVS template could be automated.

---

axspec_decls: THEORY
BEGIN

    train: TYPE;

    r,r1: VAR train;

    IMPORTING **time_thy**

    t, delta_t: VAR time;
    eps_1, eps_2, gamma_down, gamma_up, xi_1, xi_2, delta: (fintime?);

    actions : DATATYPE
      BEGIN
            nu(timeof:(fintime?)): nu?
            enterR(Rtrainof:train): enterR?
            enterI(Itrainof:train): enterI?
            exit(Etrainof:train): exit?
            lower: lower?
            raise: raise?
            up: up?
            down: down?
      END actions;

    a: VAR actions;

    train_status: TYPE = not_here,P,I;

    gate_status: TYPE = fully_up,fully_down,going_up,going_down;

    MMTstates: TYPE = [# trains_part: [train -> train_status], gate_part: gate_status #];

    IMPORTING **states**[actions,MMTstates,time,fintime?]

    s1: VAR states;

    status(r:train, s:states):train_status = trains_part(basic(s))(r);

    gate_status(s:states):gate_status = gate_part(basic(s));

    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)))

$$\&\ \text{now}(s) + \text{delta\_t} <= \text{last}(s)(\text{up})$$
$$\&\ \text{now}(s) + \text{delta\_t} <= \text{last}(s)(\text{down})),$$

enterR(r): status(r,s) = not_here,
enterI(r): status(r,s) = P & first(s)(a) <= now(s),
exit(r): status(r,s) = I,
lower: true,
raise: true,
up: gate_status(s) = going_up,
down: gate_status(s) = going_down
ENDCASES;

trans (a:actions, s:states):states =
    CASES a OF
      nu(delta_t): s WITH [now := now(s)+delta_t],
      enterR(r): s WITH [basic := basic(s) WITH
                  [trains_part := trains_part(basic(s)) WITH [r := P]],
                  first := first(s) WITH [(enterI(r)) := now(s)+eps_1],
                  last := last(s) WITH [(enterI(r)) := now(s)+eps_2]],
      enterI(r): s WITH [basic := basic(s) WITH
                  [trains_part := trains_part(basic(s)) WITH [r := I]],
                  first := first(s) WITH [(enterI(r)) := zero],
                  last := last(s) WITH [(enterI(r)) := infinity]],
      exit(r): s WITH [basic := basic(s) WITH
                  [trains_part := trains_part(basic(s)) WITH [r := not_here]]],
      lower: IF gate_status (s) = fully_up OR gate_status(s) = going_up
          THEN s WITH [basic := basic(s) WITH [gate_part := going_down],
                  last := last(s) WITH
                    [down := now(s) + gamma_down, up := infinity]]
        ELSE s ENDIF,
      raise: IF gate_status(s) = fully_down OR gate_status(s) = going_down
          THEN s WITH [basic := basic(s) WITH [gate_part := going_up],
                  last := last(s) WITH
                    [up := now(s) + gamma_up, down := infinity]]
        ELSE s ENDIF,
      up: s WITH [basic := basic(s) WITH [gate_part := fully_up],
            last := last(s) WITH [up := infinity]],
      down: s WITH [basic := basic(s) WITH [gate_part := fully_down],
            last := last(s) WITH [down := infinity]]
    ENDCASES

enabled (a:actions, s:states):bool = enabled_general(a,s) & enabled_specific(a,s);

start (s:states):bool =
    s = (# basic := (# trains_part := (LAMBDA r: not_here), gate_part := fully_up #),
        now := zero,
        first := (LAMBDA a: zero),
        last := (LAMBDA a: infinity) #);

IMPORTING machine[states, actions, enabled, trans, start]

END axspec_decls

axspec_unique_aux: THEORY

BEGIN

IMPORTING **axspec_decls**

enterR_unique: LEMMA (FORALL (r1, r2: train): (enterR(r1) = enterR(r2) => r1 = r2));
enterI_unique: LEMMA (FORALL (r1, r2: train): (enterI(r1) = enterI(r2) => r1 = r2));

31

exit_unique: LEMMA (FORALL (r1, r2: train): (exit(r1) = exit(r2) => r1 = r2));
nu_unique: LEMMA (FORALL (t1, t2: (fintime?)): (nu(t1) = nu(t2) => t1 = t2));

  END axspec_unique_aux

axspec_atexecs_aux: THEORY

  BEGIN

    IMPORTING axspec_unique_aux

    step? (s1:states, a:actions, s2:states): bool = enabled(a,s1) & s2 = trans(a,s1);

    Now (s: states): {z:real | z>=0} = dur(now(s));

    Nu (z: {z:real | z>0}): actions = nu(fintime(z: {z:real | z>=0}));

    IMPORTING atexecs [states, actions, start, Now, step?, Nu]

  END axspec_atexecs_aux

axspec_atexecs: THEORY

  BEGIN

    IMPORTING axspec_atexecs_aux

    safety: AXIOM (FORALL (alpha: atexecs): (FORALL (s: states):
      (in_atexec(alpha)(s) => ((EXISTS (r:train): status(r,s)=I) => gate_status(s)=fully_down))));

% utility_prop_a (alpha:atexecs, s:states): bool =
%   (EXISTS (s1:states):
%   (precedes(alpha)(s1,s) & (EXISTS (r:train): status(r,s1) = I) & now(s1) >= now(s) − xi_2));

    utility_prop_a (alpha:atexecs, s:states): bool =
      (EXISTS (s1:states):
        (precedes(alpha)(s1,s) & (EXISTS (r:train): status(r,s1) = I) & now(s1) + xi_2 >= now(s)));

    utility_prop_b (alpha:atexecs, s:states): bool =
      (EXISTS (s1:states):
        (precedes(alpha)(s,s1) & (EXISTS (r:train): status(r,s1) = I) & now(s1) <= now(s) + xi_1));

% utility_prop_c (alpha:atexecs, s:states): bool =
%   (EXISTS (s1,s2:states):
%   precedes(alpha)(s1,s) & precedes(alpha)(s,s2)
%   & (EXISTS (r:train):status(r,s1)=I) & (EXISTS (r:train):status(r,s2)=I)
%   & now(s2) − now(s1) <= xi_1 + xi_2 + delta);

    utility_prop_c (alpha:atexecs, s:states): bool =
      (EXISTS (s1,s2:states):
        precedes(alpha)(s1,s) & precedes(alpha)(s,s2)
      & (EXISTS (r:train):status(r,s1)=I) & (EXISTS (r:train):status(r,s2)=I)
      & now(s2) <= xi_1 + xi_2 + delta + now(s1));

    utility: AXIOM (FORALL (alpha: atexecs): (FORALL (s: states):
      ((in_atexec(alpha)(s) & NOT(gate_status(s) = fully_up)) =>
        (utility_prop_a(alpha,s) OR utility_prop_b(alpha,s) OR utility_prop_c(alpha,s))))));

  END axspec_atexecs

axspec : THEORY

  BEGIN

    IMPORTING axspec_atexecs

  END axspec

B.3 Appendix. The Timed Automaton *OpSpec* in PVS: Version 1

The timed automaton *OpSpec* defined in [12, 13] is the composition of three timed automata: *Trains*, *Gate*, and *CompSpec*. In our study, we have first composed these automata by hand into a a single timed automaton, which we then defined by completing our template specification.

The complete theory of OpSpec includes several state invariant lemmas and a few results about admissible timed executions of OpSpec—most notably, two major theorems, the Safety Property and the Utility Property for *OpSpec*, which appear in the theory **opspec_atexecs**. This theory also contains a major lemma (*lemma_E_1*) and three definitions needed to state and prove the Utility Property. A heavily annotated version of the PVS proof of *lemma_E_1* (which corresponds to Lemma E.1 in [12]) appears in Appendix E.

```
opspec_decls: THEORY

BEGIN

    train: TYPE

    r,r1: VAR train

    IMPORTING time_thy

    beta_posreal: {r:real | r > 0};
    delta_t: VAR (fintime?)
    eps_1, eps_2, gamma_down, gamma_up, xi_1, xi_2, delta: (fintime?)
    beta:(fintime?) = fintime(beta_posreal:{r:real | r >= 0});

    const_facts: AXIOM
          (eps_1 <= eps_2
        & eps_1 > gamma_down
        & xi_1 >= gamma_down + beta + eps_2 - eps_1
        & xi_2 >= gamma_up);

    actions : DATATYPE

      BEGIN

            nu(timeof:(fintime?)): nu?
            enterR(Rtrainof:train): enterR?
            enterI(Itrainof:train): enterI?
            exit(Etrainof:train): exit?
            lower: lower?
            raise: raise?
            up: up?
            down: down?

      END actions;

    a: VAR actions;

    train_status: TYPE = {not_here,P,I};

    gate_status: TYPE = {fully_up,fully_down,going_up,going_down};

    MMTstates: TYPE = [# trains_part: [train -> train_status],
                          gate_part: gate_status,
                          last_1_part, last_2_up_part, last_2_I_part: time #];

    IMPORTING states[actions,MMTstates,time,fintime?]

    s1: VAR states;

    status(r:train, s:states):train_status = trains_part(basic(s))(r);
```

33

gate_status(s:states):gate_status = gate_part(basic(s));

last_1(s:states):time = last_1_part(basic(s));

last_2_up(s:states):time = last_2_up_part(basic(s));

last_2_I(s:states):time = last_2_I_part(basic(s));

OKstate? (s:states): bool =
  ((EXISTS (r:train): status(r,s) = I) => gate_status(s) = fully_down);

OKstates: TYPE = (OKstate?);

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)))
           & now(s) + delta_t <= last(s)(up)
           & now(s) + delta_t <= last(s)(down)
           & now(s) + delta_t <= last_1(s)
           & now(s) + delta_t <= last_2_I(s)),
      enterR(r): status(r,s) = not_here,
      enterI(r): status(r,s) = P & first(s)(a) <= now(s),
      exit(r): status(r,s) = I,
      lower: true,
      raise: true,
      up: gate_status(s) = going_up,
      down: gate_status(s) = going_down
    ENDCASES;

trans (a:actions, s:states):states =
    CASES a OF
      nu(delta_t): s WITH [now := now(s)+delta_t],
      enterR(r): s WITH [basic := basic(s) WITH
             [trains_part := trains_part(basic(s)) WITH [r := P]],
           first := first(s) WITH [(enterI(r)) := now(s)+eps_1],
           last := last(s) WITH [(enterI(r)) := now(s)+eps_2]],
      enterI(r): s WITH [basic := basic(s) WITH
             [trains_part := trains_part(basic(s)) WITH [r := I],
             last_1_part := infinity,
             last_2_up_part := infinity,
             last_2_I_part := infinity],
           first := first(s) WITH [(enterI(r)) := zero],
           last := last(s) WITH [(enterI(r)) := infinity]],
      exit(r): LET s1 = s WITH [basic := basic(s) WITH
               [trains_part := trains_part(basic(s)) WITH
               [r := not_here]]]
        IN IF (FORALL (r1: train): (NOT (r1 = r)) => (NOT status(r1,s) = I))
          THEN s1 WITH [basic := basic(s1) WITH
               [last_2_up_part := now(s) + xi_2,
               last_2_I_part := now(s) + xi_2 + delta + xi_1]]
         ELSE s1 ENDIF,
      lower: IF gate_status (s) = fully_up OR gate_status(s) = going_up
        THEN LET s1 = s WITH
              [basic := basic(s) WITH [gate_part := going_down],

```
                              last := last(s) WITH
                                [down := now(s) + gamma_down, up := infinity]]
                    IN IF last_1_part(basic(s)) = infinity
                         THEN s1 WITH
                                    [basic:= basic(s1) WITH [last_1_part:= now(s)+xi_1]]
                         ELSE s1 ENDIF
              ELSE s ENDIF,
        raise: IF gate_status(s) = fully_down OR gate_status(s) = going_down
                 THEN s WITH [basic := basic(s) WITH [gate_part := going_up],
                                last := last(s) WITH
                                    [up := now(s) + gamma_up, down := infinity]]
                 ELSE s ENDIF,
        up: LET s1 = s WITH [basic := basic(s) WITH [gate_part := fully_up],
                                last := last(s) WITH [up := infinity]]
             IN IF now(s) <= last_2_up_part(basic(s))
                  THEN s1 WITH [basic := basic(s1) WITH
                                       [last_2_up_part:= infinity, last_2_I_part:= infinity]]
                  ELSE s1 ENDIF,
        down: s WITH [basic := basic(s) WITH [gate_part := fully_down],
                        last := last(s) WITH [down := infinity]]
      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 := (# trains_part := (LAMBDA r: not_here),
                         gate_part := fully_up,
                         last_1_part := infinity,
                         last_2_up_part := infinity,
                         last_2_I_part := infinity #),
              now := zero,
              first := (LAMBDA a: zero),
              last := (LAMBDA a: infinity) #)


rans, start]

    END opspec_decls


 opspec_unique_aux: THEORY

    BEGIN

    IMPORTING opspec_decls

    enterR_unique: LEMMA (FORALL (r1, r2: train): (enterR(r1) = enterR(r2) => r1 = r2));
    enterI_unique: LEMMA (FORALL (r1, r2: train): (enterI(r1) = enterI(r2) => r1 = r2));
    exit_unique: LEMMA (FORALL (r1, r2: train): (exit(r1) = exit(r2) => r1 = r2));
    nu_unique: LEMMA (FORALL (t1, t2: (fintime?)): (nu(t1) = nu(t2) => t1 = t2));

    END opspec_unique_aux

 opspec_invariants: THEORY

    BEGIN

    IMPORTING opspec_unique_aux
```

Inv_4_1_1(s: states):bool =
(EXISTS (r: train): (status(r,s) = I)) => gate_status(s) = fully_down;

lemma_4_1_1: LEMMA (FORALL (s: states): reachable(s) => Inv_4_1_1(s));

Inv_4_1_2(s: states):bool = (last_2_up(s) + delta + xi_1 = last_2_I(s));

lemma_4_1_2: LEMMA (FORALL (s: states): reachable(s) => Inv_4_1_2(s));

Inv_4_2_1(s: states):bool = (now(s) <= last_1(s));

lemma_4_2_1: LEMMA (FORALL (s: states): reachable(s) => Inv_4_2_1(s));

Inv_4_2_2(s: states):bool = (now(s) <= last_2_I(s));

lemma_4_2_2: LEMMA (FORALL (s: states): reachable(s) => Inv_4_2_2(s));

Inv_4_2_3(s: states):bool =
(NOT (last_1(s) = infinity)) => (last_1(s) <= now(s) + xi_1);

lemma_4_2_3: LEMMA (FORALL (s: states): reachable(s) => Inv_4_2_3(s));

Inv_4_2_4(s: states):bool =
(NOT (last_2_I(s) = infinity)) => (last_2_I(s) <= now(s) + xi_2 + delta + xi_1);

lemma_4_2_4: LEMMA (FORALL (s: states): reachable(s) => Inv_4_2_4(s));

Inv_4_2_5(s: states):bool =
(NOT (last_2_up(s) = infinity)) => (last_2_up(s) <= now(s) + xi_2);

lemma_4_2_5: LEMMA (FORALL (s: states): reachable(s) => Inv_4_2_5(s));

END opspec_invariants

opspec_atexecs_aux: THEORY

BEGIN

IMPORTING **opspec_invariants**

step? (s1:states, a:actions, s2:states): bool = enabled(a,s1) & s2 = trans(a,s1);

Now (s: states): {z:real | z>=0} = dur(now(s));

Nu (z: {z:real | z>0}): actions = nu(fintime(z: {z:real | z>=0}));

IMPORTING **atexecs** [states, actions, start, Now, step?, Nu]

A: var atexecs;

reach_equiv: LEMMA (FORALL (s: states): (FORALL (n: nat):
steps_reach(n, s) => reachable(s)));

reach_equiv_2: LEMMA (FORALL (s: states): (EXISTS (n: nat):
steps_reach(n, s)) => reachable(s));

reachability: LEMMA (FORALL (alpha: atexecs): (FORALL (s: states):
(in_atexec(alpha)(s) => reachable(s))));

last_1_interval_0: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat): (FORALL (s: states):
(in_trajectory(w(alpha)(j))(s) => (last_1(s) = last_1(fstate(w(alpha)(j))))))));

last_1_interval: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat):
(last_1(lstate(w(alpha)(j))) = last_1(fstate(w(alpha)(j)))))));

gate_status_interval_0: LEMMA (FORALL(alpha:atexecs):(FORALL(j:nat):(FORALL(s:states):
(in_trajectory(w(alpha)(j))(s) => (gate_status(s) = gate_status(fstate(w(alpha)(j))))))));

gate_status_interval: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat):
  (gate_status(lstate(w(alpha)(j))) = gate_status(fstate(w(alpha)(j))))));

status_interval_0: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat): (FORALL (s: states):
  (in_trajectory(w(alpha)(j))(s)
    => (FORALL (r: train): (status(r,s) = status(r,fstate(w(alpha)(j)))))))));

status_interval: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat): (FORALL (r: train):
  (status(r,lstate(w(alpha)(j))) = status(r,fstate(w(alpha)(j)))))));

last_2_up_interval_0: LEMMA (FORALL (alpha:atexecs): (FORALL (j:nat): (FORALL (s:states):
  (in_trajectory(w(alpha)(j))(s) => (last_2_up(s) = last_2_up(fstate(w(alpha)(j))))))));

last_2_up_interval: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat):
  (last_2_up(lstate(w(alpha)(j))) = last_2_up(fstate(w(alpha)(j))))));

last_2_I_interval_0: LEMMA (FORALL (alpha:atexecs): (FORALL (j:nat): FORALL (s:states):
  (in_trajectory(w(alpha)(j))(s) => (last_2_I(s) = last_2_I(fstate(w(alpha)(j))))))));

last_2_I_interval: LEMMA (FORALL (alpha: atexecs): (FORALL (j: nat):
  (last_2_I(lstate(w(alpha)(j))) = last_2_I(fstate(w(alpha)(j))))));

trajectory_unique: AXIOM (FORALL(alpha:atexecs):(FORALL (s:states):(FORALL (n1,n2:nat):
  (in_trajectory(w(alpha)(n1))(s) & in_trajectory(w(alpha)(n2))(s)) => n1 = n2)));

last_2_I_fixed: LEMMA (FORALL (alpha: atexecs): (FORALL (j,k: nat):
  (j <= k
  & (FORALL (m: nat): (j < m & m <= k) => (not(exit?(pi(alpha)(m)))
                                          & not(enterI?(pi(alpha)(m)))
                                          & not(up?(pi(alpha)(m))))))
    => last_2_I(fstate(w(alpha)(k))) = last_2_I(fstate(w(alpha)(j)))));

last_2_up_fixed: LEMMA (FORALL (alpha: atexecs): (FORALL (j,k: nat):
  (j <= k
  & (FORALL (m: nat): (j < m & m <= k) => (not(exit?(pi(alpha)(m)))
                                          & not(enterI?(pi(alpha)(m)))
                                          & not(up?(pi(alpha)(m))))))
    => last_2_up(fstate(w(alpha)(k))) = last_2_up(fstate(w(alpha)(j)))));

END opspec_atexecs_aux

opspec_strat_aux: THEORY

BEGIN

IMPORTING **opspec_atexecs_aux**

event_times: LEMMA (FORALL (alpha:atexecs, n:nat):
  ftime(w(alpha)(n)) = t(alpha)(n) &
  Now(path(w(alpha)(n))(t(alpha)(n))) = t(alpha)(n) &
  Now(path(w(alpha)(n))(ftime(w(alpha)(n)))) = t(alpha)(n) &
  dur(now(path(w(alpha)(n))(t(alpha)(n)))) = t(alpha)(n) &
  dur(now(path(w(alpha)(n))(ftime(w(alpha)(n))))) = t(alpha)(n) &
  (n > 0 => (ltime(w(alpha)(n - 1)) = t(alpha)(n) &
             ftime(w(alpha)(n - 1)) + length(w(alpha)(n−1)) = t(alpha)(n) &
             Now(path(w(alpha)(n - 1))(t(alpha)(n))) = t(alpha)(n) &
             Now(path(w(alpha)(n - 1))(ftime(w(alpha)(n)))) = t(alpha)(n) &
             dur(now(path(w(alpha)(n - 1))(t(alpha)(n)))) = t(alpha)(n) &
             dur(now(path(w(alpha)(n - 1))(ftime(w(alpha)(n)))))) = t(alpha)(n))));

same_states: LEMMA (FORALL (alpha:atexecs, n:nat):
  fstate(w(alpha)(n)) = path(w(alpha)(n))(t(alpha)(n)) &
  lstate(w(alpha)(n)) = path(w(alpha)(n))(t(alpha)(n + 1)) &
  (n > 0 => lstate(w(alpha)(n−1)) = path(w(alpha)(n−1))(t(alpha)(n))) &
  trans(pi(alpha)(n + 1), lstate(w(alpha)(n))) = path(w(alpha)(n + 1))(t(alpha)(n + 1)) &
  trans(pi(alpha)(n + 1), path(w(alpha)(n))(t(alpha)(n + 1)))
      = path(w(alpha)(n + 1))(t(alpha)(n + 1)) &
  LET (dt:{z:real|z>=0}) = t(alpha)(n + 1) − t(alpha)(n) IN
      (dt >= 0 & (dt > 0 => trans(nu(fintime(dt)),fstate(w(alpha)(n))) = lstate(w(alpha)(n)))));

event_times_1: LEMMA (FORALL (alpha:atexecs, n:nat):
  ftime(w(alpha)(n)) = t(alpha)(n));
event_times_2: LEMMA (FORALL (alpha:atexecs, n:nat):
  (t(alpha)(n+1) − t(alpha)(n) >= 0) = TRUE);
event_times_3: LEMMA (FORALL (alpha:atexecs, n:nat):
  (t(alpha)(n+1) >= t(alpha)(n)) = TRUE);
event_times_4: LEMMA (FORALL (alpha:atexecs, n:nat):
  (t(alpha)(n) <= t(alpha)(n+1)) = TRUE);
event_times_5: LEMMA (FORALL (alpha:atexecs, n:nat):
  Now(path(w(alpha)(n))(t(alpha)(n))) = t(alpha)(n));
event_times_6: LEMMA (FORALL (alpha:atexecs, n:nat):
  Now(path(w(alpha)(n))(ftime(w(alpha)(n)))) = t(alpha)(n));
event_times_7: LEMMA (FORALL (alpha:atexecs, n:nat):
  dur(now(path(w(alpha)(n))(t(alpha)(n)))) = t(alpha)(n));
event_times_8: LEMMA (FORALL (alpha:atexecs, n:nat):
  dur(now(path(w(alpha)(n))(ftime(w(alpha)(n))))) = t(alpha)(n));
event_times_9: LEMMA (FORALL (alpha:atexecs, n:nat):
  now(path(w(alpha)(n))(t(alpha)(n))) = fintime(t(alpha)(n)));
event_times_10: LEMMA (FORALL (alpha:atexecs, n:nat):
  now(path(w(alpha)(n))(ftime(w(alpha)(n)))) = fintime(t(alpha)(n)));
event_times_11: LEMMA (FORALL (alpha:atexecs, n:nat):
  ltime(w(alpha)(n)) = t(alpha)(n+1));
event_times_12: LEMMA (FORALL (alpha:atexecs, n:nat):
  ftime(w(alpha)(n)) + length(w(alpha)(n)) = t(alpha)(n+1));
event_times_13: LEMMA (FORALL (alpha:atexecs, n:nat):
  length(w(alpha)(n)) + ftime(w(alpha)(n)) = t(alpha)(n+1));
event_times_14: LEMMA (FORALL (alpha:atexecs, n:nat):
  t(alpha)(n) + length(w(alpha)(n)) = t(alpha)(n+1));
event_times_15: LEMMA (FORALL (alpha:atexecs, n:nat):
  length(w(alpha)(n)) + t(alpha)(n) = t(alpha)(n+1));
event_times_16: LEMMA (FORALL (alpha:atexecs, n:nat):
  Now(path(w(alpha)(n))(t(alpha)(n+1))) = t(alpha)(n+1));
event_times_17: LEMMA (FORALL (alpha:atexecs, n:nat):
  Now(path(w(alpha)(n))(ftime(w(alpha)(n+1)))) = t(alpha)(n+1));
event_times_18: LEMMA (FORALL (alpha:atexecs, n:nat):
  dur(now(path(w(alpha)(n))(t(alpha)(n+1)))) = t(alpha)(n+1));
event_times_19: LEMMA (FORALL (alpha:atexecs, n:nat):
  dur(now(path(w(alpha)(n))(ftime(w(alpha)(n+1))))) = t(alpha)(n+1));
event_times_20: LEMMA (FORALL (alpha:atexecs, n:nat):
  now(path(w(alpha)(n))(t(alpha)(n+1))) = fintime(t(alpha)(n+1)));
event_times_21: LEMMA (FORALL (alpha:atexecs, n:nat):
  now(path(w(alpha)(n))(ftime(w(alpha)(n+1)))) = fintime(t(alpha)(n+1)));
same_states_22: LEMMA (FORALL (alpha:atexecs, n:posnat):
  (t(alpha)(n) − t(alpha)(n−1) >= 0) = TRUE);

event_times_23: LEMMA (FORALL (alpha:atexecs, n:posnat):
   (t(alpha)(n) >= t(alpha)(n−1)) = TRUE);
event_times_24: LEMMA (FORALL (alpha:atexecs, n:posnat):
   (t(alpha)(n−1) <= t(alpha)(n)) = TRUE);
event_times_25: LEMMA (FORALL (alpha:atexecs, n:posnat):
   ltime(w(alpha)(n−1)) = t(alpha)(n));
event_times_26: LEMMA (FORALL (alpha:atexecs, n:posnat):
   ftime(w(alpha)(n−1)) + length(w(alpha)(n−1)) = t(alpha)(n));
event_times_27: LEMMA (FORALL (alpha:atexecs, n:posnat):
   length(w(alpha)(n−1)) + ftime(w(alpha)(n−1)) = t(alpha)(n));
event_times_28: LEMMA (FORALL (alpha:atexecs, n:posnat):
   t(alpha)(n−1) + length(w(alpha)(n−1)) = t(alpha)(n));
event_times_29: LEMMA (FORALL (alpha:atexecs, n:posnat):
   length(w(alpha)(n−1)) + t(alpha)(n−1) = t(alpha)(n));
event_times_30: LEMMA (FORALL (alpha:atexecs, n:posnat):
   Now(path(w(alpha)(n−1))(t(alpha)(n))) = t(alpha)(n));
event_times_31: LEMMA (FORALL (alpha:atexecs, n:posnat):
   Now(path(w(alpha)(n−1))(ftime(w(alpha)(n)))) = t(alpha)(n));
event_times_32: LEMMA (FORALL (alpha:atexecs, n:posnat):
   dur(now(path(w(alpha)(n−1))(t(alpha)(n)))) = t(alpha)(n));
event_times_33: LEMMA (FORALL (alpha:atexecs, n:posnat):
   dur(now(path(w(alpha)(n−1))(ftime(w(alpha)(n))))) = t(alpha)(n));
event_times_34: LEMMA (FORALL (alpha:atexecs, n:posnat):
   now(path(w(alpha)(n−1))(t(alpha)(n))) = fintime(t(alpha)(n)));
event_times_35: LEMMA (FORALL (alpha:atexecs, n:posnat):
   now(path(w(alpha)(n−1))(ftime(w(alpha)(n)))) = fintime(t(alpha)(n)));

trans_facts: LEMMA (FORALL (alpha:atexecs, n:nat):
   trans(pi(alpha)(n+1), lstate(w(alpha)(n))) = path(w(alpha)(n+1))(t(alpha)(n+1)) &
   trans(pi(alpha)(n+1), path(w(alpha)(n))(t(alpha)(n+1)))
      = path(w(alpha)(n+1))(t(alpha)(n+1)) &
   ((t(alpha)(n+1) − t(alpha)(n) > 0) =>
   trans(nu(fintime((t(alpha)(n+1) − t(alpha)(n)):{r:real|r>=0})), fstate(w(alpha)(n)))
      = lstate(w(alpha)(n))) &
   (n > 0 =>
   trans(pi(alpha)(n), lstate(w(alpha)(n−1))) = path(w(alpha)(n))(t(alpha)(n)) &
   trans(pi(alpha)(n), path(w(alpha)(n−1))(t(alpha)(n))) = path(w(alpha)(n))(t(alpha)(n)) &
   ((t(alpha)(n) − t(alpha)(n−1) > 0) =>
   trans(nu(fintime((t(alpha)(n) − t(alpha)(n−1)):{r:real|r>=0})), fstate(w(alpha)(n−1)))
      = lstate(w(alpha)(n−1)))));

same_states_1: LEMMA (FORALL (alpha:atexecs, n:nat):
   fstate(w(alpha)(n)) = path(w(alpha)(n))(t(alpha)(n)));
same_states_2: LEMMA (FORALL (alpha:atexecs, n:nat):
   lstate(w(alpha)(n)) = path(w(alpha)(n))(t(alpha)(n+1)));
same_states_3: LEMMA (FORALL (alpha:atexecs, n:posnat):
   lstate(w(alpha)(n−1)) = path(w(alpha)(n−1))(t(alpha)(n)));

reachable_states: LEMMA (FORALL (alpha:atexecs, n:nat):
   reachable(fstate(w(alpha)(n))) &
   reachable(lstate(w(alpha)(n))) &
   reachable(path(w(alpha)(n))(ftime(w(alpha)(n)))) &
   reachable(path(w(alpha)(n))(ltime(w(alpha)(n)))) &
   reachable(path(w(alpha)(n))(t(alpha)(n))) &
   reachable(path(w(alpha)(n))(t(alpha)(n+1))));

glb_fact: LEMMA (FORALL (alpha:atexecs, B:future):
    (EXISTS (k:nat): t(alpha)(k) <= B & B < t(alpha)(k+1)));

time_relation: LEMMA (FORALL (alpha:atexecs, t1,t2:nat):
    (t1 <= t2 => t(alpha)(t1) <= t(alpha)(t2)) & (t2 <= t1 => t(alpha)(t2) <= t(alpha)(t1)));

END opspec_strats_aux

opspec_atexecs: THEORY

BEGIN

IMPORTING **opspec_strat_aux**

lemma_E_1: LEMMA (FORALL (alpha: atexecs): (FORALL (n: pos_nat):
        pi(alpha)(n) = lower
          & (gate_status(lstate(w(alpha)(n−1))) = going_up OR
            gate_status(lstate(w(alpha)(n−1))) = fully_up))
      => (EXISTS (m: pos_nat):
          (m > n & (EXISTS (r: train): pi(alpha)(m) = enterI(r))
              & fintime(t(alpha)(m)) <= fintime(t(alpha)(n)) + xi_1))));

safety: THEOREM (FORALL (alpha: atexecs): (FORALL (s: states):
    (in_atexec(alpha)(s) => ((EXISTS (r:train): status(r,s)=I) => gate_status(s)=fully_down))));

% utility_prop_a (alpha:atexecs, s:states): bool =
%   (EXISTS (s1:states):
%   (precedes(alpha)(s1,s) & (EXISTS (r:train): status(r,s1) = I) & now(s1) >= now(s) − xi_2));

utility_prop_a (alpha:atexecs, s:states): bool =
    (EXISTS (s1:states):
      (precedes(alpha)(s1,s) & (EXISTS (r:train): status(r,s1) = I) & now(s1) + xi_2 >= now(s)));

utility_prop_b (alpha:atexecs, s:states): bool =
    (EXISTS (s1:states):
      (precedes(alpha)(s,s1) & (EXISTS (r:train): status(r,s1) = I) & now(s1) <= now(s) + xi_1));

% utility_prop_c (alpha:atexecs, s:states): bool =
%   (EXISTS (s1,s2:states):
%   precedes(alpha)(s1,s) & precedes(alpha)(s,s2)
%   & (EXISTS (r:train):status(r,s1)=I) & (EXISTS (r:train):status(r,s2)=I)
%   & now(s2) − now(s1) <= xi_1 + xi_2 + delta);

utility_prop_c (alpha:atexecs, s:states): bool =
    (EXISTS (s1,s2:states):
        precedes(alpha)(s1,s) & precedes(alpha)(s,s2)
      & (EXISTS (r:train):status(r,s1)=I) & (EXISTS (r:train):status(r,s2)=I)
      & now(s2) <= xi_1 + xi_2 + delta + now(s1));

utility: THEOREM (FORALL (alpha: atexecs): (FORALL (s: states):
    ((in_atexec(alpha)(s) & NOT(gate_status(s) = fully_up)) =>
        (utility_prop_a(alpha,s) OR utility_prop_b(alpha,s) OR utility_prop_c(alpha,s))))));

END opspec_atexecs

opspec : THEORY

BEGIN

IMPORTING **opspec_atexecs**

END opspec

B.4 Appendix. The Timed Automaton *SystImpl* in PVS

The PVS specification of the full theory of *SystImpl* is structured analogously to that of *Trains*.

---

systimpl_decls: THEORY

BEGIN

  train: TYPE

  r,r1: VAR train

  IMPORTING **time_thy**

  beta_posreal: {r:real | r > 0};
  delta_t: VAR (fintime?)
  eps_1, eps_2, gamma_down, gamma_up, xi_1, xi_2, delta: (fintime?)
  beta:(fintime?) = fintime(beta_posreal:{r:real | r >= 0});

  const_facts: AXIOM
        (eps_1 <= eps_2
        & eps_1 > gamma_down
        & xi_1 + eps_1>= gamma_down + beta + eps_2
        & xi_2 >= gamma_up);

  actions : DATATYPE

    BEGIN
        nu(timeof:(fintime?)): nu?
        enterR(Rtrainof:train): enterR?
        enterI(Itrainof:train): enterI?
        exit(Etrainof:train): exit?
        lower: lower?
        raise: raise?
        up: up?
        down: down?
    END actions;

  a: VAR actions;

  train_status: TYPE = {not_here,P,I};

  gate_status: TYPE = {fully_up,fully_down,going_up,going_down};

  comp_train_status: TYPE = {comp_not_here,R};

  comp_gate_status: TYPE = {comp_up,comp_down};

  MMTstates: TYPE = [# trains_part: [train -> train_status],
                        gate_part: gate_status,
                        comp_train_status_part: [train --> comp_train_status],
                        comp_sched_time_part: [train -> time],
                        comp_gate_status_part: comp_gate_status #];

  IMPORTING **states**[actions,MMTstates,time,fintime?]

  s1: VAR states;

  status(r:train, s:states):train_status = trains_part(basic(s))(r);

  gate_status(s:states):gate_status = gate_part(basic(s));

  comp_status(r:train, s:states):comp_train_status = comp_train_status_part(basic(s))(r);

  sched_time(r:train, s:states):time = comp_sched_time_part(basic(s))(r);

  comp_gate_status(s:states):comp_gate_status = comp_gate_status_part(basic(s));

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)))
                    & now(s) + delta_t <= last(s)(up)
                    & now(s) + delta_t <= last(s)(down)
                    & (comp_gate_status(s) = comp_up =>
        enterR(r): status(r,s) = not_here,
        enterI(r): status(r,s) = P & first(s)(a) <= now(s),
        exit(r): status(r,s) = I,
                        (FORALL r: now(s) + delta_t + gamma_down < sched_time(r,s)))
                    & (comp_gate_status(s) = comp_down =>
                        (EXISTS r: sched_time(r,s) <=
                                now(s) + gamma_up + delta + gamma_down))),
        lower: comp_gate_status(s) = comp_up
            & (EXISTS r: sched_time(r,s) <= now(s) + gamma_down + beta),
        raise: comp_gate_status(s) = comp_down
            & (NOT (EXISTS r: sched_time(r,s) <= now(s)+gamma_up+delta+gamma_down)),
        up: gate_status(s) = going_up,
        down: gate_status(s) = going_down
    ENDCASES;

trans (a:actions, s:states):states =
    CASES a OF
        nu(delta_t): s WITH [now := now(s) + delta_t],
        enterR(r): s WITH [basic := basic(s) WITH
                                [trains_part := trains_part(basic(s)) WITH [r := P],
                                comp_train_status_part :=
                                    comp_train_status_part(basic(s)) WITH [r := R],
                                comp_sched_time_part :=
                                    comp_sched_time_part(basic(s)) WITH
                                        [r := now(s)+eps_1]],
                            first := first(s) WITH [(enterI(r)) := now(s)+eps_1],
                            last := last(s) WITH [(enterI(r)) := now(s)+eps_2]],
        enterI(r): s WITH [basic := basic(s) WITH
                                [trains_part := trains_part(basic(s)) WITH [r := I]],
                            first := first(s) WITH [(enterI(r)) := zero],
                            last := last(s) WITH [(enterI(r)) := infinity]],
        exit(r): s WITH [basic := basic(s) WITH
                            [trains_part := trains_part(basic(s)) WITH [r := not_here],
                            comp_train_status_part :=
                                comp_train_status_part(basic(s)) WITH [r := comp_not_here],
                            comp_sched_time_part :=
                                comp_sched_time_part(basic(s)) WITH [r := infinity]]],
        lower: IF gate_status (s) = fully_up OR gate_status(s) = going_up
                THEN s WITH [basic := basic(s) WITH
                                [gate_part := going_down,
                                comp_gate_status_part := comp_down],
                            last := last(s) WITH
                                [down := now(s) + gamma_down, up := infinity]]
            ELSE s ENDIF,

42

raise: IF gate_status(s) = fully_down OR gate_status(s) = going_down

THEN s WITH [basic := basic(s) WITH

[gate_part := going_up,

comp_gate_status_part := comp_up],

last := last(s) WITH

[up := now(s) + gamma_up, down := infinity]]

ELSE s ENDIF,

up: s WITH [basic := basic(s) WITH [gate_part := fully_up],

last := last(s) WITH [up := infinity]],

down: s WITH [basic := basic(s) WITH [gate_part := fully_down],

last := last(s) WITH [down := infinity]]

ENDCASES

enabled (a:actions, s:states):bool = enabled_general(a,s) & enabled_specific(a,s);

start (s:states):bool =

s = (# basic := (# trains_part := (LAMBDA r: not_here),

gate_part := fully_up,

comp_train_status_part := (LAMBDA r: comp_not_here),

comp_sched_time_part := (LAMBDA r: infinity),

comp_gate_status_part := comp_up #),

now := zero,

first := (LAMBDA a: zero),

last := (LAMBDA a: infinity) #)

IMPORTING machine[states, actions, enabled, trans, start]

END systimpl_decls

systimpl_unique_aux: THEORY

BEGIN

IMPORTING systimpl_decls

enterR_unique: LEMMA (FORALL (r1, r2: train): (enterR(r1) = enterR(r2) => r1 = r2));

enterI_unique: LEMMA (FORALL (r1, r2: train): (enterI(r1) = enterI(r2) => r1 = r2));

exit_unique: LEMMA (FORALL (r1, r2: train): (exit(r1) = exit(r2) => r1 = r2));

nu_unique: LEMMA (FORALL (t1, t2: (fintime?)): (nu(t1) = nu(t2) => t1 = t2));

END systimpl_unique_aux

systimpl_invariants: THEORY

BEGIN

IMPORTING systimpl_unique_aux

Inv_5_1_1(s: states):bool = (FORALL (r: train):

(comp_status(r,s) = R) IFF (status(r,s) = P OR status(r,s) = I));

lemma_5_1_1: LEMMA (FORALL (s: states): reachable(s) => Inv_5_1_1(s));

Inv_5_1_2(s: states):bool = (FORALL (r: train):

(status(r,s) = P) => (sched_time(r,s) = first(s)(enterI(r))));

lemma_5_1_2: LEMMA (FORALL (s: states): reachable(s) => Inv_5_1_2(s));

Inv_5_1_3(s: states):bool = (FORALL (r: train):

(comp_status(r,s) = R & sched_time(r,s) > now(s)) => (status(r,s) = P));

lemma_5_1_3: LEMMA (FORALL (s: states): reachable(s) => Inv_5_1_3(s));

Inv_5_1_4(s:states):bool = (FORALL (r:train): (status(r,s) = I) => (sched_time(r,s) <= now(s)));

lemma_5_1_4: LEMMA (FORALL (s: states): reachable(s) => Inv_5_1_4(s));

Inv_5_1_5(s: states):bool = (FORALL (r: train):
  (NOT (sched_time(r,s) = infinity)) => (status(r,s)=P OR status(r,s)=I));

lemma_5_1_5: LEMMA (FORALL (s: states): reachable(s) => Inv_5_1_5(s));

Inv_5_2_1(s: states):bool =
  (comp_gate_status(s) = comp_up
    IFF (gate_status(s) = fully_up OR gate_status(s) = going_up));

lemma_5_2_1: LEMMA (FORALL (s: states): reachable(s) => Inv_5_2_1(s));

Inv_5_2_2(s: states):bool =
  (comp_gate_status(s) = comp_down
    IFF (gate_status(s) = fully_down OR gate_status(s) = going_down));

lemma_5_2_2: LEMMA (FORALL (s: states): reachable(s) => Inv_5_2_2(s));

Inv_B_1_1(s: states):bool = (gate_status(s) = going_down => last(s)(down) >= now(s));

lemma_B_1_1: LEMMA (FORALL (s: states): reachable(s) => Inv_B_1_1(s));

Inv_B_1_2(s: states):bool = (gate_status(s) = going_down
  => last(s)(down) <= now(s) + gamma_down);

lemma_B_1_2: LEMMA (FORALL (s: states): reachable(s) => Inv_B_1_2(s));

Inv_6_1(s: states):bool = (FORALL (r: train):
  ((status(r,s) = P & (gate_status(s) = fully_up OR gate_status(s) = going_up))
    => first(s)(enterI(r)) > now(s) + gamma_down));

lemma_6_1: LEMMA (FORALL (s: states): reachable(s) => Inv_6_1(s));

Inv_6_2(s: states):bool = (FORALL (r: train):
  ((status(r,s) = P & gate_status(s) = going_down) => first(s)(enterI(r)) > last(s)(down)));

lemma_6_2: LEMMA (FORALL (s: states): reachable(s) => Inv_6_2(s));

Inv_6_3(s: states):bool = ((EXISTS (r: train): status(r,s) = I) => gate_status(s) = fully_down);

lemma_6_3: LEMMA (FORALL (s: states): reachable(s) => Inv_6_3(s));

Inv_6_3_A(s: states):bool = (FORALL (r: train): status(r,s) = I => gate_status(s) = fully_down);

lemma_6_3_A: LEMMA (FORALL (s: states): reachable(s) => Inv_6_3_A(s));

END systimpl_invariants

systimpl: THEORY
BEGIN
  IMPORTING systimpl_invariants
END systimpl

C Appendix. PVS Strategies for Timed Automata

Below are the definitions of the user-defined strategies that we have used in our proofs of properties of timed automata, organized by category. The strategies used in induction proofs have been rather finely tuned for efficiency on our example proofs.

Most of the strategies developed for ad hoc proofs are expected to form parts of higher-level strategies, once enhancements to PVS permits these strategies to be defined. Examples of their current and possible future use can be found in Appendix E, where we present an annotated ad hoc proof that relies on them.

The strategy file "pvs-strategies" presented below is divided into nine segments:

Segment 1: Major support strategies used in all induction strategies.

Segment 2: Specialized strategies for the timed automaton *Trains*.

Segment 3: Specialized strategies for the timed automaton *OpSpec*.

Segment 4: Specialized strategies for the timed automaton *SystImpl*.

Segment 5: Specialized simplification strategies for timed automata.

Segment 6: Apply-lemma strategies for timed automata.

Segment 7: General strategies useful in reasoning about **atexecs**.

Segment 8: A special tcc strategy.

Segment 9: Strategies for the timed_auto template version.

That many strategies rely on the form of the template is clear when one compares the strategies in Segment 9 to their counterparts in Segments 1, 3, and 7. As can be seen from examination of the strategies, both templates provide standard names for functions to be expanded as parts of strategies. The templates also guarantee the form of the definitions of these functions, and the form of critical lemmas such as "machine_induct" (the basis of the induction strategies). Knowledge of these forms is taken advantage of in various calls to SPLIT, LIFT-IF, FLATTEN, PROP, and INST. An obvious example is the definition of the strategy time_etc_simp, where explicit knowledge of the forms of the definitions in **time_thy** is used: every arithmetic operator or predicate is defined as an if_then_else, so, since these operators may occur embedded in expressions, expansion of each is followed by a LIFT-IF. A less obvious example is the prop_probe strategy in Segment 1, which is used by the induction strategies at the point where the invariant assertion in the inductive conclusion is instantiated with the state specified in some case of the case statement defining *trans*. When, as is often the case, this state is specified using an if_then_else construct, the call to LIFT-IF transforms the instantiated invariant assertion itself into an if_then_else construct, which can then be split by PROP into cases that may be provable by ASSERT.

The strategies presented divide into three categories: those specialized for proof of state invariants that are independent of timed executions, those designed to support recurring types of reasoning in ad hoc proofs about timed executions, and those that are useful in either context. The strategies in Segments 1 through 4 (and some of those in Segment 9) are in the first category. So far, the only strategies known to be in the last category are the simplification and apply-lemma strategies in Segments 5 and 6. The strategies in Segment 7 (and their relatives in Segment 9) are in the second category.

The strategies from Segment 7 generally depend on lemmas about admissible timed executions. The theory containing these lemmas is, in the conventions of our first template, the theory < *timed_automaton_name* >_strat_aux. Certain generic parts of this theory can be included as a sub-theory of **atexecs**, say, in a theory **atexecs_strat_aux**. However, other parts of this theory involve generic function names from parts of the template other than **atexecs**, and so must be included in a theory in which these function names are known. Using our first template, we are constrained to use some sub-theory of < *timed_automaton_name* > for this purpose; using our second template, we can use the generic theory **timed_auto_thy**. So far, we have only used these strategy support lemmas to reason about **opspec**;

hence, to simplify matters in using our first template, we have grouped all of the strategy support lemmas in **opspec_strat_aux** (see Appendix B.3).

The Segment 7 strategies provide automated support for handling several of the repeating patterns in proofs about admissible timed executions that were mentioned in Section 6. The strategy normalize_atexecs causes alternative representations of time and state values to be rewritten into a standard form, simplifying many proofs that two expressions represent the same value. Strategies using lemmas containing standard facts about reachability and states related by *trans* can currently be used to retrieve the information relevant to a certain state or pair of states given the name of the relevant *atexecs* value *alpha* and the index of a trajectory in *alpha* near the state or states in question. These strategies already help simplify the proof process. For example, in combination with normalize_atexecs, the reachability strategy allows one to simplify the process of confirming a particular state is reachable as a prerequisite to applying a state invariant lemma to the state. Provided sufficient tracking and recognition-by-content of assertions is added to PVS, these strategies could be refined to produce exactly the required information, with minimal user input and no extra clutter added to a sequent. In fact, it should be possible to use the reachability strategy invisibly to the user as part of an invariant-lemma strategy.

A final note on our strategies: many of them were developed to illustrate feasibility, and require further polishing. E.g., the use of standard names for skolem constants is a danger, since one must assure non-duplication of names associated with a given type. One advantage of standard names is that their significance is also standard. Ideally, automatic generation of these skolem constants will be accompanied by some method of tracking their significance. This might be done through an interface to PVS. There is another naming problem as well. When two or more automaton theories are being reasoned about simultaneously, the standard names of template operators will have more than one instantiation. For the strategies to work in such a situation, they will have to use the correct instantiations of these operators when they are invoked. The best way to obtain the information needed to determine these instantiations needs to be determined.

```
; ***                              Section 1                              ***
; ***              Major support strategies used in all induction strategies.     ***
; ***

          (defstep auto_cases (inv)
              (then* (lemma "machine_induct")
                     (expand "inductthm")
                     (inst −1 inv)
                     (split))
              ""  "Splitting into machine base and induction cases")

          (defstep base_case (inv)
              (then* (delete 2)
                     (expand "base")
                     (skolem 1 "s_1")
                     (flatten)
                     (expand "start")
                     (expand inv))
              ""   "Simplifying the machine base case")

          (defstep induct_cases (inv)
              (let ((x (format nil " ~a ~a ~a ~a ~a"
                       "(LAMBDA (a: actions): (FORALL (s: states): reachable(s) & "
                       inv
                       "(s) & enabled(a,s) => "
                       inv
                       "(trans(a,s))))")) )
              (then* (delete 2)
                     (expand "inductstep")
                     (lemma "actions_induction")
                     (inst −1 x)
```

46

```
                    (beta)
                    (branch (split)
                              ((then* (skolem 1 ("s_1" "a_1"))(inst -1 "a_1")(inst -1 "s_1"))
                               (skip)))))
          "" "Splitting the induction case on action class")

    (defstep reduce_case_one_var_exp (inv var1)
          (then* (delete 2)
                    (skolem 1 (var1))
                    (skolem 1 ("s_1"))
                    (flatten)
                    (expand "enabled")
                    (expand "trans")
                    (expand inv))
          "" "Applying the standard simplification")

    (defstep reduce_case_no_var_exp (inv)
          (then* (delete 2)
                    (skolem 1 ("s_1"))
                    (flatten)
                    (expand "enabled")
                    (expand "trans")
                    (expand inv))
          "" "Applying the standard simplification")
```

; The strategy reduce_case_no_var_rew is just like reduce_case_no_var_exp except that it calls rewrite
; instead of expand on trans in order to avoid doing the lift-if included in an expand that spoils the matching
; of universally quantified formulae in inductive hypothesis and conclusion. The choice of strategy is made
; according to whether there is an IF-THEN-ELSE at the top level of the definition for the corresponding
; case in the definition of trans. The use of rewrite on expand as well is the result of experiment showing
; it to be more efficient in this case.

```
    (defstep reduce_case_no_var_rew (inv)
          (then* (delete 2)
                    (skolem 1 ("s_1"))
                    (flatten)
                    (rewrite "enabled")
                    (rewrite "trans")
                    (expand inv))
          "" "Applying the standard simplification")
```

; The strategy prop_probe is used to test whether the remainder of a proof is "trivial". It is part of
; several other "_probe" strategies.

```
    (defstep prop_probe ()
          (then* (lift-if)
                    (prop)
                    (assert)
                    (fail))
          "" "")
```

; ***   Section 2   ***
; ***   Specialized strategies for the timed automaton trains.   ***

```
    (defstep auto_proof_trains (inv)
          (then (branch (auto_cases inv)
                    ((then (base_case inv)(trains_simp_probe)(postpone))
                     (branch (induct_cases inv)
```

47

```
((then (reduce_case_one_var_exp inv "t_1")
        (trains_simp_probe)(postpone))
 (then (reduce_case_one_var_exp inv "r_1")
        (trains_simp_probe)(postpone))
 (then (reduce_case_one_var_exp inv "r_1")
        (trains_simp_probe)(postpone))
 (then (reduce_case_one_var_exp inv "r_1")
        (trains_simp_probe)(postpone)))))))
"" "Taking care of the standard steps in the proof")

(defstep auto_proof_univ_trains (inv)
   (then (branch (auto_cases inv)
            (then (base_case inv)(trains_simp_probe)(postpone))
            (branch (induct_cases inv)
               ((then (reduce_case_one_var_exp inv "t_1")
                       (match_univ_and_trains_simp_probe)
                       (postpone))
                (then (reduce_case_one_var_exp inv "r_1")
                       (match_univ_and_trains_simp_probe)
                       (postpone))
                (then (reduce_case_one_var_exp inv "r_1")
                       (match_univ_and_trains_simp_probe)
                       (postpone))
                (then (reduce_case_one_var_exp inv "r_1")
                       (match_univ_and_trains_simp_probe)
                       (postpone))))))
   "" "Taking care of the standard steps in the proof")

(defstep trains_simp ()
   (then* (expand "OKstate?")
          (expand "status")
          (flatten))
   "" "Expanding some trains definitions")

(defstep trains_simp_probe ()
   (then (trains_simp) (prop_probe))
   "" "")

(defstep match_univ_and_trains_simp_probe ()
   (then (skolem 1 "r_2") (inst -2 "r_2") (trains_simp_probe))
   "" "")
```

```
; The auto_proof strategies are the induction strategies.

(defstep auto_proof_opspec (inv)
   (then (branch (auto_cases inv)
            (then (base_case inv)(opspec_simp_probe)(postpone))
            (branch (induct_cases inv)
               ((then (reduce_case_one_var_exp inv "t_1")
                       (opspec_simp_probe)(postpone))
                (then (reduce_case_one_var_exp inv "r_1")
                       (opspec_simp_probe)(postpone))
                (then (reduce_case_one_var_exp inv "r_1")
                       (opspec_simp_probe)(postpone))
```

```
                          (then (reduce_case_one_var_exp inv "r_1")
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_no_var_exp inv)
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_no_var_exp inv)
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_no_var_exp inv)
                                (opspec_simp_probe)(postpone))
                          (then (reduce_case_no_var_exp inv)
                                (opspec_simp_probe)(postpone))))))))
        "" "Taking care of the standard steps in the proof")

    (defstep opspec_simp ()
        (then* (expand "OKstate?")
               (expand "last_1")
               (expand "last_2_up")
               (expand "last_2_I")
               (expand "status")
               (expand "gate_status")
               (flatten))
        "" "Expanding some opspec definitions")

    (defstep opspec_simp_probe ()
        (then (opspec_simp) (prop_probe))
        "" "")
```

; The strategy do_trans_opspec expands and simplifies a state of opspec that is represented by a trans
; expression, and reasons about the result. Because trans is usually defined by a case expression, it is
; not unusual for branching to occur. It is typical for the expected number of branches to be small when
; do_trans_opspec is an appropriate proof step.

```
    (defstep do_trans_opspec ()
        (then (expand "trans")(opspec_simp)(lift-if)(assert)(assert))
        "" "")
```

; ***   Section 4   ***
; ***   Specialized strategies for the timed automaton systimpl.   ***

; The auto_proof strategies are the induction strategies.

```
    (defstep auto_proof_systimpl (inv)
        (then (branch (auto_cases inv)
                      (then (base_case inv)(systimpl_simp_probe)(postpone))
                      (branch (induct_cases inv)
                              ((then (reduce_case_one_var_exp inv "t_1")
                                     (systimpl_simp_probe)(postpone))
                               (then (reduce_case_one_var_exp inv "r_1")
                                     (systimpl_simp_probe)(postpone))
                               (then (reduce_case_one_var_exp inv "r_1")
                                     (systimpl_simp_probe)(postpone))
                               (then (reduce_case_one_var_exp inv "r_1")
                                     (systimpl_simp_probe)(postpone))
                               (then (reduce_case_no_var_exp inv)
                                     (systimpl_simp_probe)(postpone))
                               (then (reduce_case_no_var_exp inv)
                                     (systimpl_simp_probe)(postpone))
                               (then (reduce_case_no_var_exp inv)
```

```
                              (systimpl_simp_probe)(postpone))
                    (then (reduce_case_no_var_exp inv)
                              (systimpl_simp_probe)(postpone)))))))
          "" "Taking care of the standard steps in the induction proof")

(defstep auto_proof_univ_systimpl (inv)
       (then (branch (auto_cases inv)
                    (then (base_case inv)(systimpl_simp_probe)(postpone))
                    (branch (induct_cases inv)
                              ((then (reduce_case_one_var_exp inv "t_1")
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_one_var_exp inv "r_1")
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_one_var_exp inv "r_1")
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_one_var_exp inv "r_1")
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_no_var_rew inv)
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_no_var_rew inv)
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_no_var_exp inv)
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone))
                              (then (reduce_case_no_var_exp inv)
                                       (match_univ_and_systimpl_simp_probe)
                                       (postpone)))))))
          "" "Taking care of the standard steps in the induction proof")

(defstep systimpl_simp ()
       (then* (expand "OKstate?")
              (expand "beta")
              (expand "comp_status")
              (expand "comp_gate_status")
              (expand "sched_time")
              (expand "status")
              (expand "gate_status")
              (flatten))
          "" "Expanding some systimpl definitions")

(defstep systimpl_simp_probe ()
       (then (systimpl_simp) (prop_probe))
       "" "")

(defstep match_univ_and_systimpl_simp_probe ()
       (then (skolem 1 "r_2") (inst -2 "r_2") (systimpl_simp_probe))
       "" "")

(defstep direct_proof_univ_systimpl (inv)
       (then* (skolem 1 "s_1")
```

50

```
              (expand inv)
              (flatten)
              (skolem 1 "r_1")
              (systimpl_simp))
        "" "Doing the standard steps of a non-induction proof")
```

; Simplification strategies that handle time definitions and other simple types of reasoning needed for
; timed automata.

```
        (defstep time_etc_simp ()
            (then* (lift-if)
                   (prop)
                   (assert)
                   (expand "<=")
                   (lift-if)
                   (expand ">=")
                   (lift-if)
                   (expand "<")
                   (lift-if)
                   (expand ">")
                   (lift-if)
                   (expand "+")
                   (lift-if)
                   (expand "-")
                   (lift-if)
                   (repeat* (then* (assert) (prop) (lift-if))))
            "" "Doing time-arithmetic")
```

; The strategy time_etc_simp_probe tries time_etc_simp and backtracks if it does not succeed.

```
        (defstep time_etc_simp_probe ()
            (then* (lift-if)
                   (prop)
                   (assert)
                   (expand "<=")
                   (lift-if)
                   (expand ">=")
                   (lift-if)
                   (expand "<")
                   (lift-if)
                   (expand ">")
                   (lift-if)
                   (expand "+")
                   (lift-if)
                   (expand "-")
                   (lift-if)
                   (repeat* (then* (assert) (prop) (lift-if)))
                   (fail))
            "" "Doing time-arithmetic")
```

; The strategy time_simp focusses time_etc_simp on a single formula in a sequent.

```
        (defstep time_simp (fnum)
            (then* (lift-if fnum)
```

51

```
(my_prop fnum)
(assert)
(expand "<=" fnum)
(lift-if fnum)
(expand ">=" fnum)
(lift-if fnum)
(expand "<" fnum)
(lift-if fnum)
(expand ">" fnum)
(lift-if fnum)
(expand "+" fnum)
(lift-if fnum)
(expand "−" fnum)
(lift-if fnum)
(repeat* (then* (assert) (my_prop fnum) (lift-if fnum))))
"" "Doing time-arithmetic on a particular formula")
```

; The strategy my_prop focusses the standard strategy prop on a single formula in a sequent.

```
(defstep my_prop (fnum)
(try (flatten fnum) (my_prop fnum) (try (split fnum) (my_prop fnum)(skip)))
"" "")
```

; The following shorter version of time_etc_simp was provided by Shankar at SRI. It is equivalent in power
; to time_etc_simp, but testing has shown that while it is sometimes equally fast, it is sometimes several
; seconds slower.

```
(defstep time_etc_simp_shankar ()
(then (stop-rewrite)
(auto-rewrite-theory "time_thy")
(repeat* (then (lift-if)(ground))))
"" "Doing time-arithmetic")
```

; ***   Section 6   ***
; ***   Apply-lemma strategies for timed automata.   ***

; Some of the apply-lemma strategies are specialized for application of state invariant lemmas.

```
(defstep apply_lemma (lem args)
(let ((x (cons 'inst (cons −1 args))))
(then (lemma lem) x))
"" "Applying a lemma to some arguments")
```

```
(defstep apply_inv_lemma (invno &optional statevar)
(let ((lemma_name (format nil "~a ~a" "lemma_" invno))
(theorem_name (format nil "~a ~a" "lemma_" invno))
(inv_name (format nil "~a ~a" "Inv_" invno))
(state (cond (statevar) (t "s_1"))))
(then* (try (apply_lemma lemma_name (state)) (skip)
(apply_lemma theorem_name (state)))
(assert)
(expand inv_name)))
"" "Applying the appropriate invariant lemma")
```

```
(defstep apply_univ_inv_lemma (invno quantvar &optional statevar)
(let ((lemma_name (format nil "~a ~a" "lemma_" invno))
(theorem_name (format nil "~a ~a" "lemma_" invno))
(inv_name (format nil "~a ~a" "Inv_" invno))
```

52

```
                            (state (cond (statevar) (t "s_1")))))
            (then* (try (apply_lemma lemma_name (state)) (skip)
                        (apply_lemma theorem_name (state)))
                   (assert)
                   (expand inv_name)
                   (inst -1 quantvar)))
        "" "Applying the appropriate invariant lemma")
```

; ***   Section 7   ***
; ***   General strategies useful in reasoning about atexecs.   ***

; The strategy put_glb finds the time index of the last indexed time in an atexec that is less than or equal
; to the particular non-negative-real valued bound "timebound", and gives it an associated name.

```
        (defstep put_glb (atexec timebound)
            (let ((x (format nil " ~a ~a" timebound "_glb"))
                  (y timebound)
                  (z atexec))
                 (put_glb_2 x y z))
            "" "")

        (defstep put_glb_2 (boundname timebound atexec)
            (let ((x (list atexec timebound))
                  (y (list boundname)))
                 (then (apply_lemma "glb_fact" x) (skolem -1 y) (flatten)))
            "" "" )
```

; The strategy get_reachables adduces the fact of reachability for states in an atexec near time index "index",
; under various aliases.

```
        (defstep get_reachables (atexec index)
            (let ((x (list atexec index)))
                 (then (apply_lemma "reachable_states" x) (flatten)))
            "" "")
```

; The strategy trans_facts adduces the relatedness of states, under various aliases, via a transition in an
; atexec near time index "index".

```
        (defstep trans_facts (atexec index)
            (let ((x (list atexec index)))
                 (then (apply_lemma "trans_facts" x)
                       (flatten) (assert) (flatten)))
            "" "")
```

; The strategy normalize_atexecs converts all time points and state points of an admissible timed execution
; to a normal form, so that equalities may be inferred.

```
        (defstep normalize_atexecs ()
            (then (auto-rewrite-theory "opspec_strat_aux")
                  (apply (do-rewrite)))
            "" "")
```

; The strategy time_order is used to infer an inequality between time indices from the same inequality
; between the indexed times.

```
        (defstep time_order (atexec n1 n2)
            (let ((x (list atexec n1 n2)))
                 (then (apply_lemma "time_relation" x) (flatten) (simplify)))
            "" "")
```

; The strategy match_condition is used to simplify reasoning about an IF-THEN-ELSE assertion. It can
; sometimes circumvent splitting; when it does not, it can make the result of splitting more "natural".

```
(defstep match_condition (fnum)
    (then (split fnum) (flatten) (assert))
    "" "Attempting to eliminate a condition")
```

; The strategy modus_ponens is used to avoid splitting an assertion having a complex hypothesis identical
; to another assertion present.

```
(defstep modus_ponens (fnum)
    (branch (split fnum) ((skip) (assert)))
    "" "Attempting to eliminate an hypothesis")
```

; ***   Section 8   ***
; ***   A special tcc strategy.   ***

; A strategy useful in proving the tccs for the lemmas about admissible timed traces used to support
; normalize_atexecs:

```
(defstep same_states_tcc (atexec leftend rightend)
    (let ((timeseq (format nil " ~a ~a ~a" "t(" atexec ")"))
          (trajseq (format nil " ~a ~a ~a" "w(" atexec ")")))
    (then (skosimp)
          (expand "interval")
          (apply (then (typepred atexec) (hide -1 -3 -4) (inst-cp -1 leftend)
                       (inst -1 rightend)))
          (apply (then (typepred timeseq) (hide -1) (inst -1 leftend rightend)))
          (apply (then (typepred trajseq) (inst -1 leftend)))
          (expand "ltime")
          (assert)))
    "" "")
```

; ***   Section 9   ***
; ***   Strategies for the timed_auto template version.   ***

```
(defstep auto_proof_opspec_timed_auto (inv)
    (then (branch (time (auto_cases inv))
                  (then (base_case_timed_auto inv)(opspec_simp_probe)(postpone))
                  (branch (induct_cases inv)
                          ((then (reduce_case_timed_auto_one_var_exp inv "t_1")
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_one_var_exp inv "r_1")
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_one_var_exp inv "r_1")
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_one_var_exp inv "r_1")
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_no_var_exp inv)
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_no_var_exp inv)
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_no_var_exp inv)
                                 (opspec_simp_probe)(postpone))
                           (then (reduce_case_timed_auto_no_var_exp inv)
                                 (opspec_simp_probe)(postpone))))))
    "" "Taking care of the standard steps in the proof")
```

```
(defstep base_case_timed_auto (inv)
    (then* (delete 2)
            (expand "base")
            (skolem 1 "s_1")
            (flatten)
            (expand "start")
            (flatten)
            (expand "basic_start")
            (expand inv))
    "" "Simplifying the auto base case")

(defstep reduce_case_timed_auto_one_var_exp (inv var1)
    (then* (delete 2)
            (skolem 1 (var1))
            (skolem 1 ("s_1"))
            (flatten)
            (expand "enabled")
            (expand "trans")
            (expand "basic_trans")
            (expand inv))
    "" "Applying the standard simplification")

(defstep reduce_case_timed_auto_no_var_rew (inv)
    (then* (delete 2)
            (skolem 1 ("s_1"))
            (flatten)
            (rewrite "enabled")
            (rewrite "trans")
            (expand "basic_trans")
            (expand inv))
    "" "Applying the standard simplification")

(defstep reduce_case_timed_auto_no_var_exp (inv)
    (then* (delete 2)
            (skolem 1 ("s_1"))
            (flatten)
            (expand "enabled")
            (expand "trans")
            (expand "basic_trans")
            (expand inv))
    "" "Applying the standard simplification")

(defstep normalize_atexecs_timed_auto ()
    (then (auto-rewrite-theory
            "timed_auto_thy [ basic_states, actions, nu, nu?, timeof, basic_start,
                                first_start, last_start, basic_trans, first_trans,
                                basic_trans, enabled_specific, OKstate?]")
            (apply (do-rewrite)))
    "" "")

(defstep do_trans_opspec_timed_auto ()
    (then (expand "trans")
            (expand "basic_trans")(expand "first_trans")(expand "last_trans")
            (opspec_simp)(lift-if)(assert)(assert))
    "" "")
```

D Appendix. PVS Proofs of State Invariants

In this Appendix, we present our PVS proofs of all the state invariants that we have proved for our timed automata models. With the exceptions noted, these proofs all follow this standard script:

- If the proof is an induction proof, apply the appropriate induction strategy; otherwise, apply the appropriate direct-proof strategy.

- For each generated subgoal, introduce the facts and case splits appealed to in the hand proof. The facts appealed to may be the transition precondition (if the subgoal is an action case in the induction proof), axioms about constants in the automaton definition, applications of invariant lemmas, or applications of other lemmas.

- Attempt to complete the proof with an appeal to the strategy TIME_ETC_SIMP.

- In cases where this fails—often, these are cases dismissed as trivial in the hand proof—appeal to one of the following: the precondition, an appropriate uniqueness lemma, or facts about the constants associated with the timed automaton. Then again call TIME_ETC_SIMP.

Below is the proof of the single state invariant for the timed automaton *Trains*. It follows the standard script.

---

trains_invariants.lemma_3_1:

```
("" (AUTO_PROOF_UNIV_TRAINS "Inv_3_1")
 (("1" (APPLY (TIME_ETC_SIMP) "Case enterR(r_1)."))
  ("2" (APPLY (THEN (APPLY_LEMMA "enterI_unique" ("r_1" "r_2")) (TRAINS_SIMP))
               "Case enterI(r_1).")
       (TIME_ETC_SIMP))))
```

Below are the proofs of the seven state invariants for the timed automaton *OpSpec*. All of these proofs follow the standard script.

---

opspec_invariants.lemma_4_1_1:

```
("" (AUTO_PROOF_OPSPEC "Inv_4_1_1"))
```

opspec_invariants.lemma_4_1_2:

```
("" (AUTO_PROOF_OPSPEC "Inv_4_1_2")
 (("1" (APPLY (TIME_ETC_SIMP) "Base case." "Infinity plus finite equals infinity."))
  ("2" (APPLY (TIME_ETC_SIMP) "Case enterI(r_1)." "Infinity plus finite equals infinity."))
  ("3" (APPLY (TIME_ETC_SIMP) "Case up." "Infinity plus finite equals infinity."))))
```

opspec_invariants.lemma_4_2_1:

```
("" (AUTO_PROOF_OPSPEC "Inv_4_2_1")
 (("1" (APPLY (TIME_ETC_SIMP) "Base case." "zero <= infinity."))
  ("2" (APPLY (THEN (EXPAND "enabled_specific") (OPSPEC_SIMP)) "Case nu(t_1)."))
  ("3" (APPLY (TIME_ETC_SIMP) "Case enterI(r_1)." "finite <= infinity."))
  ("4" (APPLY (TIME_ETC_SIMP) "Case lower." "a <= a + b."))))
```

opspec_invariants.lemma_4_2_2:

```
("" (AUTO_PROOF_OPSPEC "Inv_4_2_2")
 (("1" (APPLY (TIME_ETC_SIMP) "Base case." "Finite <= infinity."))
```

("2" (APPLY (THEN (EXPAND "enabled_specific") (OPSPEC_SIMP)) "Case nu(t_1)."))
("3" (APPLY (TIME_ETC_SIMP) "Case enterI(r_1)." "Finite <= infinity."))
("4" (APPLY (TIME_ETC_SIMP) "Case exit." "a <= a + b + c + d."))
("5" (APPLY (TIME_ETC_SIMP) "Case up." "Finite <= infinity."))))

opspec_invariants.lemma_4_2_3:

("" (AUTO_PROOF_OPSPEC "Inv_4_2_3")
  (("1" (APPLY (TIME_ETC_SIMP) "Case nu(t_1)." "Finite <= infinity."))
   ("2" (APPLY (TIME_ETC_SIMP) "Case lower." "a <= a."))))

opspec_invariants.lemma_4_2_4:

("" (AUTO_PROOF_OPSPEC "Inv_4_2_4")
  (("1" (APPLY (TIME_ETC_SIMP) "Case nu(t_1)." "Finite <= infinity."))
   ("2" (APPLY (TIME_ETC_SIMP) "Case exit(r_1)." "a <= a."))))

opspec_invariants.lemma_4_2_5:

("" (AUTO_PROOF_OPSPEC "Inv_4_2_5")
  (("1" (APPLY (TIME_ETC_SIMP) "Case nu(t_1)." "a <= b + c implies a <= b + d + c."))
   ("2" (APPLY (TIME_ETC_SIMP) "Case exit(r_1)." "a <= a."))))

Below are the proofs of the thirteen invariant lemmas of the timed automaton *SystImpl*. The ones whose proofs differ from the standard script are lemma_6_1 and lemma_6_3.

The major differences in lemma_6_3, which is logically equivalent to lemma_6_3_A, result from the fact that it is formulated with an embedded existential quantifier rather than a top-level universal quantifier, making it difficult to predict how to match skolemization and instantiation in the induction steps. The differences in lemma_6_1 consist of uses of MODUS_PONENS, ASSERT, and INST. MODUS_PONENS is used to eliminate from some implication-assertions their hypotheses that would have been eliminated by the calls to ASSERT in the apply-invariant-lemma strategies if they had been simpler in form. The call to ASSERT then eliminates another hypothesis that was the conclusion of one of the original implication-assertions; combining it with INST accomplishes the required instantiation of part of the precondition of the action nu(t_1).

With appropriate enhancements to PVS, these deviations from the standard script can be eliminated, except possibly for the "appropriate instantiation of the precondition"; whether this step can be automated as part of a general apply-the-precondition strategy remains to be determined.

---

systimpl_invariants.lemma_5_1_1:

("" (AUTO_PROOF_UNIV_SYSTIMPL "Inv_5_1_1")
  (("1" (APPLY (TIME_ETC_SIMP) "Base case."
              "Retrieving function defs from state s_1 and doing beta reduction."))
   ("2" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP)) "Case enterI(r_1).")
        (TIME_ETC_SIMP))))

systimpl_invariants.lemma_5_1_2:

("" (AUTO_PROOF_UNIV_SYSTIMPL "Inv_5_1_2")
  (("1" (APPLY (APPLY_LEMMA "enterI_unique" ("r_1" "r_2")) "Case enterR(r_1).")
        (TIME_ETC_SIMP))
   ("2" (APPLY (APPLY_LEMMA "enterI_unique" ("r_1" "r_2")) "Case enterI(r_1).")
        (TIME_ETC_SIMP))))

systimpl_invariants.lemma_5_1_3:

    ("" (AUTO_PROOF_UNIV_SYSTIMPL "Inv_5_1_3")
       (("1" (APPLY (TIME_ETC_SIMP) "Case nu(t_1)." "a > b + c implies a > b."))
        ("2" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP)) "Case enterI(r_1).")
           (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_2" "r_1") (SYSTIMPL_SIMP)))
           (APPLY (TIME_ETC_SIMP) "Doing obvious case-based reasoning."))))

systimpl_invariants.lemma_5_1_4:

    ("" (DIRECT_PROOF_UNIV_SYSTIMPL "Inv_5_1_4")
      (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_1" "r_1") (SYSTIMPL_SIMP)))
      (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_3" "r_1") (SYSTIMPL_SIMP)))
      (TIME_ETC_SIMP))

systimpl_invariants.lemma_5_1_5:

    ("" (AUTO_PROOF_UNIV_SYSTIMPL "Inv_5_1_5"))

systimpl_invariants.lemma_5_2_1:

    ("" (AUTO_PROOF_SYSTIMPL "Inv_5_2_1")
       (("1" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP)) "Case up.")
         (TIME_ETC_SIMP))
        ("2" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP)) "Case down.")
         (TIME_ETC_SIMP))))

systimpl_invariants.lemma_5_2_2:

    ("" (DIRECT_PROOF_UNIV_SYSTIMPL "Inv_5_2_2")
      (APPLY (THEN (APPLY_INV_LEMMA "5_2_1") (SYSTIMPL_SIMP)))
      (TIME_ETC_SIMP))

systimpl_invariants.lemma_B_1_1:

    ("" (AUTO_PROOF_SYSTIMPL "Inv_B_1_1")
       (("1" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP)) "Case nu(t_1).")
         (APPLY (HIDE −5 −8 −9) "Hiding quantified formulae before using (time_etc_simp).")
         (APPLY (TIME_ETC_SIMP) "Doing time-arithmetic: reversing an inequality."))
       ("2" (APPLY (TIME_ETC_SIMP) "Case lower."
               "Doing propositional reasoning plus time arithmetic."))))

systimpl_invariants.lemma_B_1_2:

    ("" (AUTO_PROOF_SYSTIMPL "Inv_B_1_2")
       (("1" (APPLY (TIME_ETC_SIMP) "Case nu(t_1)."))
       ("2" (APPLY (TIME_ETC_SIMP) "Case lower."))))

systimpl_invariants.lemma_6_1:

    ("" (AUTO_PROOF_UNIV_SYSTIMPL "Inv_6_1")
       (("1" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP))
           "Case nu(t_1). Invoke the precondition.")
        (APPLY (THEN (APPLY_INV_LEMMA "5_2_1") (SYSTIMPL_SIMP)))

```
              (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_2" "r_2") (SYSTIMPL_SIMP)))
              (MODUS_PONENS -3)
              (MODUS_PONENS -5)
              (ASSERT)
              (INST -11 "r_2")
              (APPLY (HIDE -8) "Hiding quantified formulas.")
              (TIME_ETC_SIMP))
        ("2" (APPLY (THEN (APPLY_LEMMA "const_facts" NIL)
                          (APPLY_LEMMA "enterI_unique" ("r_1" "r_2")))
                    "Case enterR(r_1). Appeal to some standard facts.")
              (APPLY (TIME_ETC_SIMP) "Combine some propositional and time-arithmetic reasoning."))
        ("3" (APPLY (APPLY_LEMMA "enterI_unique" ("r_1" "r_2"))
                    "Case enterI(r_1). Appeal to some standard facts.")
              (TIME_ETC_SIMP))
        ("4" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP))
                    "Case raise. Invoke the precondition.")
              (INST 1 "r_2")
              (APPLY (THEN (APPLY_UNIV_INV_LEMMA "5_1_2" "r_2") (SYSTIMPL_SIMP)))
              (TIME_ETC_SIMP))
        ("5" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP))
                    "Case up. Invoke the precondition.")
              (APPLY (TIME_ETC_SIMP) "Applying simple propositional reasoning."))))
```

systimpl_invariants.lemma_6_2:

```
  ("" (AUTO_PROOF_UNIV_SYSTIMPL "Inv_6_2")
    (("1" (APPLY (THEN (APPLY_INV_LEMMA "B_1_2") (SYSTIMPL_SIMP)) "Case enterR(r_1).")
          (APPLY (APPLY_LEMMA "const_facts" NIL) "Appealing to some standard facts.")
          (APPLY (TIME_ETC_SIMP) "Combining reasoning about cases and time arithmetic."))
      ("2" (APPLY (APPLY_LEMMA "enterI_unique" ("r_1" "r_2")) "Case enterI(r_1)."
                  "Appealing to a standard fact.")
          (APPLY (TIME_ETC_SIMP) "Doing some propositional reasoning."))
      ("3" (APPLY (THEN (APPLY_UNIV_INV_LEMMA "6_1" "r_2") (SYSTIMPL_SIMP))
                  "Case lower." "Apply invariant lemma 6_1.")
          (APPLY (TIME_ETC_SIMP) "Doing propositional reasoning by cases."))))
```

systimpl_invariants.lemma_6_3:

```
  ("" (APPLY (AUTO_PROOF_SYSTIMPL "Inv_6_3") "Use induction.")
    (("1" (APPLY (TIME_ETC_SIMP) "Case enterR(r_1)."
                  "Prepare sequent for matched SKOLEM and INST.")
          (APPLY (THEN (SKOLEM -4 "r_2") (INST 1 "r_2")) "Setting r = r_2.")
          (APPLY (TIME_ETC_SIMP) "Confirm that this case is trivial."))
      ("2" (APPLY (TIME_ETC_SIMP) "Case enterI(r_1)."
                  "Prepare sequent for matched SKOLEM and INST.")
          (APPLY (THEN (SKOLEM -4 "r_2") (INST 1 "r_2")) "Setting r = r_2.")
          (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP))
                  "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 the invariant lemma B_1, part 1.")
                (APPLY (TIME_ETC_SIMP) "Derive contradiction with the precondition."))))
    ("3" (APPLY (TIME_ETC_SIMP) "Case exit(r_1)."
                "Prepare sequent for matched SKOLEM and INST.")
          (APPLY (THEN (SKOLEM −4 "r_2") (INST 1 "r_2")) "Setting r = r_2.")
          (APPLY (TIME_ETC_SIMP) "Confirm that this case is trivial."))
    ("4" (APPLY (TIME_ETC_SIMP) "Case raise.")
          (APPLY (THEN (SKOLEM −5 "r_2"))
                "Matching formula for INST was eliminated. Setting r = r_2.")
          (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP) (INST 1 "r_2"))
                "Invoke and specialize the pre-condition.")
          (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."))
    ("5" (APPLY (THEN (EXPAND "enabled_specific") (SYSTIMPL_SIMP))
                "Case up. Invoke the precondition.")
          (APPLY (TIME_ETC_SIMP) "Derive contradiction with the precondition."))))

systimpl_invariants.lemma_6_3_A:

    ("" (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 1 "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.")
          (APPLY (TIME_ETC_SIMP) "Derive contradiction with the precondition."))))

E Appendix. Lessons from the PVS Proof of Lemma E.1

In the first section of this Appendix, we present the statement of Lemma E.1 from Appendix B.3 of [12], and its proof in PVS, annotated by an equivalent interleaved English language proof. The PVS proof was developed as follows: First, an attempt was made to follow the hand proof in [12] as closely as possible. Many missing details needed for the PVS prover were incorporated in this proof, resulting in a PVS proof running to over 6 pages, not counting comments. Strategies were designed to abbreviate many of the repeated sequences in reasoning; these sequences were identified by the purpose they served, and were therefore not always precisely syntactically isomorphic. The proof was then redone using the new strategies to replace longer sequences. At this stage, the structure of the PVS proof became much clearer, and it was possible to eliminate duplicated efforts and simply wasteful steps. Comments were incorporated with the proof steps to help clarify this structure. The resulting PVS proof was then annotated in fuller detail in English.

Note that this is not the method by which we expect other ad hoc timed automaton proofs in PVS to be developed. One result of our study of such proofs should be the identification of parts of English language proofs that need additional detail to be added when translated for the automatic prover, as well as the type of additional detail they need. With this knowledge, a hand proof can be expanded to a more detailed one more suitable for straightforward translation. Used in combination with the more sophisticated large step strategies whose development will become possible when appropriate enhancements are added to PVS, this approach can bring us closer to the ideal of mimicking a natural hand proof in PVS. The addition to PVS of a facility for attaching comments to goals rather than proof steps will also aid in the documentation of the correspondence between natural language proof and PVS proof.

Included among our annotations of the Lemma E.1 proof are indications of step sequences that have the potential of being replaced by calls to large step strategies supported by PVS enhancements. These sequences are denoted by asterisks along the right margin, with number tags in the form (m.n), where m denotes the m-th potential strategy, and n denotes the n-th place where this strategy could be used. The second section of this Appendix describes the desired effects and possible implementations of these strategies, and shows for each potential use just how the strategy might be invoked.

E.1 The PVS Proof of Lemma E.1 with Annotations

```
; Lemma_E_1: Let alpha be an admissible timed execution of OpSpec.    Let Pi be any lower event
; occurring in alpha from a state in which Gate.status is in {going-up, up}.    Then there is an enterI
; event Phi occurring after Pi in alpha, with time(Phi) <= time(Pi) + xi_1.
("”

; Let A_1 be an atexec of OpSpec and n_1 be the index of the action Pi = pi(A_1)(n_1), which occurs
; from the state lstate(w(A_1)(n_1 − 1)) in which the gate is either going_up or fully_up.
(SKOLEM 1 "A_1")
(SKOLEM 1 "n_1")
(FLATTEN)

; For convenience, we will call the state just before Pi, namely lstate(w(A_1)(n_1 − 1)), s_1, and the
; state after Pi, namely fstate(w(A_1)(n_1)), s_2.
(APPLY (THEN (NAME "s_1" "lstate(w(A_1)(n_1 − 1))")
            (NAME "s_2" "fstate(w(A_1)(n_1))"))
       "Give names s_1 and s_2 to the states just before and after Pi.")

; Supposition 1: The time of Pi, t(A_1)(n_1), equals now(s_1) and now(s_2).
(APPLY (CASE "now(s_2) = now(s_1) & Now(s_1) = t(A_1)(n_1)")
       "Assume the time equivalences from the first sentence of the hand
        proof.    The proof of these facts below depends mainly on
        normalize_atexecs.")
(("1"
```

; Supposition 2: last_1(s_2) <= now(s_2) + xi_1.

    (APPLY (CASE "last_1(s_2) <= now(s_2) + xi_1")
               "Assume the inequality that concludes the first paragraph of the
               hand proof.   It will be proved later below.")
  (("1"
    (TIME_SIMP -1)
    (FLATTEN)

; Call the value last_1(s_2), b.

    (NAME "b" "dur(last_1(s_2))")
    (APPLY (REPLACE -1) "Using the name b."
           "Note that it would be helpful to have a general strategy
           replace_names that could periodically be invoked to call
           replace on all definitions entered via NAME, and perhaps
           certain others. Marked definitions could be kept hidden
           until recalled for this purpose or for a general review
           of definitions.")

; Call the value $t(A_1)(n_1) + dur(xi_1)$, B.

    (NAME "B" "t(A_1)(n_1) + dur(xi_1)")

; Let B_glb name the largest index for which $t(A_1)(B\_glb) <= B$.

    (PUT_GLB "A_1" "B")

; Supposition 3: B_glb + 1 > n_1.

    (CASE "B_glb + 1 > n_1")
    (("1"

; Note that part of the hypothesis of lemma_E_1 is that the gate is either going_up or fully_up in state s_1.

      (NORMALIZE_ATEXECS)
      (REPLACE -11)

; Supposition A(B): We suppose something that appears, a priori, stronger than the negation of what
; we hope to prove, in order to derive a contradiction: namely, that there is no enterI event from event
; Pi to time B, *and* that the last_1 component of the state after every event from Pi to B has the
; same value last_1(s_2). Actually, the second part is redundant, but is needed in the induction proof
; that comes later.

; We actually state this supposition in a more complex form. This more complex form of Supposition
; A(B) says that every event from the n_1-th event Pi to time B has property A, where property A of
; any m-th event is that every p-th event from event Pi through that m-th event has property A-hat:
; namely, it is not an enterI event and the last_1 component of its result state equals last_1(s_2).

    (APPLY (CASE "(forall (m:pos_nat): (m >= n_1 & t(A_1)(m) <= B) =>
                 (forall (p:pos_nat): ((n_1 <= p & p <= m) =>
                 (not (exists (r: train): pi(A_1)(p) = enterI(r)) &
                 last_1(fstate(w(A_1)(p))) = last_1(s_2)))))")
        "Call this assertion A(B). We hope it is not true. Thus, we
        expect to obtain a contradiction by assuming it. It says,
        in effect, that every event before and up to time bound B
        is the last in a chain of events starting with Pi that have
        the same last_1 value when they occur, and that are not an
        enterI.")
    (("1"

; In particular, event B_glb has property A.

      (INST -1 "B_glb")   *
      (("1"   * (1.1)
        (ASSERT)   *

; So, event B_glb has property A-hat.
```
                    (APPLY (INST −1 "B_glb")                                    *
                            "Starting to show that last_1 still has the value it did    *
                            at s_2 when the B_glb+1-th event occurs.   This uses the     *
                            fact that time has advanced beyond B, in a hidden way—       * (1.2)
                            via lemma last_1_interval.   It will lead to a contradiction.")   *
                    (ASSERT)                                                     *
                    (FLATTEN)                                                    *
```
; Let s_3 name the state just after the B_glb-th event, and s_4 name the state just before the
; (B_glb + 1)-th event.
```
                    (APPLY (THEN (NAME "s_3" "fstate(w(A_1)(B_glb))")
                                 (NAME "s_4" "lstate(w(A_1)(B_glb))"))
                            "Let s_3 and s_4 be the left and right endpoints of the
                            B_glb-th interval, which spans the time B.")
```
; A standard lemma says that components of the state other than the "now" component are not
; affected by time passage.   In particular, the value of last_1 is the same at s_4 as it was at s_3.
```
                    (APPLY_LEMMA "last_1_interval" ("A_1" "B_glb"))              *
                    (NORMALIZE_ATEXECS)                                          * (2.1)
                    (APPLY (THEN (REPLACE −2) (REPLACE −3))                      *
                            "Using the names s_3 and s_4.")                      *
```

; Supposition 4: Now(s_4) = t(A_1)(B_glb+1).
```
                    (CASE "Now(s_4) = t(A_1)(B_glb+1)")
                    (("1"
```
; Since s_4 is reachable, we can apply the invariant lemma 4_2_1, which says that
; now(s_4) <= last_1(s_4).
```
                    (GET_REACHABLES "A_1" "B_glb")                               *
                    (APPLY_INV_LEMMA "4_2_1" "s_4")                              * (3.1)
                    (APPLY (HIDE −2 −3 −4 −5 −6 −7)                              *
                            "Remove unneeded reachability info.")                *
```
; But this contradicts what we know about now(s_4): namely, that Now(s_4) = t(A_1)(B_glb+1) > B,
; last_1(s_4) = last_1(s_3), last_1(s_3) = last_1(s_2) (because event B_glb has property A-hat),
; and last_1(s_2) <= B.
```
                    (EXPAND "Now")
                    (TIME_SIMP −1))
```
; Proof of Supposition 4: Follows from the definition of s_4 and standard simplifications.
```
                    ("2"
                    (APPLY (REPLACE −2 1 RL)
                            "Using the def of s_4 in the subgoal consequent.")
                    (NORMALIZE_ATEXECS)
                    (SIMPLIFY))))
```
; We prove a TCC showing that B_glb is within the range of values for which property A-hat holds,
; given that B_glb has property A.
```
                    ("2" (ASSERT))))
```
; We now prove Supposition A(B) by induction on the variable m.
```
                    ("2"
                    (APPLY (THEN (INDUCT "m") (ASSERT))
                            "Note that inducting this way avoids creating 4 subgoals."
```

"This is the proof of assertion A(B); rather, it is the proof
that if assertion A(B) is false, then one can find an enterI
event between Pi and B, which claim is assertion A(B)'s
companion assertion in the subgoal's consequent.")

; The base case and some TCC's were proved by ASSERT, i.e., by the decision procedures,
; simplification, and propositional reasoning in PVS.   We prove the induction step.

(APPLY (THEN (SKOLEM 1 "j_1") (FLATTEN)))

; Suppose that j_1 is some integer such that if j_1 > 0 and the j_1-th event is the n_1-th event or later
; and comes before time B, then j_1 has property A.   We will show that if the conclusion of lemma_E_1
; is false, then if the (j_1 + 1)-th event is the n_1-th event or later and comes before time B, then
; j_1 + 1 has property A.   To do this, suppose that indeed the (j_1 + 1)-th event is the n_1-th event
; or later and comes before time B.

; If j_1 + 1 = n_1, it trivially has property A, since pi(A_1)(n_1) = lower and
; last_1(path(w(A_1)(n_1))(t(A_1)(n_1))) = last_1(s_2).

(CASE "j_1+1 = n_1")
(("1" (SKOLEM 1 "p_1") (FLATTEN) (ASSERT))

; So we may assume that j_1 + 1 is greater than n_1.

("2"

; We may then deduce that j_1 > 0 (since n_1 is positive), and that j_1 >= n_1.

(ASSERT)

; We may also deduce that the j_1-th event comes before time B, because the (j_1 + 1)-th does.

(APPLY (TIME_ORDER "A_1" "j_1" "j_1+1")   *
        "This should mimic doing a SPLIT without getting   *
         a companion assertion.")   * (4.1)
(ASSERT)   *
(APPLY (HIDE −1) "Finish the mimicking of SPLIT.")   *

; Let p_1 be some positive natural number.   We will show that if n_1 <= p_1 <= j_1 + 1, then p_1
; has property A-hat.   Since p_1 is arbitrary, it will follow that j_1 + 1 has property A.

(SKOLEM 2 "p_1")

; By inductive hypothesis, if n_1 <= p_1 <= j_1, then p_1 has property A-hat.

(INST −1 "p_1")

; So, suppose that n_1 <= p_1 <= j_1 + 1.

(APPLY (FLATTEN)
        "Note, above is an instance of matched skolem + inst.")

; Then if p_1 is in fact less than or equal to j_1, it has property A-hat.

(ASSERT)

; So, consider the case that p_1 = j_1 + 1.

(APPLY (CASE "p_1 = j_1 + 1")
        "This is the meaty case, since if p_1 < j_1 + 1, then the
         induction hypothesis, which applies to all p_1 between
         n_1 and j_1, inclusive, assures that last_1 at p_1 is
         the same as at n_1, and that event p_1 is not enterI.")

; Then we will show that either lemma_E_1 holds with the p_1-th event being the event Phi that comes
; after Pi and before B that is an enterI, or else p_1 has property A-hat.

```
               (("1"
                (INST 3 "p_1")
                (ASSERT)
```

; Since the p_1-th event is after Pi, this is equivalent to showing that (1) either the p_1-th event is an
; enterI or p_1 has property A-hat and (2) either the p_1-th event happens before B or p_1 has property
; A-hat.

```
               (SPLIT 3)
```

; We first prove (1).

```
               (("1"
```

; We recall that the induction hypothesis says that any integer between n_1 and j_1 has property A-hat.

```
                (APPLY (REVEAL −1)
                       "Grab the general form of the induction hypothesis.")
```

; This applies in particular to j_1 itself.

```
                (APPLY (INST −1 "j_1")                               *
                       "We now prepare to do the real induction step in this   *
                       proof, namely, to show that last_1 is unchanged by       *
                       event j_1 + 1 provided it is not an enterI.")            *
                (ASSERT)                                              *
                                                                      * (1.3)
```

; We will show that either the p_1-th event is an enterI or else last_1 in the state   *
; following this event is last_1(s_2).   *
                                                                                               *
```
                (APPLY (SPLIT 3)                                      *
                       "Using a SPLIT to match a complex expression that   *
                       ASSERT missed.")                               *
```

; We first replace our formulation of the assertion that the p_1-th event is an enterI (given as an
; existentially quantified expression over trains r that might do an enterI action) by the data-type
; recognizer formulation of this assertion: enterI?(pi(A_1)(p_1)).

```
                (APPLY (CASE "enterI?(pi(A_1)(p_1))")                *
                       "Expressing enterI−ness better.")             *
                (("1"                                                * (G)
                 (INST 2 "Itrainof(pi(A_1)(p_1))")                   *
                 (APPLY−EXTENSIONALITY 2))                           *
                ("2"                                                 *
```

; We can now proceed with our proof.   Since j_1 has property A-hat, the value of last_1 in the state
; following the j_1-th event is last_1(s_2).

```
               (FLATTEN)
```

; For convenience, we assign names to certain states: s_8 is the state just after the j_1-th event, s_9 is
; the state just before the (j_1 + 1)-th event, and s_10 is the state just after the (j_1 + 1)-th event.

```
               (APPLY (THEN (NAME "s_8" "fstate(w(A_1)(j_1))")
                            (NAME "s_9" "lstate(w(A_1)(j_1))")
                            (NAME "s_10" "fstate(w(A_1)(j_1+1))"))
                      "Name the states relevant to the preservation of
                      last_1 from the time of event j_1 to that of event
                      j_1 + 1.")
```

; We note that the value of last_1 does not change in passing from s_8 to s_9.

```
               (APPLY_LEMMA "last_1_interval" ("A_1" "j_1"))
```

; We also note that s_10 is the state reached by transitioning on the (j_1 + 1)-th action in state s_9.

(TRANS_FACTS "A_1" "j_1")   *
                    (APPLY (THEN (REPLACE −8) (REPLACE −9))   *
                         "Note that trans_facts has invoked rewriting, and   *
                          thus accomplished a normalize_atexecs.   We now use * (5.1)
                          the names s_9 and s_10, which will also help   *
                          to find which of the trans_facts are not needed.")   *
                    (APPLY (HIDE −2 −3 −4 −5 −6)   *
                         "Hide the irrelevant trans facts.")   *

; Since p_1 = j_1 + 1, we know that what we have to prove is that either the (j_1 + 1)-th event is an
; enterI or else last_1(s_10) = last_1(s_2).

                    (APPLY (THEN (REPLACE −7) (REPLACE −3))
                         "Using the equality of p_1 to j_1 + 1 and
                          the definition of s_10 to rephrase part of
                          the consequent to last_1(s_10) = last_1(s_2).")

; We now show that since s_10 is the result of transitioning on the (j_1 + 1)-th action in state s_9, the
; desired conclusion follows.

                    (APPLY (THEN (REPLACE −1 + RL) (HIDE −1))   *
                         "Using the trans version of s_10 in the consequent.")   * (6.1)
                    (DO_TRANS_OPSPEC))))   *

; We prove (2): either the p_1-th event happens before B or p_1 has property A-hat, by showing that
; the p_1-th event indeed happens before B (because we have supposed that the (j_1 + 1)-th event
; does, and we have assumed that p_1 = j_1 + 1).

                    ("2" (TIME_SIMP 1))))

; We now consider the case p_1 < j_1 + 1; but the fact that the p_1-th event has property A-hat for
; any such p_1, which is what we must prove, follows trivially from the inductive hypothesis.

                    ("2" (ASSERT))))))))

; We now prove Supposition 3: B_glb + 1 > n_1.   This follows from the fact that the (B_glb + 1)-th
; event happens after the n_1-th event.

              ("2" (TIME_ORDER "A_1" "n_1" "B_glb + 1") (ASSERT))))   * (4.2)

; We now prove Supposition 2: last_1(s_2) <= now(s_2) + xi_1.

              ("2"
               (HIDE 2)

; We first isolate the fact that now(s_1) = now(s_2).

                (FLATTEN)

; We adduce the fact that state s_2 is the result of transitioning on the n_1-th action Pi on state s_1.

                (TRANS_FACTS "A_1" "n_1−1")   *
                (NORMALIZE_ATEXECS)   *
                (APPLY (THEN (REPLACE −7) (REPLACE −8)) "Using the names s_1 and s_2.") * (5.2)
                (APPLY (HIDE −2 −3 −4)   *
                     "The object here is to hide all instances of trans that   *
                      are not connected to s_2 or are redundant.")   *

; Applying these two facts, together with the knowledge of the effect of the lower action Pi in a state
; in which the gate is either going_up or fully_up, we see that it is enough to show that either
; last_1(s_1) = infinity or else last_1(s_1) <= now(s_1) + xi_1.

                (APPLY (REPLACE −2)
                     "Using the fact that now(s_2) = now(s_1); note that it is
                      critical to do this before the next step.")
                (APPLY (THEN (REPLACE −1 + RL) (HIDE −1))   *
                     "Using the trans version of s_2 in the consequent.")   * (6.2)
                (DO_TRANS_OPSPEC)   *

66

```
        (APPLY (MATCH_CONDITION 1)
                    "Forcing the if-then-else with condition fully_up or going_up
                    to simplify as if it were recognized that the condition is
                    equivalent to going_up or fully_up.")
        (APPLY (MATCH_CONDITION 1)
                    "This splits and flattens the new if-then-else, giving the
                    effect of case splitting on last_1(s_1) = infinity.")
; In the case last_1(s_1) = infinity, the result is trivial.
        (("1" (TIME_SIMP 1))
; So, we may assume that last_1(s_1) <> infinity.
        ("2"
; Now, state s_1 is reachable.
            (APPLY (GET_REACHABLES "A_1" "n_1 - 1")                    *
                    "Note that this is one case where one wants to keep both      *
                    assertions in the consequent.")                              * (3.2)
; Therefore, we can apply the invariant lemma 4_2_3 to s_1:                      *
            (APPLY_INV_LEMMA "4_2_3" "s_1")                                       *
; which is precisely what is required.
            (APPLY (THEN (OPSPEC_SIMP_2) (TIME_SIMP 2)))))))))
; Proof of Supposition 1: The time of Pi, t(A_1)(n_1), equals now(s_1) and now(s_2).   This follows
; from standard equivalences, the definitions of s_1 and s_2, and standard PVS simplifications and
; decision procedures.
        ("2"
        (APPLY (HIDE 2) "Finally proving the time equivalence facts.")
        (APPLY (THEN (REPLACE −1 + RL) (REPLACE −2 + RL))
                    "Expanding the names s_1 and s_2 in the consequent.")
        (NORMALIZE_ATEXECS)
        (SIMPLIFY))))
```

E.2   Potential New Strategies for Lemma E.1 from PVS Enhancements

In the annotated proof in Appendix E.1, we have indicated places where six new strategies could be applied, if we had the tools in PVS to define them. Here, we describe the effects (when successful) and possible implementation of each of these proposed new strategies, and indicate how they would be invoked at each of the indicated places in the Lemma E.1 proof. As will be seen, all the proposed implementations require naming, analysis, and recognition capabilities for assertions that are not currently available in PVS. To avoid repetition, the proposed implementations omit mentioning the anticipated use of assertion naming by strategies that rely on a lemma; such naming will aid in the recognition and removal of redundant or irrelevant information generated from the lemma application. The proposed implementations also ignore an important feature that will need to be incorporated in each: the generation of useful error messages.

(1) (CONCLUDE <proposition> <instantiation>)

Effect. Puts all conclusions in <proposition>, applied to <instantiation>, in the hypotheses of the current sequent. <proposition> may be denoted by a tag (this requires enhancement to PVS) or by an assertion number, and should refer to a current subgoal antecedent formula in the form of a universally quantified implication whose hypothesis is satisfied by ¡instantiation¿.

Proposed implementation. Implement by a sequence INST, MODUS_PONENS, FLATTEN focused on <proposition>, which may be indicated either by a name rather than an assertion number. MODUS_PONENS will be a more sophisticated version of MODUS_PONENS as defined in Appendix C;

it will focus on removing the highest-level hypothesis from an assertion, provided it can be deduced from other antecedent formulae in a sequent.

Invocations in the Lemma E.1 proof.

(1.1) (CONCLUDE "Claim_A(B)" "B_glb")

(1.2) (CONCLUDE "Claim_A(B)_conclusion" "B_glb")

(1.3) (CONCLUDE "Claim_A(B)_ind_hyp" "j_1")

Comments. We have used assertion names rather than assertion numbers in these example invocations. It is anticipated that the assertion name "Claim_A(B)" would be provided by the user at the time this particular assertion was introduced using CASE. Names of the related assertions would be automatically generated according to the structure of the original assertion (as in "Claim_A(B)_conclusion") or as the result of its manipulation by other strategies (such as the call to "(INDUCT "m")", that locates and then operates on assertion "Claim_A(B)").

If one uses ASSERT instead of MODUS_PONENS, the invocation (1.1) will handle an associated TCC automatically. The proposed implementation of CONCLUDE probably will be modified to maximize the set of cases in which this will happen.

(2) (SAMEVAL <component_name> <state_1> <state_2>)

Effect. Adds the fact that the state function or component <component_name> has the same value at <state_1> and <state_2> to the hypotheses of the current goal.

Proposed implementation. The implementation must rely on the existence of a lemma about <component_name> with a standard derived name that guarantees that <component_name> is constant in a trajectory. State arguments that are simply names may have to be looked up in the visible or hidden part of the antecedent hypothesis list to determine the appropriate relevant *atexecs* and *nat* instantiations for the lemma, and to replace the states in the conclusion of the lemma by their names.

Invocations in the Lemma E.1 proof.

(2.1) (SAMEVAL "last_1" "s_3" "s_4")

Comments. Automatic generation and proof of the necessary supporting lemmas is a possibility.

(3) (INVARIANT <inv_name> <state>)

Effect. Adds the fact that invariant <inv_name> holds for <state> to the hypotheses of the current goal.

Proposed implementation. Use GET_REACHABLES to retrieve information about the reachability of states in the neighborhood of <state>. This neighborhood can be deduced from the arguments of types *atexecs* and *nat* in the representation of <state>, which, in turn, will either be explicitly present or retrievable from some visible or hidden equality in the current goal. This information can be retrieved, used in the call to APPLY_INV_LEMMA, and then hidden.

Invocations in the Lemma E.1 proof.

(3.1) (INVARIANT "4_2_1" "s_4")

(3.2) (INVARIANT "4_2_3" "s_1")

Comments. The retrieval of reachability information about <state> could include a check on whether this information is present in either the visible or hidden part of the current goal, if this increases efficiency.

(4) (TIME_RELATION <index_1> <index_2>)

Effect. Puts the fact that event <index_1> comes before event <index_2>, or vice-versa (whichever is correct) in the hypotheses of the current goal.

Proposed implementation. Use the strategy TIME_ORDER, followed by ASSERT (to deduce and apply the appropriate inequality between <index_1> and <index_2>); then hide or delete the extra assertion generated (that relates to the "inappropriate" inequality).

Invocations in the Lemma E.1 proof.
(4.1) (TIME_RELATION "j_1" "j_1 + 1")
(4.2) (TIME_RELATION "n_1" "B_glb + 1")

Comments. An argument of type *atexecs* may also be added to this strategy. The alternative is to provide some means to retrieve the appropriate instantiation or instantiations from the current goal. When reasoning about properties of one timed automaton, there will typically be only one such instantiation. When reasoning about simulations between timed automata, there may be two.

(5) (TRANS_RELATION <state_1> <state_2>)

Effect. Puts the fact that <state_2> is the result of a transition from <state_1>, or vice-versa, in the antecedent, with an instantiation of the action associated with the transition.

Proposed implementation. The neighborhood of <state_1> and <state_2> (that is, the relevant *atexecs* and *nat* values) are retrieved either directly or by looking state names up in the visible or hidden part of the antecedent of the current goal. The strategy TRANS_FACTS can then be invoked to get all likely candidates for the transition relation. State values are then normalized by NORMALIZE_ATEXECS, and those equal to <state_1> and <state_2> are replaced by <state_1> and <state_2>. Irrelevant or redundant assertions generated by TRANS_FACTS are then removed; these are recognized by a combination of name and content.

Invocations in the Lemma E.1 proof.
(5.1) (TRANS_RELATION "s_9" "s_10")
(5.2) (TRANS_RELATION "s_1" "s_2")

Comments. If <state_1> and <state_2> are expressions rather than names, replacing equal state values by these names will require some care, since these expressions may be altered by NORMALIZE_ATEXECS.

(6) (COMPUTE_TRANS <state> <assertion>)

Effect. Replaces <state> in <assertion> by its value computed as the result of a transition.

Proposed implementation. If the representation of <state> as the result of at transition is present in the antecedent of the current goal, it can be recognized, and used to replace <state> in <assertion>. A version of DO_TRANS that focuses only on <assertion> can then be applied.

Invocations in the Lemma E.1 proof.
(6.1) (COMPUTE_TRANS "s_10" "Claim_A(B)_ind_concl")
(6.2) (COMPUTE_TRANS "s_2" "Supposition_2")

Comments. DO_TRANS is timed-automaton-dependent, since it calls the standard simplification strategy of the timed automaton in which the transition takes place. However, the name of the timed automaton, and hence that of its simplification strategy, could be deduced from the type information for <state>.

Note that we have marked one sequence in the proof of Lemma E.1 with an "(G)". At point "(G)", the representation of a certain fact in the sequent is changed by supplying a rather cryptic instantiation and applying the PVS strategy APPLY-EXTENSIONALITY. Both representations of the particular fact correspond to the same high-level English language description "the p_1-th event is an enterI event". Thus, sequence "(G)" does not exactly correspond to any step in an English language proof of Lemma E.1; the need for it in the PVS proof is really an artifact of the representation in PVS of the timed automaton OpSpec. "(G)" may be one example of a point where a certain amount of "PVS glue" is required in translating from hand proof to PVS.

F Appendix. A Second PVS Template for Timed Automata

The theory **timed_auto_decls**, if used as one of the fixed underlying template theories, is designed to allow the PVS typechecker to enforce many of the template conventions, such as the existence of a time passage action, the usage of the separate parts of *enabled*, the fact that the *now* component of a start state must be zero, and so on. It has an accompanying theory, **timed_auto_thy**, which we do not show since it is essentially identical to the theory **opspec_strat_aux** (see Appendix B.3).

One of the benefits of including the two theories among the fixed template theories is that the lemmas in **timed_auto_thy** become independent of the automaton being specified, and can be pre-proved prior to specifying any particular timed automaton. They do not then have to be re-proved in order to be used (by way of our specialized strategies or otherwise) in constructing "guaranteed sound" PVS proofs of properties of a particular timed automaton.

We first present the theory **timed_auto_decls**, and then show how the timed automaton *OpSpec* would be defined in PVS using the resulting new template.

F.1 Appendix. The Theory timed_auto_decls

```
timed_auto_decls [ basic_states, actions: TYPE,
%                       Importing time_thy defines the type "time" that behaves like the non-negative
%                       reals except for having an infinite value included.
                    (IMPORTING time_thy,
%                       Importing states[...] defines the type states, whose elements are records with
%                       indices "basic" (basic_states),"now" (a (fintime?) value), "first" and "last"
%                       (maps from actions to time).
                    states[actions,basic_states,time,fintime?])
                    nu: [(fintime?) -> actions],
                    nu?: [actions -> bool],
                    timeof: [(nu?) -> (fintime?)],
%                       The "start" predicate on states is split into three parts to emphasize its
%                       structure and to enforce "now(s) = zero".
                    basic_start: [basic_states -> bool],
                    first_start: [basic_states,actions -> time],
                    last_start: [basic_states,actions -> time],
%                       The "trans" operation of actions on states is also split into three parts to
%                       emphasize its independence from "now" except in the special case of a
%                       time-step action nu.
                    basic_trans: [[actions,states] -> basic_states],
                    first_trans: [[actions,states] -> [actions->time]],
                    last_trans: [[actions,states] -> [actions->time]],
                    enabled_specific: [[actions,states] -> bool],
                    OKstate?: [states -> bool] ] : THEORY
```

% The theory timed_auto_decls is the main template specification for timed automata. Instantiation
% of this template is done by importing the companion specification timed_auto_thy with the
% appropriate parameters.

% The expected instantiations of the specification parameters to timed_auto_decls and timed_auto_thy
% are as follows:

%   basic_states: some encoding of that part of the states that is separate from the "now"
%   component (a time value) and the "first" and "last" components (maps from actions
%   (events) to time).

%   actions: an abstract data type whose members are actions, that contains a "nu" action
%   parameterized by (non-zero, non-infinite) time.

%   nu: the time-step element of "actions", parameterized by "(fintime?)"

%   nu?: a predicate on actions that identifies just when an action is a "nu" time-step action.

%   timeof: extracts the "time" parameter from time-step actions.

%   basic_start: the predicate that identifies the basic parts of start states.

%   first_start: the function that maps states and actions to the initial "first" value for that
%   action with respect to the basic part of the state.

%   last_start: the function that maps states and actions to the initial 'last" value for that
%   action with respect to the basic part of the state.

%   basic_trans: this is the part of "trans" that does not deal with changes to "now", "first" and
%   "last".

%   first_trans: this is the part of "trans" that describes how one action affects the "first" time of
%   another.

%   last_trans: this is the part of "trans" that describes how one action affects the "last" time of
%   another.

%   enabled_specific: this is the part of enabled that maps an action to the non-default part of the
%   pre-condition predicate on the state.

%   OKstate?: this is a predicate on states that can be used to enforce one or more state invariants
%   by restricting the reachable states directly.

  BEGIN

%   Before importing atexecs, one needs to define start, Now, step?, and nu; step? depends on the
%   definitions of enabled and trans, so must define these also. Note that one can then go ahead and
%   import machine as well.

start(s:states):bool = (s = (# basic := basic(s),
                            now := zero,
                            first := (LAMBDA(a:actions): first_start(basic(s),a)),
                            last := (LAMBDA(a:actions): last_start(basic(s),a)) #)
                & basic_start(basic(s)));

Now(s:states):{r:real | r >= 0} = dur(now(s));

Nu (z: {z:real | z>0}): actions = nu(fintime(z: {z:real | z>=0}));

trans(a:actions,s:states):states =
  IF nu?(a) THEN s WITH [now := now(s) + timeof(a)]
    ELSE s WITH [basic := basic_trans(a,s),
               first := (LAMBDA(b:actions):first_trans(b,s)(a)),
               last := (LAMBDA(b:actions):last_trans(b,s)(a))]
    ENDIF;

enabled_general(a:actions,s:states):bool =
  IF nu?(a) THEN dur(timeof(a)) > 0 ELSE first(s)(a) <= now(s) & now(s) <= last(s)(a) ENDIF;

enabled(a:actions,s:states):bool =
  enabled_general(a,s) & enabled_specific(a,s) & OKstate?(trans(a,s));

step? (s1:states, a:actions, s2:states): bool = enabled(a,s1) & s2 = trans(a,s1);

IMPORTING atexecs [states, actions, start, Now, step?, Nu]
IMPORTING machine[states, actions, enabled, trans, start]

  END timed_auto_decls

F.2 Appendix. The Timed Automaton *OpSpec* in PVS: Version 2

The specification of the theory **opspec** in the alternative template (and actually, the new template itself) is rather more messy, than the specification in B.3, since certain functions have been decomposed into several functions. However, at least some of this messiness could be hidden by an appropriate interface external to PVS.

```
opspec_decls: THEORY
  BEGIN
    train: TYPE

    r,r1: VAR train

    IMPORTING time_thy

    beta_posreal: {r:real | r > 0};
    delta_t: VAR (fintime?)
    eps_1, eps_2, gamma_down, gamma_up, xi_1, xi_2, delta: (fintime?)
    beta:(fintime?) = fintime(beta_posreal:{r:real | r >= 0});

    const_facts: AXIOM ( eps_1 <= eps_2
                        & eps_1 > gamma_down
                        & xi_1 >= gamma_down + beta + eps_2 − eps_1
                        & xi_2 >= gamma_up );

    actions : DATATYPE
      BEGIN
            nu(timeof:(fintime?)): nu?
            enterR(Rtrainof:train): enterR?
            enterI(Itrainof:train): enterI?
            exit(Etrainof:train): exit?
            lower: lower?
            raise: raise?
            up: up?
            down: down?
      END actions;

    a: VAR actions;

    train_status: TYPE = {not_here,P,I};

    gate_status: TYPE = {fully_up,fully_down,going_up,going_down};

    basic_states: TYPE = [# trains_part: [train −> train_status],
                             gate_part: gate_status,
                             last_1_part, last_2_up_part, last_2_I_part: time #];

    IMPORTING states[actions,basic_states,time,fintime?]
    s1: VAR states;
    b: VAR basic_states;

    status(r:train, s:states):train_status = trains_part(basic(s))(r);

    gate_status(s:states):gate_status = gate_part(basic(s));

    last_1(s:states):time = last_1_part(basic(s));

    last_2_up(s:states):time = last_2_up_part(basic(s));

    last_2_I(s:states):time = last_2_I_part(basic(s));

    OKstate? (s:states): bool = ((EXISTS (r:train): status(r,s) = I) => gate_status(s) = fully_down);
```

enabled_specific (a:actions, s:states):bool =
    CASES a OF
        enterR(r): status(r,s) = not_here,
        enterI(r): status(r,s) = P & first(s)(a) <= now(s),
        exit(r): status(r,s) = I,
        nu(delta_t): (delta_t > zero
               & (FORALL r: now(s) + delta_t <= last(s)(enterI(r)))
               & now(s) + delta_t <= last(s)(up)
               & now(s) + delta_t <= last(s)(down)
               & now(s) + delta_t <= last_1(s)
               & now(s) + delta_t <= last_2_I(s)),
      lower: true,
      raise: true,
      up: gate_status(s) = going_up,
      down: gate_status(s) = going_down
    ENDCASES;

basic_trans (a:actions, s:states):basic_states =
    CASES a OF
        enterR(r): basic(s) WITH [trains_part := trains_part(basic(s)) WITH [r := P]],
        enterI(r): basic(s) WITH
               [trains_part := trains_part(basic(s)) WITH [r := I],
               last_1_part := infinity,
               last_2_up_part := infinity,
               last_2_I_part := infinity],
      exit(r): LET b = basic(s) WITH [trains_part:= trains_part(basic(s)) WITH [r:= not_here]]
          IN IF (FORALL (r1: train): (NOT (r1 = r)) => (NOT status(r1,s) = I))
             THEN b WITH
                 [last_2_up_part := now(s) + xi_2,
                  last_2_I_part := now(s) + xi_2 + delta + xi_1]
             ELSE b ENDIF,
      nu(delta_t): basic(s),
      lower: IF gate_status (s) = fully_up OR gate_status(s) = going_up
          THEN LET b = basic(s) WITH [gate_part := going_down]
             IN IF last_1(s) = infinity
               THEN b WITH [last_1_part:= now(s)+xi_1]
               ELSE b ENDIF
          ELSE basic(s) ENDIF,
      raise: IF gate_status(s) = fully_down OR gate_status(s) = going_down
          THEN basic(s) WITH [gate_part := going_up]
          ELSE basic(s) ENDIF,
      up: LET b = basic(s) WITH [gate_part := fully_up]
        IN IF now(s) <= last_2_up(s)
          THEN b WITH [last_2_up_part:= infinity, last_2_I_part:= infinity]
          ELSE b ENDIF,
      down: basic(s) WITH [gate_part := fully_down]
    ENDCASES

first_trans (a:actions, s:states):[actions->time] =
    CASES a OF
      enterR(r): first(s) WITH [(enterI(r)) := now(s) + eps_1],
      enterI(r): first(s) WITH [(enterI(r)) := zero]
    ELSE first(s)
    ENDCASES

last_trans (a:actions, s:states):[actions—>time] =
    CASES a OF
        enterR(r): last(s) WITH [(enterI(r)) := now(s)+eps_2],
        enterI(r): last(s) WITH [(enterI(r)) := infinity],
        exit(r): last(s),
        nu(delta_t): last(s),
        lower: IF gate_status (s) = fully_up OR gate_status(s) = going_up
                THEN last(s) WITH [down := now(s)+gamma_down, up := infinity]
                ELSE last(s) ENDIF,
        raise: IF gate_status(s) = fully_down OR gate_status(s) = going_down
                THEN last(s) WITH [up := now(s)+gamma_up, down := infinity]
                ELSE last(s) ENDIF,
        up: last(s) WITH [up := infinity],
        down: last(s) WITH [down := infinity]
    ENDCASES

basic_start (b:basic_states):bool =
    b = (# trains_part := (LAMBDA r: not_here),
            gate_part := fully_up,
            last_1_part := infinity,
            last_2_up_part := infinity,
            last_2_I_part := infinity #);

first_start (b:basic_states, a:actions):time = zero;

last_start (b:basic_states, a:actions):time = infinity;

IMPORTING timed_auto_thy [basic_states, actions, nu, nu?, timeof,
                            basic_start, first_start, last_start,
                            basic_trans, first_trans, last_trans,
                            enabled_specific, OKstate?]

END opspec_decls

74