This paper has been published in Annals of Mathematics and Artificial Intelligence,
Vol. 29 (2000), No. 1–4, pp. 139-181.

TAME: Using PVS Strategies for Special-Purpose Theorem Proving *

Myla Archer

Code 5546, Naval Research Laboratory, Washington, DC 20375
E-mail: archer@itd.nrl.navy.mil

TAME (Timed Automata Modeling Environment), an interface to the theorem proving system PVS, is designed for proving properties of three classes of automata: I/O automata, Lynch-Vaandrager timed automata, and SCR automata. TAME provides templates for specifying these automata, a set of auxiliary theories, and a set of specialized PVS strategies that rely on these theories and on the structure of automata specifications using the templates. Use of the TAME strategies simplifies the process of proving automaton properties, particularly state and transition invariants. TAME provides two types of strategies: strategies for "automatic" proof and strategies designed to implement "natural" proof steps, i.e., proof steps that mimic the high-level steps in typical natural language proofs. TAME's "natural" proof steps can be used both to mechanically check hand proofs in a straightforward way and to create proof scripts that can be understood without executing them in the PVS proof checker. Several new PVS features can be used to obtain better control and efficiency in user-defined strategies such as those used in TAME. This paper describes the TAME strategies, their use, and how their implementation exploits the structure of specifications and various PVS features. It also describes several features, currently unsupported in PVS, that would either allow additional "natural" proof steps in TAME or allow existing TAME proof steps to be improved. Lessons learned from TAME relevant to the development of similar specialized interfaces to PVS or other theorem provers are discussed.

Keywords: theorem proving, strategies, PVS

AMS Subject classification: 68T15, 68N30, 68Q60, 18B20

1. Introduction

Developers in industry often view model checking as a more practical formal method than theorem proving for establishing system properties. But although model checking is an important technique for developing correct systems, it does not solve all problems associated with verification. For example, while model checking is often regarded as automatic and therefore requiring less expertise from the user, the user must typically model check an abstraction of a given

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

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2001		**00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
TAME: Usisng PVS Strategies for Special-Purpose Theorem Proving	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5546,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**45**	

system rather than the full system. Not only is abstraction essential in model checking software systems with infinite state spaces, but in addition, most practical finite state systems have such large state spaces that abstraction is still necessary. Finding the appropriate abstraction often requires user ingenuity and creativity. Even when abstraction is used, state explosion can prevent a model checker from running to completion, and thus from establishing the correctness of a property. Moreover, in specifications involving parameters, model checking alone can verify correctness only for specific (usually small), rather than arbitrary, values of the parameters. Thus, model checkers are often better for *debugging* than for verification. To *verify* system properties, theorem proving is usually needed.

However, most system developers view theorem proving as impractical. The first difficult problem is to specify the system being developed in the language of the theorem prover. The second is to master the use of the theorem prover itself, whose proof steps are often a poor match to the reasoning steps humans naturally use in establishing the truth of a proposition. Third, even for sophisticated users, applying a mechanical theorem prover can require a prohibitive amount of time and effort. What is therefore needed is a natural theorem-proving language that will simplify theorem proving just as higher-level programming languages (from FORTRAN and other early examples on) simplified the programming process by permitting programmers to think at the algorithmic, rather than the machine language, level.

TAME (Timed Automata Modeling Environment) [4,6,5,8,3] is an interface to the theorem proving system PVS [48] that provides a high-level language for proving properties of several classes of automata used to represent systems: I/O automata [39], Lynch-Vaandrager (LV) timed automata [40], and SCR automata—the automata model underlying the SCR (Software Cost Reduction) method for specifying software requirements [2,20,25,22,21]. A major goal of TAME is to allow users to specify and to prove properties of such automata without exceptional effort, and thus allow mechanical theorem proving to become a practical part of the software development process. To achieve this goal, TAME provides a set of specification templates, a set of standard theories, and a set of specialized proof steps that allow users to create proofs using "natural" or automatic proof steps, without learning the details of the PVS proof steps. The templates provide users with standard ways of defining the parts of an automaton—its state space, initial states, and transitions—and standard formats for invariant properties. The special-purpose nature of the proofs, together with the uniform specification structure ensured by the templates, makes the implementation of both the natural and automatic proof steps in TAME possible.

A second important goal of TAME is to provide better user feedback than that provided by PVS. This feedback is provided by the TAME strategies in several ways. For SCR automata, a special feedback strategy is provided. The TAME strategies for natural proof steps provide improved feedback when a proof

fails because they are designed to create a saved proof that can be understood without executing it in the PVS proof checker. A saved proof created with the TAME strategies has a recognizable structure, both because the names of the strategies correspond to the natural proof steps they implement, and because the strategies generate comments that indicate the significance of the branches in a proof. Such information makes it easy to determine the significance of the places in a saved proof where the proof is incomplete. In addition, when a TAME proof succeeds, it is easy to determine from the saved proof which facts were used in the proof, and thus to determine whether a property holds for a more trivial reason than expected. Unexpectedly trivial proofs are often the result of a specification error. In addition to feedback from saved proofs, TAME improves the understandability of the subgoals of proofs in progress by means of labels that indicate the origins of individual formulae.

TAME is based on a general-purpose higher-order logic theorem prover, rather than built from scratch, for several reasons. First, there was no reason to rebuild specification, typechecking, and basic proof support when existing systems already provide this support in well-worked-out forms. Basing TAME on an existing general-purpose system allows its proof steps to be implemented based on proof steps and theorems proved in the existing system. If the proof steps of the existing system are sound, this will guarantee that the TAME proof steps are also sound. Very powerful proof steps can be implemented in this way when the existing system supports higher-order logic. Finally, desired additions to the proof support TAME provides can be implemented with a small effort and without low-level programming or soundness justifications. PVS was chosen as the basis for TAME because it uses standard logic (most likely to be understandable by developers), has a relatively user-friendly interface, saves rerunnable proof scripts, and includes decision procedures that handle many low-level proof details.

TAME is not the first interface to PVS designed to support special-purpose theorem proving. Earlier interfaces include a Duration Calculus Proof Assistant [49], an interface for the TRIO logic [1], and an interface to support proofs of invariant properties of DisCo specifications [33]. All of these interfaces provide proofs steps, based on PVS strategies, intended to implement inference rules in other *logics*. Unlike the strategies in these other interfaces, the TAME strategies are not designed to support reasoning in another logic. Instead, as noted above, they are designed to support both natural human-style reasoning and some automatic reasoning about automata models.

While not originally designed to support special-purpose interfaces, PVS has always had features that support the creation of new high-level proof steps—a strategy language and support for term rewriting, forward chaining, and generic theories and other higher-order features. In addition, new PVS features extend the possible capabilities of user-defined strategies. Several new features have played an important role in the refinement of TAME's strategies. The development of the TAME strategies has also helped to identify a few other features,

currently missing in PVS, that would be helpful in supporting more high-level human-style steps.

The TAME strategies have been used successfully in proving properties of many specifications, including the Generalized Railroad Crossing problem [23,4], the Steam Boiler Controller [35,6], a Group Communication Service [18], the RPC-Memory problem [47,46,7,44], part of the IEEE 1394 bus protocol [15,14, 7,44], and the requirements specification for a U.S. Navy communications device [28,29]. In addition to its developers, TAME has had two other users. Reference [7] describes the positive experience of a new TAME user (with no previous experience with PVS), who was able to check all the proofs of invariants from [46] and [14] in about four weeks. The feedback [13] from a user who used TAME in verifying a secure group membership protocol was also positive, indicating that TAME significantly simplified mechanizing the proofs of invariants.

This paper is organized as follows. Section 2, which describes TAME's proof support, first discusses the types of automaton properties for which TAME provides proof support and then gives an overview of the major TAME strategies and their use. Section 3 describes the automata models for which TAME provides proof support, how these automata are specified in TAME, and how properties of the specifications are used to advantage by the TAME strategies. It also discusses the relationship between SCR and TAME, and how other tools in the SCR* toolset help with both specifying SCR automata and proving their properties in TAME. Section 4 provides examples of proofs constructed with the TAME strategies and of the feedback TAME provides. Section 5, which describes the implementation of the TAME strategies, first discusses strategy construction in PVS, and then describes the features of PVS that are particularly useful in creating natural proof steps and in designing proof steps for efficiency. Finally, it illustrates how these features have been used in various TAME strategies, and describes how the efficiency of the automatic proof steps has been improved significantly since their initial implementation. Section 6 discusses features desirable in a theorem prover to support high-level proof steps like TAME's, and the availability of these features in PVS and in other theorem provers. Finally, Section 7 discusses further related work, and Section 8 presents some conclusions.

2. TAME Proof Support

As noted above, TAME's major goals are to simplify specifying automata models in PVS, to simplify proving properties of these models, and to provide the user with meaningful feedback about the proofs. The three automata models currently supported in TAME—LV timed automata, I/O automata, and SCR automata—possess similarities which TAME's current specification and proof support exploit. In particular, in each model, the states of an automaton are determined by assignments to state variables, the initial states are defined by a state predicate, and transitions are described by preconditions and postcondi-

tions on state variable values. Associated with all the models are the notions of *execution sequence* (a sequence of states connected by transitions), *reachable state* (a state reachable from an initial state by an execution sequence), and *reachable transition* (a prestate/poststate pair in which the prestate is reachable).

Section 2.1 describes the classes of properties of automata for which TAME provides proof support, and Section 2.2 gives an overview of the major TAME strategies.

2.1. Properties of Automata Supported

Properties which one may want to prove about automata include properties of a single automaton, such as invariants and properties of timed executions, and properties of pairs of automata, such as simulation, refinement, or implementation. Invariant properties include *state invariants* (properties of all reachable states), defined by predicates on the variables of a single state, and *transition invariants* (properties of all reachable transitions), defined by predicates on the variables of the two states in a transition. TAME provides extensive support for proofs of invariant properties and a few strategies to support proofs of timed execution properties. Currently, TAME does not support proofs of properties of pairs of automata.

Timed execution properties generally express facts about the relative timing of events. For example, the utility property in the Generalized Railroad Crossing Problem in [23,24], a timed execution property, states (in a precise way): "If the gate is down in the crossing, then either a train is present, a train has recently left, or a train is about to enter." Although such properties can be regarded as invariants of a timed system, their proofs are not as straightforward as proofs of state or transition invariants. The proofs do not have a uniform structure, and formulating and proving the properties requires additional definitions and proof steps based on a theory of timed executions. Designing more complete TAME style support for proving timed execution properties is a topic for further research. However, the initial investigation of appropriate strategies for these properties led to the desire for at least one additional capability in PVS—the ability to do resolution-style reasoning (see Section 6).

In contrast to proofs of timed execution properties, proofs of state invariants and of simulation (of which refinement and implementation are special cases) usually are based on induction, and thus have a standard structure, with a base case involving initial states and a case for each possible action (or "external" action, for simulation). This makes them especially good targets for mechanization. Direct (non-induction) proofs are also possible for these types of properties; such proofs are usually quite simple, and appeal to previously proved invariant or simulation properties. Transition invariants are seldom appropriate candidates for proof by induction, since there is usually little relation between the transitions from a given state and those from one of its successor states. These properties

are usually proved from the definition of the possible transitions. TAME supports this type of proof along with induction and direct proofs of state invariants. Extending TAME's proof support to cover proofs of simulation and its variants requires a feature currently missing in PVS, but planned for a future release [36], namely, theory instantiations and theory parameters to theories. This feature will permit generic formulations of simulation theorems between automata that allow separately proved state invariants of the automata to be applied in their proofs.

2.2. Overview of the TAME Strategies

TAME provides proof steps that free the user from handling PVS proof details that are trivial, tedious, or obscure. These proof steps are implemented as strategies defined in the PVS strategy language (see Section 5.2). TAME strategies fall into two classes: 1) those for proving properties of an automaton of any of the classes supported in TAME, 2) those designed specifically for SCR automata. The first class, the *general* strategies, allow users to mechanize a hand proof in PVS interactively, or to construct a proof interactively that can later be understood analogously to a hand proof—i.e., without running the proof in PVS. These strategies support proof steps that mimic the high-level steps in hand proofs that appear in the literature. The second class of strategies, the *SCR-specific* strategies, are partly based on the general strategies. To the extent feasible, the SCR-specific strategies efficiently automate proofs of invariants for SCR automata. They also provide appropriate feedback when a proof fails. These "automatic" TAME strategies were designed to be used with the SCR* toolset, which makes a wide variety of analyses as automatic as possible. The automatic SCR-specific strategies are made possible by the relatively simple SCR automata model and the fact that the state variables in SCR specifications are restricted to boolean, enumerated, and numeric types, for which the PVS decision procedures can handle much of the required reasoning. Appendix B contains a table listing the major PVS strategies.

Hand proofs of state invariants usually use proof steps from a fixed set. This fixed set of proof steps, implemented as general TAME strategies, is sufficient for proving most state invariant properties of most of the LV timed or I/O automata to which TAME has been applied. It is sometimes necessary to supplement the TAME strategies with standard PVS steps: for example, use of the PVS step INST for instantiating a universally quantified formula among the assumptions is common in TAME proofs. When the types of the state variables in an application are complex data types, the TAME steps need to be supplemented with proof steps for reasoning about these types. This happened in the Group Communication Service example from [18], mentioned in the introduction, in which certain state variables were queues accompanied by special operations for manipulating and observing their content. Despite the need to reason about these types in the

invariant proofs, the TAME strategies were still very useful in mechanizing the proofs.

The general TAME strategies designed for setting up proofs of state invariants include an induction strategy **AUTO_INDUCT**, which sets up a proof by structural induction over the reachable states of an automaton, and a strategy **DIRECT_PROOF**, which sets up a "direct" (i.e., non-induction) proof of a state invariant. (Here and below, names in bold capital letters refer to TAME strategies.) State invariant proofs are then completed by introducing a set of facts and applying straightforward reasoning such as propositional logic, equational logic, and standard facts about data types. Guidance in the form of case breakdowns is also sometimes used. TAME supplies several strategies for introducing facts, whose names describe their purpose, including **APPLY_SPECIFIC_PRECOND** for introducing the precondition of an action in an induction step, **APPLY_INV_LEMMA** and **APPLY_LEMMA** for using invariant lemmas and lemmas about datatypes, and **APPLY_IND_HYP** for applying the inductive hypothesis in an induction step.[1] TAME also supplies a strategy **TRY_SIMP** that does most of the straightforward reasoning required, and a strategy **SUPPOSE** for splitting (and labeling) cases.

Some of the general TAME strategies, such as **APPLY_INV_LEMMA** and **APPLY_LEMMA**, can also be used in proving other types of properties, including transition invariants, properties of timed executions, and (in principle) proofs of simulation. Although proof support for timed execution properties is not extensive, TAME does supply two special strategies for these properties (see Appendix B).

The automatic SCR-specific TAME strategies include **SCR_INV_PROOF**, which proves many state and transition invariants of SCR automata automatically, and **ANALYZE**, which is used to analyze dead ends in a proof for which **SCR_INV_PROOF** fails. **SCR_INV_PROOF** takes advantage of the result of an analysis by one of the other SCR* tools: the list of invariants produced by the automatic invariant generation algorithm [31]. These generated invariants can often be used to complete the proofs of dead ends reached by one of the basic invariant proof strategies underlying **SCR_INV_PROOF**. By calling **SCR_INV_PROOF$** instead of **SCR_INV_PROOF**, the user can determine from the saved TAME proof whether any of the invariants automatically generated by the SCR* invariant generator were used in the proof. This helps the user determine whether the proof succeeded for expected reasons.

Proof dead ends generated by **SCR_INV_PROOF** correspond to a class of transitions that may not preserve (or satisfy) the state (or transition) invariant. Applying **ANALYZE** to a dead end provides the user with a subgoal whose hypothesis contains information about this class of transitions, including:

[1] **APPLY_IND_HYP** is needed only in the rare case when the inductive hypothesis needs to be applied to values other than the Skolem constants from the inductive conclusion.

- Prestate and poststate values of state variables, if known,
- State variables known only to have unchanged values, and
- The input event corresponding to the transition class.

This information helps the user determine whether 1) an appeal to a state invariant will discharge the subgoal, or 2) some transition in the class is reachable, and the invariant being proved is false. In some cases, **ANALYZE** will describe the value of a variable in the poststate in terms of a complex expression involving many cases. Because it is unlikely that the invariant being proved would be true for a different reason in each case, **ANALYZE** does not further divide the subgoal with respect to these cases. A prototype translator from the output of **ANALYZE** into an easy-to-read format understandable by users of the SCR* toolset has been implemented.

3. TAME Specification Support for Automata

Because of the importance of the structure of TAME specifications to the TAME strategies, this section describes in detail how LV timed automata, I/O automata, and SCR automata are specified using TAME templates, how the templates are applied by a TAME user, and how specific features of the templates are used to advantage by the TAME strategies. Section 3.1 describes the template used for LV timed automata and I/O automata; because I/O automata are essentially LV timed automata without any references to time, the LV timed automaton template can be used to specify automata from either class. Simple default values are used for timing information in specifications of I/O automata. Section 3.2 describes how SCR automata are specified in TAME. Because SCR automata can be viewed as I/O automata, the LV timed automaton template could in principle also be used for SCR automata. However, as will be explained in Section 3.2, a slightly modified template is used for the sake of proof efficiency. Section 3.2 also describes how, in specifications of SCR automata, TAME benefits from analyses performed by other tools in the SCR* toolset.

3.1. LV Timed Automata and I/O Automata

The LV timed automata model [40], an extension of the I/O automata model [39], is designed to specify real-time algorithms and systems. In both models, a system is represented as a set of individual automata which interact by means of common actions. For verification purposes, these interacting automata can be composed into a single automaton by combining corresponding output and input actions. Every I/O or LV timed automaton is described by a set of states, some of which are initial states; a set of actions (input, output, and internal); and a transition relation coupling a state-action pair with another state. The current state is determined by the current values of the state variables. In a particular state, individual actions may or may not be enabled.

Transitions in I/O automata specifications are typically deterministic, so that the transition relation can be represented as a (partial) function on states and actions which maps a state and an action enabled in that state to a new state. For LV timed automata, transitions are also often specified deterministically; however, when this model is used in specifying a hybrid system, there may be nondeterminism [35,6]. Because experimentation has shown that proofs of state invariant properties for nondeterministic I/O or LV timed automata are more efficient if the transition relation is represented as a function, the values of variables that may be changed nondeterministically by an action are normally represented in TAME through use of the Hilbert choice operator ϵ.

Because the I/O automaton model and the LV timed automaton model are similar, TAME uses the same template for both. The essential features of this template are shown in Figure 1. The state of any LV timed automaton has three standard variables that are time-related. These are `now`, whose value is the current time, and `first` and `last`, whose values are functions mapping actions to the lower and upper bounds on their scheduled execution times. The "basic state" type defined in `MMTstates` is the type of the fourth state component, `basic`, which covers the remaining state variables. In addition to filling in the template, the user must supply any additional PVS type and constant declarations needed to support the definitions of `MMTstates` and `const_facts`.

Currently, instantiating the template for LV timed or I/O automata must be done by hand. For each application, the TAME strategies also require certain auxiliary local theories and strategies associated with the template instantiation. Although these auxiliary theories and strategies can in principle be generated from the particular template instantiation, they also must currently be produced by hand. An interface to generate a template instantiation and its auxiliary theo-

Template Part	User Fills In	Remarks
`actions`	Declarations of non-time-passage actions	**`actions`** is a PVS datatype
`MMTstates`	Type of the "basic state" representing the state variables	Usually a record type
`OKstate?`	An arbitrary state predicate restricting the set of states	Default is **`true`**
`enabled_specific`	Preconditions for all the non-time-passage actions	**`enabled_specific(a)`** = specific precondition of action **`a`**
`trans`	Effects of all the actions	**`trans(a,s)`** = state reached from state **`s`** by action **`a`**
`start`	State predicate defining the initial states	Preferred forms: **`s = ...`** or **`s = (# basic := basic(s) WITH ... ... #)`**
`const_facts`	Predicate describing relations assumed among the constants	Optional

Figure 1. Elements of the TAME template for LV timed and I/O automata.

ries and strategies from minimal information provided by the user is planned. The local theories contain lemmas that are used for rewriting and forward chaining to expedite reasoning about "obvious" properties of the PVS datatypes appearing in the template instantiation. The most important local strategy expands abbreviated representations of state variables that the user has used in filling in the template and in formulating properties to be proved.

The TAME strategies take advantage of several features of the template shown in Figure 1. First, they take advantage of the uniform naming conventions. Besides the standard names for the features (i.e., transition function, preconditions on the actions, start state predicate, etc.), there are standard naming conventions for the template instantiation and local strategies: the PVS theory corresponding to the template instantiation is named `<automaton-name>_decls`, where `<automaton-name>` is the name of the automaton, and the local strategies have names which can be computed from this name. The TAME strategies thus can refer to standard names to obtain definitions of standard automaton features, and compute the names of local strategies as needed. Second, the actions of an automaton are represented as a PVS datatype. The induction scheme used by TAME for proving state invariants is based on the induction scheme for this datatype, which is automatically provided by PVS. Third, the preferred form of the start state predicate—an equality of the state `s` to some record value—is used to create an efficient base case strategy within TAME's induction strategy. Finally, the separation of the definitions of the preconditions and effects of actions is used in supporting a separate proof step for applying the precondition of an action in an induction proof, making it possible to determine whether the precondition is actually needed in any particular induction case.

3.2. SCR and SCR Automata

SCR (Software Cost Reduction) is a formal method for specifying and analyzing the requirements of safety-critical control systems. Since its introduction in 1978, the SCR requirements method has been applied successfully to a wide range of critical systems, including avionics systems, space systems, telephone networks, and control systems for nuclear power plants. See, e.g., [26,43,17,16,41,38]. The SCR* toolset [25,22] is a set of tools that support development and analysis of SCR specifications. SCR* supports several complementary analysis techniques, including consistency checking, variable dependency analysis, invariant generation, model checking, and theorem proving (through TAME). An important goal of each SCR* tool is to perform its analyses as automatically as possible.

An SCR specification defines an SCR automaton, whose current state is determined by the values of state variables falling into four classes: *monitored* and *controlled variables* representing quantities in the system environment that the system monitors and controls, *mode classes* (whose values are called *modes*), and *terms*. Mode classes and terms are often used to capture historical information.

In the SCR automaton model, the system environment nondeterministically produces a sequence of input events, where an *input event* is a change in the value of some monitored variable. Executions of the system begin in some initial state, after which the system responds to each input event in turn by making a transition to a new state. An SCR specification includes assumptions about the environment that can affect the behavior (e.g., the possible changes of value) of monitored and controlled variables.

An SCR specification consists of a set of tables and dictionaries. Each dependent variable (i.e., mode class, term, or controlled variable) has a corresponding table that defines how the value of the variable is updated in response to an input event. Each controlled variable or term is defined by either an *event table* describing how to update the variable on the occurrence of various events (changes in the value of one or more variables under certain conditions) or a *condition table* describing how to update the variable under certain conditions. Each mode class is defined by a *mode transition table* that maps old modes to new modes when various events occur. Besides tables, an SCR specification contains dictionaries of *types*, *variable declarations*, *constant declarations*, *environmental assumptions*, and *assertions*. The assertion dictionary records required system properties. Our experience with practical systems is that most desired system properties are either state invariants or transition invariants.

The TAME template for SCR automata is identical to the template in Figure 1, except that `trans`, for efficiency reasons, is represented as a relation rather than a function. Matching an SCR specification to TAME's SCR template is straightforward, and, in fact, a prototype SCR-to-TAME translator has been implemented [8]. The various dictionaries and tables of an SCR specification can be translated naturally into various parts of the TAME template. Most parts of the template can be filled in from information in the dictionaries. The contents of the dictionaries of types and constant declarations are easily represented by PVS type definitions and constant declarations. The names and types of the fields in the record type `MMTstates` correspond to the names and types of variables in the variable declaration dictionary. The variable declaration dictionary indicates which state variables are monitored variables, and thus also provides the information needed to complete the definition of the `actions` datatype. In particular, the non-time actions in `actions` correspond to the monitored variables: for each monitored variable `m`, there is a constructor in `actions` to represent input events corresponding to changes in the value of `m`; this action constructor has as parameter the new value of `m`. The variable declaration dictionary is also used in a third way: because it records the initial values of the state variables, it is used in filling in the definition of the start state predicate `start`. The environmental assumptions dictionary records any restrictions on how monitored variables can change. Such restrictions are in effect preconditions on the input events, i.e., on the actions; therefore, they are used in completing the definition of `enabled_specific`. Besides the dynamic information on how monitored vari-

ables can change, the environmental assumptions dictionary may also contain static restrictions on the possible relations between monitored and controlled quantities in the environment. If present, such restrictions become conjuncts in the `OKstate?` predicate. Because SCR specifications do not contain any uninterpreted constants, specifying relationships among the constants is unnecessary; therefore, the axiom `const_facts` can be omitted (or given the default definition `TRUE`).

The tables of an SCR specification provide the information needed to fill in the final item in the template: the transition relation `trans`. Equation (1) shows the high-level form of `trans`:

$$\texttt{trans(s\_old,a,s\_new)} = (\texttt{s\_new} = F(\texttt{s\_old,s\_new})) \tag{1}$$

In more detail, F`(s_old,s_new)` is a case expression over the actions. Each action case corresponds to a change in a monitored variable that may result in changes to some of the dependent variables. The monitored variable's new value is the parameter to the action; changes to any dependent variables are computed from their respective tables. Thus the form of definition shown in (1) expands to:

```
trans(s_old,a,s_new) =
   s_new = CASES a of
      monvar1(monvar1value):
         s_old WITH { monvar1 := monvar1value,
                      depvar1 := update_depvar1(s_old,s_new)      (2)
                      ...  } ,
      monvar2(monvar2value):
         s_old WITH { ...  } ,
      ...
      ENDCASES
```

where `monvar1`, `monvar2`, etc. are monitored variables; `depvar1`, etc. are dependent variables; and `update_depvar1`, etc. are *update functions* that compute the updated values of dependent variables in terms of values of variables in the old state and new state. The update function for a given variable is in turn defined in terms of simpler functions of the old and new state that compute the values of the expressions appearing in various cells of the table for that variable. Thus, the full TAME representation of `trans` has multiple layers.

Information provided by other tools in the SCR* toolset—the results of disjointness checks and the variable dependency analysis—are used to determine the best representation of an SCR specification in TAME. Disjointness in the tables implies that the specified SCR automaton is deterministic, and permits the updated values of dependent variables to be specified using update functions rather than update relations. If disjointness fails to hold for one or more tables,

the updated values of the corresponding variables are nondeterministic; the form (2) above would have to be modified for each nondeterministic variable to replace the use of an update function with some other mechanism for reasoning about its value in the new state. Currently, TAME depends on the SCR specification passing all disjointness checks. Variable dependency analysis provides information about dependencies among values of variables in the new state. This information can be used to improve the efficiency of proving properties of an SCR automaton by minimizing the number of update assignments required to express its transition relation in the form (2).

As with LV timed and I/O automata, there are auxiliary local theories and strategies associated with any template instantiation of an SCR automaton in TAME. These local theories and strategies, which are generated automatically by the SCR-to-TAME translator, are more extensive than those for LV timed and I/O automata. The local theories contain additional lemmas relating to enumerated types (which are very common in SCR specifications), and the local strategies include strategies used to apply these lemmas and to expand the definition of the transition relation in a controlled way.

The TAME strategies for SCR automata exploit template features, auxiliary theories and strategies, and naming conventions analogously to the TAME strategies for LV timed and I/O automata. In addition, their implementation takes advantage of layering in the representation of the transition relation to achieve improvements in efficiency. Section 5.5 discusses how the efficiency of the automatic SCR proof strategies has been improved over time.

Because TAME requires SCR automata to be deterministic, it is in principle possible to represent the transition relation `trans` as a function, as in the LV timed automaton template. Because earlier experiments with LV timed automata found that representing `trans` as a relation made proofs of state invariants less efficient, initial experiments with representing SCR automata in TAME also defined `trans` as a function. The experience of a colleague [30] with proving properties of SCR automata directly in PVS led to experimentally representing `trans` as a relation instead. This led to dramatic improvements in proof efficiency for invariants of SCR automata. The reason for this reverse experience with SCR automata is probably that updating an SCR automaton state is done by successively propagating updated variable values to functions that use them to compute updated values of further variables, where a large number of variables and a large dependency depth may be involved. By contrast, in a typical LV timed automaton or I/O automaton, updating the state is usually accomplished by assigning new values to a few state variables in one stage, simultaneously. For SCR automata, representing `trans` in the relational form (2) shown above permits reasoning about an updated state without fully computing it.

4. Using the TAME Strategies

This section provides examples to illustrate how the general and automatic TAME strategies are used in proofs of invariants, and the feedback they provide to the user. The first two examples are typical TAME proofs of state invariants for an I/O automaton. The second two examples illustrate the use of the special TAME strategies for SCR automata.

4.1. Proving Properties of an I/O Automaton

Devillers et al. [15] define an I/O automaton called *TIP* that specifies a tree identify protocol, and identify a set of state invariants for *TIP*. Appendix A shows the specification of *TIP* from [15] and the invariants referred to in this section, and indicates how the notation from [15] is rendered in TAME. The invariants were proved by Devillers et al. using PVS directly. Hand proofs of the invariants were constructed by Devillers [14], but no attempt was made to follow the hand proofs in the PVS proofs. The hand proofs were constructed in the Lamport style [34], which represents a proof in detailed tree format, with a justification for each step.

Figure 2 shows the definition in TAME of two of the *TIP* invariants, Invariant I_{10} and Invariant I_{13}, together with the standard TAME formulation of the associated invariant lemmas.

```
Inv_10(s:states): bool = (FORALL (e:Edges):
    (NOT(mq(e,s)=null) & NOT(car(mq(e,s)))) => NOT(child(reverse_edge(e),s)));

lemma_10: LEMMA (FORALL (s:states): reachable(s) => Inv_10(s));

Inv_13(s:states): bool = (FORALL (e:Edges): root(target(e),s) => child(e,s));

lemma_13: LEMMA (FORALL (s:states): reachable(s) => Inv_13(s));
```

Figure 2. Sample invariants with their invariant lemmas in TAME.

TAME proofs of Invariants I_{10} and I_{13} are shown in Figures 3 and 4. When a user creates a proof interactively in PVS, PVS saves an executable script of the proof. This script records both the proof steps invoked by the user and the branching structure of the proof. In the example TAME proofs, the proof steps supplied by the user are shown in Roman font, the names of TAME strategies are in bold, and the parts of the proof scripts created by PVS are shown in italics. The italic numbers in quotes represent the addresses of the proof branches in the tree, and hence show the tree structure. The TAME proofs include comments (in italics, and preceded by semicolons) automatically generated by the TAME strategies.

The proofs in Figures 3 and 4 were constructed by Riccobene [44,7], who derived TAME proofs of all of the *TIP* invariants from Devillers' hand proofs.

```
("" 
 (AUTO_INDUCT)
    (("1" ;;Case add_child(addE_action)
     (SUPPOSE "e_theorem=addE_action")
        (("1.1" ;;Suppose e_theorem=addE_action
         (APPLY_SPECIFIC_PRECOND)
         (APPLY_INV_LEMMA "5" "e_theorem")
         (TRY_SIMP))
        ("1.2" ;;Suppose not [e_theorem=addE_action]
         (APPLY_SPECIFIC_PRECOND)
         (APPLY_LEMMA "lemma_aux" "addE_action" "e_theorem")
         (APPLY_INV_LEMMA "2" "e_theorem")
         (TRY_SIMP))))
    ("2" ;;Case ack(ackE_action)
     (SUPPOSE "ackE_action = e_theorem")
        (("2.1" ;;Suppose ackE_action = e_theorem
         (APPLY_SPECIFIC_PRECOND)
         (APPLY_INV_LEMMA "5" "e_theorem")
         (TRY_SIMP))
        ("2.2" ;;Suppose not [ackE_action = e_theorem]
         (TRY_SIMP))))
    ("3" ;;Case resolve_contention(resE_action)
     (SUPPOSE "resE_action=reverse_edge(e_theorem)")
        (("3.1" ;;Suppose resE_action=reverse_edge(e_theorem)
         (APPLY_SPECIFIC_PRECOND)
         (APPLY_LEMMA "lemma_aux" "resE_action" "e_theorem")
         (APPLY_INV_LEMMA "8" "e_theorem")
         (TRY_SIMP))
        ("3.2" ;;Suppose not [resE_action=reverse_edge(e_theorem)]
         (TRY_SIMP))))))
```

Figure 3. TAME proof (nonverbose) of lemma_10 from Figure 2.

```
("" 
 (AUTO_INDUCT)
 ;;Case root(rootV_action)
 (SUPPOSE "rootV_action = target(e_theorem)")
    (("1" ;;Suppose rootV_action = target(e_theorem)
     (APPLY_SPECIFIC_PRECOND)
     ;;Applying the precondition
     ;;NOT (init(rootV_action, prestate))
     ;;    & NOT (contention(rootV_action, prestate))
     ;;      & NOT (root(rootV_action, prestate))
     ;;       & (FORALL (e: tov(rootV_action)): child(e, prestate))
     (INST "specific-precondition_part_4" "e_theorem")
     (TRY_SIMP))
    ("2" ;;Suppose not [rootV_action = target(e_theorem)]
     (TRY_SIMP))))
```

Figure 4. TAME proof (verbose) of lemma_13 from Figure 2.

Each TAME proof was constructed following the case breakdown used in the corresponding hand proof, giving the TAME proof the same tree structure as the hand proof. The step **AUTO_INDUCT** was used for the initial breakdown into induction cases, and the TAME step **SUPPOSE** was used for the subsequent case splittings.[2] Appropriate TAME steps were used in each TAME proof to introduce the various facts at the point where they were appealed to in the hand proof. The TAME strategy **TRY_SIMP** was then used to supply the straightforward reasoning needed to complete the proofs. TAME was used in verbose mode in the proof of Invariant I_{13}, and in nonverbose mode in the proof of Invariant I_{10}. In verbose mode, the facts introduced by TAME steps are printed as comments in the saved proof.

The uses of **APPLY_LEMMA** in the proof in Figure 3 correspond to steps in the hand proof where the justification was "math". The mathematics used, which concerned properties of the type "Edges" referred to in Invariants I_{10} and I_{13}, was expressed as a set of lemmas in PVS and proved in PVS. Therefore, these steps required more creativity from the TAME user than the other steps in the proof. The use of the PVS step INST in the proof in Figure 4 illustrates the occasional need for a standard PVS step in a TAME proof.

As indicated above, TAME proofs, such as the proofs in Figures 3 and 4, that are constructed using TAME's general strategies can usually be understood completely without executing them in PVS. To illustrate this, Figures 5 and 6, respectively, show English explanations of the TAME proofs in Figure 3 and 4 generated by a prototype translator of saved TAME proofs. The proofs in Figures 5 and 6 can be checked by referring to the *TIP* specification and the *TIP* invariants, provided one knows that `e_theorem` is the Skolem constant for the universally quantified variable `e` in the lemma being proved, and that `specific-precondition_part_4` refers to the fourth conjunct of the specific precondition of the current action. The saved-proof translator could provide more detailed explanations by including the extra comments in verbose proofs, such as the comment following **(APPLY_SPECIFIC_PRECOND)** in Figure 4, in the explanations. Proof explanations such as those in Figures 5 and 6 could be further improved by generating them directly from the TAME strategies rather than from the saved TAME proof. This would permit yet further information, such as the invariant itself and the significance of any type-correctness proof obligations generated by PVS, to be incorporated into the explanations. Enhancements to the TAME strategies to accomplish this are planned.

Figures 7 and 8 show two subgoals that arise when the proof in Figure 4 is executed. PVS represents proof goals as Gentzen style sequents; thus the object for each goal is to prove that the conjunction of the formulae above the turnstile

[2] In two cases where the invariant being proved was a conjunction of simpler invariants, the PVS step SPLIT was used to create branches for the individual threads for the simpler invariants that were followed in the hand proofs.

(indicated by negative numbers) implies the disjunction of the formulae below the turnstile (indicated by positive numbers). The subgoals in Figures 7 and 8 illustrate TAME's feedback in the form of comments and formula labels during an interactive proof. The subgoal in Figure 7 is the unique subgoal returned after the TAME step **AUTO_INDUCT** is applied. The comment at the top of this subgoal shows that it corresponds to the induction case for the action `root(rootV_action)`. The actual parameter `rootV_action` is the automatically

---

Proof. The proof is by induction. The base case is trivial. There are 3 nontrivial action cases.

• **Consider the action** `add_child(addE_action)`. The proof in this case is as follows. Suppose first that `e_theorem` = `addE_action`. Apply the precondition of the action `add_child(addE_action)`. Apply Invariant *5* to `e_theorem`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `e_theorem` = `addE_action`. Apply the precondition of the action `add_child(addE_action)`. Apply lemma *lemma_aux* to `addE_action` and `e_theorem`. Apply Invariant *2* to `e_theorem`. The rest of the proof in this case is obvious. This completes the proof for the action `add_child(addE_action)`.

• **Consider the action** `ack(ackE_action)`. The proof in this case is as follows. Suppose first that `ackE_action` = `e_theorem`. Apply the precondition of the action `ack(ackE_action)`. Apply Invariant *5* to `e_theorem`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `ackE_action` = `e_theorem`. The rest of the proof in this case is obvious. This completes the proof for the action `ack(ackE_action)`.

• **Consider the action** `resolve_contention(resE_action)`. The proof in this case is as follows. Suppose first that `resE_action` = `reverse_edge(e_theorem)`. Apply the precondition of the action `resolve_contention(resE_action)`. Apply lemma *lemma_aux* to `resE_action` and `e_theorem`. Apply Invariant *8* to `e_theorem`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `resE_action` = `reverse_edge(e_theorem)`. The rest of the proof in this case is obvious. This completes the proof for the action `resolve_contention(resE_action)`. □

Figure 5. English explanation of the TAME proof in Figure 3.

Proof. The proof is by induction. The only nontrivial case is the single action case `root(rootV_action)`.

• **Consider the action** `root(rootV_action)`. The proof in this case is as follows. Suppose first that `rootV_action` = `target(e_theorem)`. Apply the precondition of the action `root(rootV_action)`. Instantiate specific-precondition_part_4 with the value `e_theorem`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `rootV_action` = `target(e_theorem)`. The rest of the proof in this case is obvious. This completes the proof for the action `root(rootV_action)`. □

Figure 6. English explanation of the TAME proof in Figure 4.

```
lemma_13 :
;;;Case root(rootV_action)

{-1,pre-state-reachable}
    reachable(prestate)
{-2,inductive-hypothesis}
    (root(basic(prestate))(target(e_theorem))
          => child(basic(prestate))(e_theorem))
{-3,general-precondition}
    enabled_general(root(rootV_action), prestate)
{-4,specific-precondition}
    enabled_specific(root(rootV_action), prestate)
{-5,post-state-reachable}
    reachable(poststate)
{-6,inductive-conclusion_part_1,inductive-conclusion}
    root(basic(prestate)) WITH [(rootV_action) := TRUE](target(e_theorem))
  |-------
{1,inductive-conclusion_part_2,inductive-conclusion}
    child(basic(prestate))(e_theorem)
```

Figure 7. Subgoal produced by AUTO_INDUCT in Figure 4.

```
lemma_13.1:
;;;Applying the precondition
;;;NOT (init(rootV_action, prestate)) &
;;; NOT (contention(rootV_action, prestate)) &
;;;  NOT (root(rootV_action, prestate)) &
;;;   (FORALL (e: tov(rootV_action)): child(e, prestate))

[-1,(Suppose)]
      rootV_action = target(e_theorem)
[-2,(pre-state-reachable)]
      reachable(prestate)
[-3,(inductive-hypothesis)]
      root(basic(prestate))(target(e_theorem)) =>
       child(basic(prestate))(e_theorem)
[-4,(general-precondition)]
      enabled_general(root(rootV_action), prestate)
{-5,(specific-precondition_part_4 specific-precondition)}
      FORALL (e: tov(rootV_action)): child(basic(prestate))(e)
[-6,(post-state-reachable)]
      reachable(poststate)
[-7,(inductive-conclusion_part_1 inductive-conclusion)]
      root(basic(prestate)) WITH [(rootV_action) := TRUE](target(e_theorem))
  |-------
{1,(specific-precondition_part_1 specific-precondition)}
      init(basic(prestate))(rootV_action)
{2,(specific-precondition_part_2 specific-precondition)}
      contention(basic(prestate))(rootV_action)
{3,(specific-precondition_part_3 specific-precondition)}
      root(basic(prestate))(rootV_action)
[4,(inductive-conclusion_part_2 inductive-conclusion)]
      child(basic(prestate))(e_theorem)
```

Figure 8. Subgoal after APPLY_SPECIFIC_PRECOND in Figure 4.

generated Skolem constant for the formal parameter `rootV` of the parameterized action `root`. Besides generating comments distinguishing the cases in an induction proof, **AUTO_INDUCT** gives all the formulae labels that indicate their significance. The subgoal in Figure 8 is from deeper in the proof in Figure 4, after **SUPPOSE** and **APPLY_SPECIFIC_PRECOND** have been applied. The comment at the top of this subgoal shows the precondition that has been introduced. The supposition introduced with **SUPPOSE** is labeled `Suppose`, and the individual parts of the specific precondition both retain their label `specific-precondition` and acquire individual labels of their own indicating where they came from in the introduced precondition.

4.2. Proving Properties of an SCR Automaton

As indicated in Section 2.2, there is a single automatic TAME strategy, **SCR_INV_PROOF**, for proving both state and transition invariants. The strategy **SCR_INV_PROOF** first applies a strategy appropriate to the particular invariant which takes the proof as far as possible without appealing to other invariants. State invariants usually are proved by induction. Transition invariants (as noted in Section 2.1) are usually not proved by induction, since the fact that all the transitions from a given state satisfy the transition invariant rarely implies anything useful about whether the transitions from its successor states satisfy the invariant. A transition invariant is normally proved by directly applying the definition of the transition relation `trans` (see Section 3.2) and, if necessary, by appealing to other invariants. Thus, for state invariants, **SCR_INV_PROOF** first calls the special SCR induction strategy **SCR_INDUCT_PROOF**, based on the general induction strategy **AUTO_INDUCT**, and for transition invariants, **SCR_INV_PROOF** first calls the special SCR strategy **SCR_DIRECT_PROOF**, based on the general non-induction strategy **DIRECT_PROOF**. If the initial strategy applied by **SCR_INV_PROOF** does not succeed in completing the proof, **SCR_INV_PROOF** determines whether application of invariants generated by the SCR* automatic invariant generation algorithm will complete the proof. As noted in Section 2.2, the user can determine which, if any, generated invariants were needed in the proof by using **SCR_INV_PROOF$** instead of **SCR_INV_PROOF**.

Figure 9 shows the formulation of a transition invariant for an SCR automaton called *CD* (for communications device) [29]. The invariant says that if, in a transition, backup power goes undervoltage when primary power is unavailable, the mode of operation of the communication device after the transition is either off or alarm.

The TAME strategy **SCR_INV_PROOF** succeeds in proving the transition invariant in Figure 9. Using **SCR_INV_PROOF$** instead saves the proof in the form shown in Figure 10. This saved proof shows that **SCR_INV_PROOF** has proved Invariant *CD_4* by first applying **SCR_DIRECT_PROOF** and then, on

```
Inv_CD_4(s:states):bool =
  ( FORALL (a: actions, new_s:states): enabled(s,a,new_s) & trans(s,a,new_s) =>
     ((not(BackupPower(s) = undervoltage) & BackupPower(new_s) = undervoltage
       & PrimaryPower(s) = unavailable)
      => (Operation(new_s) = Alarm OR Operation(new_s) = Off)) );

lemma_CD_4: LEMMA ( FORALL (s: states): reachable(s) => Inv_CD_4(s) );
```

Figure 9. Invariant *CD_4* of the SCR automaton *CD*.

```
("" 
 (SCR_DIRECT_PROOF)
    (("1" (APPLY_INV_LEMMA "Operation_Initialization"))
     ("2" (APPLY_INV_LEMMA "Operation_Configuration"))
     ("3" (APPLY_INV_LEMMA "Operation_Idle"))
     ("4" (APPLY_INV_LEMMA "Operation_TrafficProcessing"))))
```

Figure 10. Proof generated for Invariant *CD_4* by **SCR_INV_PROOF$**.

each of four unfinished subgoals in the proof, applying an appropriate generated invariant. The names of the invariants used indicate that they were generated from the table for the variable `Operation` in the SCR specification of *CD*.

When a proof using **SCR_INV_PROOF** fails, the user can apply the TAME strategy **ANALYZE** to any dead end in the proof to create a sequent which exhibits the details about the class of problem transitions associated with the dead end. Figure 11 shows a dead end reached in the proof of Invariant *WCP_5* for an SCR automaton *WCP* (weapons control panel) [21]. Invariant *WCP_5* expresses the property "When the pressure-hold signal is received, the pressurize valve shuts and the vent-blocking valve opens." Figure 12 shows an extract from the sequent produced by applying **ANALYZE** to the subgoal in Figure 11. In these sequents, `prestate` represents the prestate of the transition, `new_s_theorem` represents the poststate, and `a_theorem` represents the action. The complex formula expressing the fact that `prestate`, `a_theorem`, and `new_s_theorem` satisfy `trans` is hidden, but used in obtaining the second sequent from the first.

The second formula in Figure 12 is included as an example of a formula stating explicitly that a variable has not changed; the other such formulae are represented by ellipses. The formulae containing an ellipsis abbreviate formulae equating the value of a variable in the new state with a complex expression. The remaining formulae give explicit values known for either old-state or new-state variables. Because `mPRESSURE_HOLD` (corresponding to the pressure-hold signal) is a monitored variable of *WCP*, the first and fourth formulae in Figure 12 indicate that the input event corresponding to this transition class is `mPRESSURE_HOLD` changing from `FALSE` to `TRUE`.

```
lemma_WCP_5 :

{-1,(conclusion)}  mPRESSURE_HOLD?(a_theorem)
{-2,(conclusion)}  mPRESSURE_HOLD_value_of(a_theorem)
{-3,(poststate-reachable)}  reachable(new_s_theorem)
{-4,(prestate-reachable)}  reachable(prestate)
{-5,(general-precondition)}  enabled_general(a_theorem, prestate)
  |-------
{1,(conclusion)}  mPRESSURE_HOLD_part(basic(prestate))
```

Figure 11. Dead end in the proof of Invariant *WCP_5*.

```
lemma_WCP_5 :

{-1,(analysis)}  mPRESSURE_HOLD(prestate) = FALSE
{-2,(analysis)}
      tTRANS_SEL_B_FAIL(new_s_theorem) = tTRANS_SEL_B_FAIL(prestate)
   ...
{-20,(analysis)}  tPRESSURIZING(new_s_theorem) = FALSE
   ...
{-31,(analysis)}  mPRESSURE_HOLD(new_s_theorem) = TRUE
   ...
{-38,(analysis)}  cPVC_PRESSURIZE_SOLENOID(new_s_theorem) = FALSE
   ...
{-39,(analysis)}  tVENTING(new_s_theorem) = ...
{-40,(analysis)}  tPRESSURIZING_LATCH(new_s_theorem) = ...
{-41,(analysis)}  tPRESSURE_HOLD_LATCH(new_s_theorem) = ...
   ...
  |-------
   ...
```

Figure 12. Result of applying **ANALYZE** to the subgoal in Figure 11.

The strategy **ANALYZE** is designed to support giving a user of the SCR* toolset feedback about proof dead ends in the notation of SCR. The ultimate goal is to permit an SCR* user to apply TAME and understand feedback from it without having to interact directly with PVS (or TAME). As noted in Section 2.2, there is a prototype translator of information such as that in Figure 12 into SCR notation. In SCR notation, the value of a variable in the new state is distinguished from the value of that variable in the old state by the addition of a prime. Thus, in the translation of Figure 12, the first formula becomes `mPRESSURE_HOLD = FALSE`, the second becomes `tTRANS_SEL_B_FAIL′ = tTRANS_SEL_B_FAIL`, the third becomes `t_PRESSURIZING′ = FALSE`, and so on. The translation also presents the information about the state variables in a more readable order. Although this information could be deduced by a sophisticated user of TAME and PVS from the sequent in Figure 11 extended by revealing the hidden transition relation formula, this extended sequent is not in a form easily converted into SCR notation, and in the general case may refer to update functions requiring computation.

5. Implementing the TAME strategies in PVS

PVS was not originally designed to support strategies for special-purpose theorem proving, but rather to support direct use of the theorem prover in a user-friendly manner. As a result, many PVS proof commands tend to be the "wrong size" to use as building blocks for strategies, because they do too much for the user. Further, users were not expected to need to observe any information about the current proof state except the currently unproved goals, or to require any access to the PVS internals. These original design decisions created difficulties in implementing each of the interfaces described in [49,1,33]. The Duration Calculus interface required certain modifications to PVS, provided by a co-author of [49], who is also one of the PVS developers. The developers of the TRIO interface, who lacked this advantage, used a somewhat unnatural encoding of the TRIO logic in PVS. The DisCo interface, which uses PVS strategies to implement inference rules from Lamport's Temporal Logic of Actions, compiles separate strategies for each application. TAME avoided most of these problems by exploiting several new features added to PVS to support special-purpose interfaces. These new features, which are described later in this section, both make PVS a more "open", programmable system and support control and feedback in some ways not available in analogous systems.

This section discusses the resources available in PVS for developing user-defined strategies, and then describes how the TAME strategies have been implemented and refined. Section 5.1 discusses how the proof rules supplied by PVS differ from those of other higher-order-logic theorem provers, and how this can affect the implementation of strategies. Section 5.2 describes the strategy-building commands in PVS, from which much of the control possible in user-defined strategies is derived. Section 5.3 discusses the PVS features that were especially helpful in implementing the TAME strategies. It also identifies those features new in PVS, and those features not provided in analogous systems. Section 5.4 illustrates how the various features were used to advantage in implementing the general TAME strategies. Finally, Section 5.5 discusses how the SCR-specific TAME strategies are implemented and how their efficiency has been improved.

5.1. Using PVS Proof Steps in Strategies

As noted above, a major reason for choosing PVS as the basis for TAME is the availability of the rule SIMPLIFY and its variants, which apply rewrite rules together with decision procedures for propositional simplification, linear arithmetic, and equality to handle much of the low-level reasoning in a mechanized proof. The controlled used of SIMPLIFY relieves the strategy developer, like the direct user of PVS, of much tedium.

Developing user-defined strategies in PVS is similar to developing user-

defined tactics in other higher-order-logic theorem provers such as HOL, Isabelle, and Coq, but is different in one significant way: fine control of the effect of a sequence of proof steps in PVS can be difficult to achieve because the exact effects of some of the proof rules supplied by PVS are not completely predictable. Although the soundness of the PVS proof rules is grounded in a set of straightforward inference rules, the proof steps they effect do not always correspond exactly to applications of individual basic or derived inference rules. Rather, they may perform many transformations and simplifications in a single step, and may also perform hidden operations that can influence the amount of extra simplification performed by subsequent PVS proof steps. When PVS is used directly, this feature is normally a convenience to the user. While it can shorten the number of steps required in a strategy, and thus be a convenience in strategy development, it can also cause difficulty for the strategy developer.

To illustrate the difficulty that can be caused by overeager simplification, consider a particular case that initially caused a problem in TAME. When the primitive PVS rule EXPAND is used to expand the definition of the outermost function in an expression, the expansion may be accompanied by additional simplifications of the resulting expression and additional transformations of its context. As a result, the form of a context in which a definition has been expanded may or may not change, and this can make it difficult to select a uniform next step in a strategy. This caused a problem in developing the strategy **AUTO_INDUCT**. In doing standard reasoning about the inductive conclusion in the induction step for some action `a`, **AUTO_INDUCT** must simplify the expression `inv(trans(a,s))`. This expression asserts that the invariant `inv` holds for `trans(a,s)`, the value of the state reached from `s` by a transition on the action `a`. Whenever, for a particular action `a`, applying EXPAND to `trans` in `inv(trans(a,s))` simplifies `trans(a,s)` to an expression of the form `IF b THEN s1 ELSE s2`, extra transformations are then applied that change `inv(IF b THEN s1 ELSE s2)` into `IF b THEN inv(s1) ELSE inv(s2)`. When `inv` is a universally quantified invariant, expanding its definition then leads to two embedded universal quantifiers, each of which must ultimately be skolemized. This led to a problem with finding a uniform way for **AUTO_INDUCT** to coordinate the skolemization of the inductive conclusion with the instantiation of the inductive hypothesis. This particular problem can be solved by expanding the definition of `inv` before expanding that of `trans`; however, the need to order definition expansions complicates the process of designing a strategy.

To some extent, one can achieve the effect in PVS of applying individual inference rules by supplying special arguments to PVS steps to focus them on particular formulae or limit the operations they perform. Recent enhancements to PVS include the addition of several finer-grained steps which increase the degree to which this can be done. For example, a version of EXPAND that simply expands a function definition can now be effected by supplying an optional extra argument to EXPAND. This finer-grained EXPAND is what TAME uses

to circumvent the problem of coordinating skolemization and instantiation in its induction strategies.

A second difference between the proof rules provided by PVS and those of analogous higher-order logic theorem provers that can complicate strategy development is that they occasionally generate "extra" subgoals corresponding to type correctness conditions (TCCs). This difference exists because of the richness of the PVS type system, which permits predicate subtypes and dependent types. The possible generation of these extra subgoals must be considered when defining a strategy whose correctness depends on known branching structures arising at certain points in its execution.

5.2. The Strategy Language of PVS

The standard PVS commands can be classified into rules and "strategicals"[3] (i.e., strategy-building commands), the latter being PVS commands analogous to the tacticals of other theorem provers such as HOL, Isabelle, or Coq. Proof commands in PVS are either *primitive rules*, *defined rules*, or *strategies* built from rules and other strategies with the strategicals. The strategicals include commands for sequencing, backtracking, choosing the next proof command, treating generated subgoals in distinct ways, repetition, and other purposes. Most of the PVS strategicals have close analogues in other theorem proving systems, but two seem to be unique: APPLY, which turns a strategy into a defined rule (i.e., makes the strategy act as a single proof step), and WITH-LABELS, which is used to label the new formulae inside the subgoals generated by a proof step. A table listing and describing the major PVS strategicals can be found in Appendix C.

Combining PVS proof rules using the strategicals produces simple PVS strategy expressions that can be used as strategies directly. However, it is possible to define more complex strategies using Common Lisp, the implementation language of PVS. To support the use of Common Lisp in strategy definitions, PVS provides a Lisp macro `defstep`. This macro expects as arguments the strategy name, a list specifying the strategy's arguments, the strategy body, and two strings used to describe the successful and error behaviors of the strategy when it is used interactively. The strategy body is a Lisp expression which, when the strategy's arguments are supplied, must evaluate to a simple PVS strategy expression; thus, user defined rules and strategies created using `defstep` are always conservative extensions of the standard set of PVS proof rules, in the sense that no inference can be made by a user defined rule or strategy that could not be made by applying an appropriate selection of standard PVS proof rules with appropriate arguments.

[3] The term "strategicals" is used in this paper for brevity; it is not official PVS terminology.

5.3. PVS Features Useful in Strategy Development

PVS provides several features useful in developing user-defined strategies:

1. Availability of "baby steps"
2. Support for backtracking
3. Support for rewriting
4. The ability to define and apply generic lemmas
5. Access to data structures and functions used by the proof engine
6. Support for forward chaining
7. The ability to hide and reveal formulae
8. Support for labeling formulae
9. Support for comments

The first six of the above features are available in other higher-order logic systems to a greater or lesser degree than in PVS. The last three features appear to be unique to PVS. All nine features are used to advantage by TAME's strategies.

Features 5, 8, and 9 are recent additions to PVS. As noted above, Feature 1, which can be necessary for fine control in user-defined strategies, has been recently enhanced. The absence of Feature 5 created many of the difficulties mentioned above for the earlier PVS interfaces for TRIO [1] and DisCo [33] as well as difficulties in the earliest version of TAME.

5.4. Implementing Natural Proof Steps in TAME

This section provides four examples that illustrate the techniques by which TAME's general strategies are implemented using the PVS features described in Section 5.3. Each example first describes a natural proof step supported in TAME, and then describes how the TAME strategy providing this step is implemented. These examples include **AUTO_INDUCT**, one of the most complex TAME strategies; **APPLY_IND_HYP**, representative of a very simple TAME strategy for which **AUTO_INDUCT** lays the groundwork; **TRY_SIMP**, a strategy that relies heavily on the local theories and strategies generated for a specification; and **USE_EPSILON**, a strategy that uses access to the proof state to relieve the user from having to supply many tedious details to accomplish a simple proof step.

Example 1: **AUTO_INDUCT.** This strategy performs the work represented in a typical natural language proof of a state invariant by the phrase "the proof is by induction". It both sets up the induction proof and performs many of the standard steps that are implicit at the beginning of a natural language proof. The induction being referred to is structural induction over the reachable states of an automaton, in which the proof is broken down into cases: a base case,

and an induction step for each action. In a typical hand proof "by induction", the expansion of certain definitions—such as (for induction steps) the effects of the individual actions and the invariant in the prestate and the poststate of the transition—is implicit. In addition, when any of the cases—the base case and any of the action cases—is trivial, this is often noted, with no further discussion of the case. Finally, many state invariants (such as those in Figure 2 in Section 4) are universally quantified, and in the proof of such invariants, it is very common for the quantified variable to be specialized in the same way in both the inductive hypothesis ("the invariant holds in the prestate") and the inductive conclusion ("the invariant holds in the poststate") in every induction step. Doing this usually (though not invariably) is a sufficient application of the inductive hypothesis in the proof. **AUTO_INDUCT** performs all the steps indicated above automatically. In addition, any proof goals it returns are formulated in a way that supports the subsequent application of other TAME steps.

The techniques used in implementing **AUTO_INDUCT** use almost all of the features discussed in Section 5.3. **AUTO_INDUCT** uses two generic lemmas from a generic parameterized theory `machine`. The first generic lemma, which expresses the equivalence of a structural induction proof that an invariant holds in all reachable states to a proof of the same fact by mathematical induction over the number of automaton steps to reach a state, is used in transforming the original goal of proving a property to be an invariant into a structural induction proof. The second generic lemma infers the reachability of the poststate in a transition from the reachability of the prestate of the transition, and includes it among the hypotheses, to facilitate applying invariant lemmas to the poststate as easily as to the prestate.

Access to the PVS data structures is used by **AUTO_INDUCT** to obtain the list of constructors from the `actions` data type, together with the parameter list of each constructor. This information is then used in computing standard Skolem constant names for an action's parameters, in computing an expression for each action in which the action constructor is applied to the Skolem constants, and, finally, in computation of a command list argument to the PVS strategical BRANCH. The resulting BRANCH command performs the standard initial steps, such as definition expansions, in the individual branches of an induction proof, and uses the comment feature to annotate the branches, annotating the base case with *Base case* and each action case with the expression computed for the action (as in Figures 3 and 4). Using the same specializations for universally quantified variables in both inductive hypothesis and inductive conclusion in each induction step is done by coordinating skolemization of the inductive conclusion with instantiation of the inductive hypothesis. To do this, **AUTO_INDUCT** refers to these formulae by the labels `inductive-hypothesis` and `inductive-conclusion` it has attached to them as they are produced. As noted above, care has to be taken when expanding the definition of the transition relation `trans` to avoid a transformation of the inductive conclusion that interferes with its skolemization.

Thus, the "baby step" version of EXPAND is used in expanding `trans`.

AUTO_INDUCT attempts to discharge any subgoals among the base and action cases that a human would consider trivial. It first installs the lemmas from a mix of generic and application-specific theories as auto-rewrites in PVS. Thereafter, these rewrite rules are invoked by any appeal to the PVS command ASSERT. A subsidiary strategy that usually proves the base case automatically does a substitution based on the start state formula (using its label `start-state`), provided this formula is in the preferred form of an equality (see Figure 1), and then calls ASSERT. Typically, several of the action cases are also trivial, and can be proved using the sequence of PVS steps LIFT-IF, PROP, ASSERT. Because this sequence may split the subgoal, **AUTO_INDUCT** uses backtracking, if necessary, to avoid returning multiple subgoals. Thus, the proof attempt is actually done with the (auxiliary) strategy

```
(APPLY (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL))),        (3)
```

which executes the above PVS steps and backtracks if they fail to prove the subgoal.[4] Experience with checking natural language proofs of invariant properties with TAME has shown that there is a close correlation between the action cases considered trivial by various authors [24,37,35,50] and the action cases proved automatically by the strategy (3).

Example 2: **APPLY_IND_HYP**. As indicated in Example 1, for a universally quantified invariant, the appropriate instantiation of the inductive hypothesis is usually the Skolem constant or constants from skolemization of the inductive conclusion. However, this is not always the case; it may be required in a proof to apply the inductive hypothesis to a different argument or arguments. The implementation of the strategy **APPLY_IND_HYP** that performs this step is very simple: it temporarily reveals the hidden formula labeled `inductive-hypothesis` that was created by **AUTO_INDUCT**, and instantiates it with whatever arguments the user supplies. This instantiation leads to the formula being hidden again (by PVS).

Example 3: **TRY_SIMP**. This strategy attempts to implement the proof step "it is now obvious" in proofs of properties of automata. PVS supplies a command GRIND, whose purpose is to prove many assertions totally automatically, but this command does both more and less than what is needed for the "it is now obvious" step in TAME.[5] **TRY_SIMP** does more than GRIND by more extensively automating simple reasoning about data types. For example, if `con` and `des` are a corresponding constructor-destructor pair in a datatype `A`, it is obvious to a human that `con(des(a))` = `a` whenever `a` is a "`con`" value of `A`.

[4] For the meanings of the PVS steps and "strategicals" in this strategy, see Appendix C.

[5] The command GRIND can also be called with arguments to make it do somewhat less, but even then does more than desired for **TRY_SIMP**.

TRY_SIMP also makes this inference, but GRIND does not. **TRY_SIMP** does less than GRIND by not performing the skolemization, instantiation, universal definition expansion, and massive rewriting done by GRIND. Because it does less than GRIND in these respects, a call to **TRY_SIMP** in the middle of a proof produces subgoals that are less obscure than those produced by a call to GRIND. While **TRY_SIMP** can be used in the middle of a proof, it is more appropriately used as the final step in the proof of a goal (as in Figures 3 and 4), because it does not document in a saved proof the significance of any case splits it causes.

The implementation of **TRY_SIMP** depends heavily on the auxiliary theories generated from an instantiation of the automaton specification template. The lemmas in these theories are used as temporary auto-rewrites and for forward chaining. **TRY_SIMP** also uses propositional simplification, the PVS decision procedures, and substitution of equals for equals. Because quantified formulae are usually irrelevant in the final stage of proving a subgoal, **TRY_SIMP** uses access to the proof state to identify, label, and hide them. In the rare cases where the quantified formulae are in fact relevant, **TRY_SIMP** continues simplification after restoring these formulae to the sequent. While this approach causes **TRY_SIMP** to take longer in a few cases, it nearly always increases the execution speed of **TRY_SIMP** noticeably when quantified formulae are present.

Example 4: **USE_EPSILON**. As indicated in Section 3.1, a convenient way to specify hybrid automata in which the effects of time passage on variables representing quantities in the environment are known only approximately is to specify the new value of each such variable as `epsilon(p)`, where `epsilon` is the Hilbert ϵ, and `p` is an appropriate predicate describing the set of possible values.[6] When reasoning about the ϵ expression `epsilon(p)`, one must usually introduce the fact that `p` is true of `epsilon(p)`. This is done in PVS by applying an ϵ axiom from the parameterized theory `epsilons[t:TYPE]` in the PVS prelude:

```
epsilon_ax:  AXIOM
   FORALL (p:pred[t]):                                    (4)
      (EXISTS (x:t):  p(x)) => p(epsilon(p))
```

Applying this axiom poses several inconveniences to the PVS user. First one must introduce a particular instance of the axiom—that is, one must supply the type `t` of the argument of the predicate `p`. If only one ϵ expression involving a predicate with a particular argument type `t` is present, the ϵ axiom can be applied to that predicate with the PVS command `USE("epsilon_ax[t]")`. However, if more than one such ϵ expression is present (as can happen in proofs of properties of hybrid automata), a second inconvenience is that one must also supply the

[6] This use of `epsilon` is valid for establishing invariant properties of a hybrid automaton.

predicate `p`. The declaration of one such predicate `p` taken from the TAME representation [6] of the steam boiler controller specification from [35], is:

```
water_level_pred(q_old:water_level,
                 pr:num_pumps,
                 v_old, v_new:nonnegreal,
                 delta_t:(fintime?))
                 (q_new:water_level):bool = ...
```
(5)

The full name of this (parameterized) predicate is a complex expression involving five parameters, and being required to type in this expression (no mistakes allowed!) simply to apply the ϵ axiom would be an extreme inconvenience. Once the appropriate application ϵ axiom has been introduced as a new formula, there is a third inconvenience: additional proof steps are needed to expand the definition of the predicate and split off the hypothesis of the ϵ formula as a proof obligation.

The strategy **USE_EPSILON** allows the user to apply the Hilbert ϵ axiom with only minimal effort when proving invariants of hybrid automata. Note that the ϵ axiom is only used in a proof to introduce facts about an ϵ expression already present in the current subgoal. As a result, the first two inconveniences noted above can normally be eliminated by probing the current proof state to deduce both the actual parameters of the parameterized predicate and its argument type simply from the parameterized predicate's name. This information can usually be deduced because it is rare to have two ϵ expressions present involving the same parameterized predicate. The third inconvenience—the additional proof steps required—is easily eliminated by including these steps in the strategy. Thus, when an instance `epsilon(water_level_pred(...))` is present, the user (usually) needs only to type

```
(USE_EPSILON "water_level_pred")
```

to introduce the fact that the instance `epsilon(water_level_pred(...))` satisfies the expanded `water_level_pred(...)` into the proof (with an accompanying existence proof obligation).

5.5. Implementing Automatic Strategies in TAME

This section discusses the implementation and efficiency of the two SCR-specific strategies **SCR_INV_PROOF** and **ANALYZE**. It describes how the strategies **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF**, on which **SCR_INV_PROOF** is based, have been gradually made more efficient by an order of magnitude. It also discusses the tradeoff in efficiency between **SCR_DIRECT_PROOF** and **SCR_INDUCT_PROOF** on the one hand, and **ANALYZE** on the other.

Implementing **SCR_INV_PROOF**. As noted in Section 4.2, the strategy **SCR_INV_PROOF** applies either **SCR_INDUCT_PROOF** or **SCR_DIRECT_PROOF**, and then tries to discharge any resulting subgoals by applying state invariant lemmas generated by the SCR* invariant generation algorithm. These invariants are currently translated by hand into TAME; a future version of the SCR-to-TAME translator will translate them to TAME automatically from the assertions dictionary in the SCR specification (see Section 3.2). **SCR_INV_PROOF** uses access to the proof state to determine whether the invariant being proved is a state invariant or transition invariant, and calls **SCR_INDUCT_PROOF** or **SCR_DIRECT_PROOF** as appropriate. **APPLY_INV_LEMMA** with backtracking is then used with each invariant lemma; if this fails, pairs of lemmas are tried, and so on. A more efficient version of this stage is planned that takes advantage of access to the current proof state to select lemmas intelligently.

The strategies **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF** are quite similar after their initial stages based on **AUTO_INDUCT** or **DIRECT_PROOF**. Following these initial stages, the transition relation is available in every goal in the form of an equality of the next state to some function of the current state and the next state. This equality is either in the form of the right hand side of (2) in Section 3.2 or else further simplified from this form. Both strategies continue by alternating substitution of the right hand side of the transition relation equality for the current state with simplification using case splitting, the PVS decision procedures, and other standard manipulations. How the efficiency of these strategies has evolved over time is described below.

Efficiency of **SCR_DIRECT_PROOF** *and* **SCR_INDUCT_PROOF**. The major factor in the efficiency with which the automatic proof strategies for invariants of SCR automata execute is use of the transition relation. The size of the formula expressing the transition relation is essentially quadratic in the number of state variables. This is because it is a `CASE` expression with cases corresponding to the monitored variables, in which each case defines how to update the dependent variables. The variable dependency analysis performed by the consistency checker in the SCR* toolset provides information that can be used to determine the set of dependent variables whose value will change when any given monitored variable changes value. This information can be used to minimize the size of the transition relation formula by minimizing the number of variable updates appearing in the case in the `CASE` expression corresponding to the given monitored variable. Doing this resulted in a dramatic (approximately twofold) improvement in the execution speed of proofs of properties of the SCR automaton *CD* mentioned in Section 4.2 (see also [29]).

Besides minimizing the size of the transition relation formula at the outset, carefully managing the use of this formula during execution of **SCR_DIRECT_PROOF** and **SCR_INDUCT_PROOF** has been the largest factor in

proof efficiency. The layered structure of the full definition of the transition relation `trans` shown in formula (2) in Section 3.2 has proved particularly important in managing the use of the transition relation formula. Early versions of the automatic invariant strategies did not take advantage of this layering, but instead, expanded the transition relation formula completely as soon as it was available in a proof subgoal. In these versions, after `trans` is expanded, all the update functions appearing in the resulting formula are immediately expanded, after which the large set of functions corresponding to the individual cells of the tables from which the update functions are derived are immediately expanded. The result is a formula equating the new state to a huge expression. After this stage, simplification that repeatedly applies propositional simplification and ASSERT often completes the proof of the subgoal. However, to complete the proof, it is sometimes necessary to substitute the huge expression for multiple occurrences of the new state in the sequent, compounding the size problem, and then simplify and (if necessary) repeat the process. This method is a brute force approach, but has some virtues: it is straightforward, easily captured in the strategies, and proves many properties automatically.

The complete expansion of the transition formula in the brute force approach can be particularly expensive if done naively, because PVS's EXPAND performs some simplification after expanding any definition, and a very large number of definitions are being expanded. Because of the sometimes enormous size of the expression being simplified, the extra time required to do these many extra simplifications is noticeable. All definition expansions in the current versions of **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF** are instead done with the "baby step" version of EXPAND.

These current versions also use better management of the size of the transition relation formula, resulting in some dramatic improvements. Expansion of the transition formula now stops after `trans` is expanded. Rather than applying brute force simplification to try to complete the proof, both strategies instead repeatedly use a cycle of substitution of the relatively high-level formulation of the transition relation for occurrences of the new state in the sequent, followed by some simplification, temporary hiding of the transition relation, full expansion of visible update functions, and the simplification step from the original strategies. This change alone resulted in a three-to-four-fold increase in speed, e.g., reducing the execution time of the induction proof of one state invariant of the automaton *CD* of [29] from 7904 seconds to 1768 seconds, and the execution time of the direct proof of one transition invariant of an automaton called *Autopilot* from 65 seconds to 22 seconds. The exact details of how this cycle is managed and how the steps in it are defined differ slightly between **SCR_DIRECT_PROOF** and **SCR_INDUCT_PROOF**, but have gradually been improved using several techniques to minimize the maximum complexity of individual subgoals. These techniques include: 1) several rewrite rules for boolean expressions that force PVS to simplify these expressions "in place" rather than postponing their simpli-

fication until they become top-level formulae, 2) capture and use with a new label of a simplification of the transition formula that (in proofs of most invariants) is eventually computed, and 3) control of the number of substitutions for the new state before expansion of update functions and simplification takes place. Use of these techniques have reduced the time taken by the induction proof of the *CD* property to 324 seconds and reduced the time taken by many direct proofs by as much as 20%.[7] The rewrite rules for boolean expressions are rules that frequently apply when SCR tables are simplified:

$$\texttt{NOT(P) AND P} \Longrightarrow \texttt{FALSE} \tag{6}$$

$$\texttt{P AND NOT(P)} \Longrightarrow \texttt{FALSE} \tag{7}$$

$$\texttt{IF FALSE THEN A ELSE B} \Longrightarrow \texttt{B} \tag{8}$$

Their application often dramatically reduces the size of an expression representing an update function.

Implementing **ANALYZE**. As illustrated in Section 4.2, **ANALYZE** causes PVS to display the complete details of the class of problem transitions corresponding to a dead end reached by **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF**. **ANALYZE** makes heavy use of forward chaining on lemmas from the auxiliary theories to force PVS to display in the antecedent (i.e., as positive assumptions) of the subgoal it produces all the information in the dead end proof goal relevant to the class of problem transitions, including either the actual value of each variable in the current state and next state or the relation between these two values. To accomplish this, forward chaining on many lemmas is sometimes needed. For example, for a state variable `x` of any enumerated type with values `v_1, v_2, ... , v_n`, PVS may display the fact that `x` has value `v_1` in state `s` by displaying the formulae `x(s) = v_2, ... , x(s) = v_n` in the consequent of the sequent. To force `x(s) = v_1` to appear in the antecedent, **ANALYZE** must forward-chain on the lemma

```
NOT(x(s) = v_2) AND ...  AND NOT(x(s) = v_n) => x(s) = v_1.      (9)
```

For the enumerated type `v_1, v_2, ... , v_n`, `n` such lemmas are needed. When the variable `x` takes boolean values, the fact that its value is `FALSE` in state `s` is normally represented in a sequent by simply putting `x(s)` itself as a formula in the consequent. Putting `x(s) = FALSE` in the antecedent is done by forward chaining on the lemma

```
NOT X => X = FALSE.      (10)
```

[7] All proof times are for PVS 2.1 on an UltraSPARC-II. Using PVS 2.3, the induction proof of the *CD* property runs in 286 seconds, and the direct proof of the *Autopilot* property runs in 19 seconds.

However, unconstrained use of this lemma can put many undesired facts in the antecedent. **ANALYZE** uses access to the current proof state to compute the formula numbers of consequent formulae of undesired form, label these formulae, and hide them temporarily before forward chaining on the above lemma is performed. Whether this is more efficient than forward chaining on the set of lemmas

```
NOT x(s) => x(s) = FALSE                    (11)
```

where `x` ranges over all boolean-valued state variables is a question for further study. To compute the values of new state variables and display the results in the antecedent, **ANALYZE** temporarily reveals the transition relation formula, which has the form `s_new` = F`(s_old,s_new)` (see formula (1) in Section 3.2), and uses forward chaining on the lemma

```
s_1 = s_2 => x(s_1) = x(s_2)                (12)
```

for every state variable `x`, followed by a weakened version of SIMPLIFY and any needed computations of update functions. Since SIMPLIFY tends to eliminate as redundant much of the information produced by **ANALYZE**, the order in which the steps in **ANALYZE** are done must be carefully controlled.

Efficiency of **ANALYZE***: a Tradeoff.* Unfortunately, making the strategies **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF** more efficient has required **ANALYZE** to do more work than it once did. The version of **ANALYZE** that went with the original **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF** was more efficient than the current version; the original version analyzed a proof dead end in only a few seconds, but the current version can take a minute or even longer. This is because the brute force method using full expansion of the transition relation essentially precomputes the details of the values of variables in the new state of a transition, which is much of the information **ANALYZE** is intended to display to the user. With the new versions of **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF**, **ANALYZE** frequently must expand several update functions and perform further simplifications that were previously unnecessary. Even though **ANALYZE** has become less efficient, due to the great improvement in the efficiency of **SCR_INDUCT_PROOF** and **SCR_DIRECT_PROOF**, the combined time required by either invariant strategy followed by **ANALYZE** is usually a small fraction of that previously required.

6. Discussion

The techniques used in implementing TAME rely on the PVS features discussed in Section 5. The same techniques could almost certainly be used in any higher-order-logic theorem prover with similar features to implement special-

purpose tools such as TAME, i.e., tools which support both natural, human-style reasoning and automatic reasoning in a particular problem domain. Several features are missing in PVS that would extend the set of techniques available, and allow improvements and extensions to the set of strategies supporting "natural" proof steps in TAME. This section describes some of these missing features and how they could be used to simplify, improve, or extend the TAME strategies. It then discusses the availability of both the existing and missing features in various other theorem provers.

6.1. Supporting Natural High-level Proof Steps in PVS.

Reference [5] lists a number of features missing in PVS that would allow improved or additional high-level proof steps to be implemented as user-defined strategies. Most of these, plus further missing but potentially useful features, are discussed below.

Skolemization and instantiation of embedded quantified formulae. The need to raise an embedded quantified formula to the top level in PVS before it can be skolemized or instantiated often interrupts the flow of reasoning natural to the user, and can result in multiple proof branches where a single branch would suffice. This problem has arisen in several applications, including one mechanized in TAME by Riccobene [7]. A partial solution was achieved by implementing the TAME strategies **SKOLEM_IN** and **INST_IN** (see Appendix B), but while these strategies often work well, they do not always have the desired effect: they sometimes result in multiple subgoals which have identical proofs, and thus require more effort than should be necessary on the part of the user.

Besides being a direct user convenience in interactive proofs, the ability to skolemize and instantiate embedded formulae can also be useful for internal steps in strategies. An important part of the philosophy behind TAME is that the user should be relieved of the need to formulate properties for the convenience of the theorem prover. Thus, **AUTO_INDUCT** should work equally well on the alternative (but equivalent) formulations

```
(EXISTS (r: train): status(r,s) = I)
=> gate_status(s) = fully_down                    (13)
```

and

```
FORALL (r: train):
status(r,s) = I => gate_status(s) = fully_down    (14)
```

of an invariant of state `s` of the automaton `systimpl` from the solution to the Generalized Railroad Crossing problem in [23]. The first formulation more directly expresses the invariant "If there is a train in the intersection, then the gate is fully down." But currently, **AUTO_INDUCT** handles only the second

formulation automatically. Support for skolemization and instantiation of embedded quantifiers would allow the coordinated skolemization and instantiation performed by TAME's strategy **AUTO_INDUCT** to be extended beyond properties (such as (14)) formulated using top-level universal quantifiers to include properties (such as (13)) formulated with embedded quantifiers.

Uniform treatment of subformulae in propositional reasoning. The effects of certain PVS commands are not always uniform. For example, when formulae `A` and `A => B` appear in the antecedent of a subgoal, the command ASSERT is able to deduce `B` if `A` is a ground level formula, but not if `A` is a more complex formula, e.g., a disjunction or a quantified formula. The varying behavior of ASSERT is the result of a decision made for efficiency reasons. However, for the behavior of PVS commands in a user-defined strategy to be more predictable, it would be helpful to have the option of even-handed treatment of all formulae.

Extended arithmetic decision procedures. For hybrid automata in which the updated values of state variables representing real-valued quantities in the environment are computed using nonlinear arithmetic expressions, the PVS arithmetic decision procedures are frequently inadequate to perform the arithmetic reasoning required in the proof of a property. Reference [6] identified several facts about real arithmetic, mostly having to do with the signs of products of quantities of known sign, that were a sufficient supplement to PVS's arithmetic decision procedures to complete most proofs of properties of an example hybrid automaton that required reasoning about nonlinear real arithmetic. While a decision procedure for all nonlinear arithmetic is not possible, an extension to the linear arithmetic decision procedure that takes these properties into account should be possible, and would likely be very helpful in proving properties of hybrid automata.

Parametric polymorphism. The need for parametric polymorphism or some mechanism of equivalent power and convenience has arisen in a few contexts in TAME. One such context is in the use of the Hilbert ϵ operator in specifying and proving invariants of hybrid automata (see Section 3.1). The manner in which the axiom `epsilon_ax` must be used (see Section 5.4) would be much simpler if parametric polymorphism were supported, since the domain type of the predicate to which it is applied would not need to be supplied. A second context in which parametric polymorphism would be helpful is in the general theory of timed executions for LV timed automata. In proofs of properties of timed executions, it can be helpful to use a predicate on state components of arbitrary type expressing the fact that a particular state component is not changed by a time passage action. Parametric polymorphism would make it possible to include the definition of this predicate in the theory of timed executions. As noted in Section 5.3, it is possible to achieve some of the effect of parametric polymorphism in PVS through the use of theories with type parameters. In particular, importing the generic form of a theory with a type parameter into a specification allows the

user to use instances of functions defined in the generic theory in the specification. However, when one wants to apply a lemma from the generic theory in a proof when expressions from multiple theory instances are present, the user must either invoke the correct lemma instance (by supplying the name or names of the type parameters to the generic theory) or else have imported all relevant theory instances into the specification.

Resolution-style steps. First-order resolution is one approach to automatic theorem proving that can be particularly useful for deducing a property by forward reasoning based on unification and a small body of facts. Reasoning of this kind is needed in discharging the proof obligations accompanying some of the experimental TAME proof steps for reasoning about execution sequences. However, there is not yet a good way to automate this reasoning in PVS, and thus relieve the user of tediously proving the "obvious" in these cases.

Stable access to proof state details. Currently, certain details of the proof state are found by deep probing, sometimes of depth twenty or more, into the object structure of the PVS proof state. This object structure is not fully documented, and thus a user wishing to access it to support a strategy must find the details by experiment. There are two sources of instability in the object structure. First, this structure has changed over various PVS versions. Second, attempting the proof of a theorem can actually change the internal representation of the theorem; that is, the first and later proof attempts will be applied to different representations. As a result, certain computations of specification and proof state information done by the TAME strategies have to be performed differently depending on both the PVS version and the PVS session history, even though the basic information required by the TAME strategies remains the same. A documented, uniform way of accessing this information would be very helpful both in designing the code for strategies and in assuring that this code will always produce the intended result.

6.2. Applying Techniques Used in TAME in Other Theorem Provers.

Many of the PVS features used in developing the TAME strategies are available in other higher-order logic theorem provers, as are some of the "missing" features. HOL, Isabelle, and Coq all have extensive strategy languages, and are open systems in which access to the current proof state should be possible. The basic inference rules of these systems provide the type of "baby steps" that were added to PVS to support TAME (and similar applications). These systems all have some form of support for term rewriting, and some decision procedures have been added to HOL and Isabelle. HOL supports a degree of resolution theorem proving, and a simple HOL resolution tactic implements forward-chaining. Because Isabelle can be used to support HOL, the same is true of Isabelle. Isabelle has some support for automated backtracking. All the systems support appli-

cation of lemmas, and because of parametric polymorphism, at least HOL and Isabelle support generic lemmas in somewhat more convenient form than PVS. The Stanford Temporal Prover STeP (see, e.g., [9]), which is not a higher-order system, is an example of a system supporting skolemization and instantiation of embedded quantified formulae.

Because the abilities to hide and then reveal formulae and to label formulae seems so far to be unique to PVS, it would be difficult if not impossible to implement steps in other systems that do precisely the work of the TAME steps that rely heavily on these features for their behavior. It is probable that the addition of formula labels in other systems would simplify definitions of tactics and strategies in these systems, and the further addition of the ability to temporarily hide and then reveal formulae, used in conjunction with labels, would allow improvements in execution efficiency of their tactics or strategies in analogy to the way they support execution efficiency in TAME.

7. Related Work

Strategies in theorem proving have two major purposes. One is to allow propositions to be proved automatically; the other is to provide proof steps that advance an interactive proof while saving the user from explicitly applying many fine-grained steps. TAME uses strategies for both purposes. Proof planning, which has been widely used and documented (see, e.g., [45]), is prominent among approaches to creating tactics for complete proofs—as was done for HOL in [10]—as well as partial proof steps. Such tactics are automatically developed from a proof plan. To support such automatic proof development, proof plans are specified formally and in considerable detail in a special language. An analogy can be drawn between the use of TAME and the use of proof planning: the general TAME strategies can be viewed as simplifying the translation by a user, in the limited domain of automata models, of a "proof plan" in the form of a hand proof of a property into a mechanized proof of the property. The essential part of the plan provided by a hand proof is its content rather than its form; TAME has been used to translate both Lamport-style hand proofs (see Section 4.1), and natural language hand proofs (see, e.g., [4,6]). Of course, TAME does not mechanize the translation, and is limited to theorem proving in one application domain.

Much work, of which [11,12,42] is only a very small sample, has been done to develop strategies or tactics to support high-level proof steps rather than to accomplish whole proofs. The advances in a proof to which these high-level steps correspond may or may not match high-level steps in human reasoning, as those in TAME attempt to do. The use of proof planning for normalization in [11] supports a type of proof step—normalization of an expression—that represents an identifiable high-level step meaningful in a human-style proof. By contrast, several of the proof steps (implemented in Isabelle) provided in DOVE [12] simply manipulate the current proof goal to prepare it for the application of further proof

steps. Reference [42] describes a general framework for proving properties of I/O automata of all kinds. The extent of the resemblance of the Isabelle tactics of [42] for proving invariant properties of I/O automata and the TAME strategies for invariants has yet to be studied.

There are several tools that support reasoning about automata models, including [19,9,12,42]. The tools described in [19] (based on Larch) and in [42] (based on Isabelle) support reasoning about I/O automata, while the tool DOVE described in [12] (based on Isabelle) and the tool STeP described in [9] support reasoning about automata represented as transition systems. The Larch prover LP does not support strategies, and strategy support in STeP is very minimal; both systems expect users to establish properties using a fixed set of steps, some of which invoke decision procedures. Both DOVE and the I/O automata environment of [42] use Isabelle tactics to support reasoning about automata. As noted above, the environment of [42] supports proofs of many kinds of I/O automata properties. The TAME work has instead concentrated on the problem of making theorem proving practical in developing systems modeled as automata, and on the most immediate need of most developers—the ability to establish invariant properties of their designs. How the tactics of [12] and [42] compare to the TAME strategies has been discussed above. None of the tools in [9,12,42] save proofs, and those saved by the Larch-based tool [19] bear no clear relation to hand proofs.

There is a body of work related to the use of comments and labels in theorem proving. Two examples are [32] and [27], which discuss the use of *proof annotations*. Both [32] and [27] describe the use of annotations guide automatic proof search. The annotations of [32] are entered by the user, while some annotations in [27] can be computed from other annotations. The closest analogue of such annotations in TAME is labels; though comments are automatically computed in TAME, they are provided as explanation to the user rather than for guiding a proof. TAME assigns labels automatically, and computes some of them such as the labels on formulae introduced through a lemma. As with comments, labels provide a degree of explanation to the user; however, TAME also uses them to guide the proof. Labels in TAME do not provide nearly the amount of information that can be provided by the annotations described in [32] and [27], and thus cannot be used as extensively for proof guidance. Nevertheless, they have been very valuable for this purpose in TAME.

8. Conclusion

The TAME strategies are designed to make the verification of automata models more practical in software development. To accomplish this, they implement theorem prover steps that mimic the high-level proof steps natural for humans, and for one automata model—SCR automata—provide a step that automatically proves many invariant properties. The TAME proof steps not only make it possible to use a theorem prover (PVS) to prove many properties of soft-

ware specifications and designs without expert knowledge of the theorem prover, but reduce the effort required even for an expert. Moreover, use of the natural proof steps in TAME results in saved PVS proofs whose significance is clear, because their structure and proof steps closely resemble those found in many natural language proofs. Indeed, TAME proofs can be automatically translated into natural language.

The TAME strategies illustrate several useful purposes for strategies beyond the mere implementation of steps for executing part or all of a proof. These purposes include 1) support for a "natural" interactive proof style, 2) meaningful structure in saved proofs, and 3) improved user feedback. One benefit of the meaningful structure in saved proofs is that these proofs can provide information on *why* a property holds: a property holding for unexpected reasons can indicate an error in the specification of an automaton. Improved user feedback includes providing understandable saved proofs, using labels to explain the content of the sequent in a proof goal, and supplying special proof steps such as **ANALYZE** that transform the sequent in a proof dead end into a form that is clearer and more informative to the user.

The current form of the TAME strategies would not be possible without the recent addition of new PVS features and documentation. Documentation of how to access the current proof state and other global variables maintained by PVS has been vital to making the TAME proof steps generic. The addition of support for labels and comments has permitted TAME to make both sequents and saved proofs more understandable. Support for several new "baby steps", along with labels, have permitted TAME steps to be more predictable in effect, user-friendly, and efficient.

Among the lessons learned in developing the TAME strategies is the importance of the structure of specifications to the development of generic and efficient strategies. The generic TAME proof steps depend on the template structure followed by automaton specifications. The efficiency of the TAME proof steps, particularly the automatic strategies for SCR automata, depends on certain details of the structure of definitions. A second lesson learned is that analyses of specifications performed by other tools can improve the efficiency of the strategies used by a theorem prover to prove properties of these specifications. In TAME, as noted in Sections 3.2 and 5.5, such analyses of an SCR specification are used to determine the most "efficient" definition structures that can be used to represent the SCR specification in the theorem prover. In addition, invariants of a specification produced by an automatic invariant generator have the potential to extend the number of invariants that can be proved by an automatic strategy.

Acknowledgements

I wish to thank Ralph Jeffords for many helpful discussions concerning proofs of invariants of SCR automata, Steve Sims and James Kirby for implementing

the prototype SCR-to-TAME and ANALYZE output translators, and Elvinia Riccobene for the TAME proofs for *TIP* used as examples. I also thank Ramesh Bharadwaj, Constance Heitmeyer, Ralph Jeffords, James Kirby, and Elizabeth Leonard, as well as the anonymous referees, for many insightful comments that have helped to improve this paper. Finally, I thank Natarajan Shankar and Sam Owre of SRI International for providing requested new features in PVS.

References

[1] A. Alborghetti, A. Gargantini, and A. Morzenti. Providing automated support to deductive analysis of time critical systems. In *Proc. 6th European Software Engineering Conference (ESEC/FSE'97)*, volume 1301 of *Lect. Notes in Comp. Sci.*, pages 211–226. Springer-Verlag, 1997.

[2] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A7-E aircraft. Technical Report NRL-9194, Naval Research Lab., Wash., DC, 1992.

[3] M. Archer. Tools for simplifying proofs of properties of timed automata: The TAME template, theories, and strategies. Technical Report NRL/MR/5540–99-8359, NRL, Wash., DC, 1999.

[4] 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)*, pages 192–203. IEEE Computer Society Press, 1996.

[5] M. Archer and C. 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.

[6] M. Archer and C. 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.

[7] M. Archer, C. Heitmeyer, and E. Riccobene. Using TAME to prove invariants of automata models: Case studies. In *Proc. 2000 ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP'00)*, August 2000.

[8] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers 1998 (UITP '98)*, Eindhoven, Netherlands, July 1998.

[9] N. Bjorner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. In *Proceedings of ARTS'97*, volume 1231 of *Lect. Notes in Comp. Sci.*, pages 22–43. Springer-Verlag, May 1997.

[10] R. Boulton, A. Bundy, K. Slind, and M. Gordon. An interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lect. Notes in Comp. Sci.*, pages 67–86. Springer-Verlag, 1998.

[11] A. Bundy. The use of proof plans for normalization. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 7 of *Automated Reasoning Series*, pages 149–166. Kluwer, 1991.

[12] T. Cant, K. Eastaughffe, J. Grundy, M. Ozols, and et al. Dove user manual. Trusted Computer Systems Group, Defence Science and Technology Organisation, Salisbury, Australia, October 31, 1998.

[13] O. Cheiner. Carnegie-Mellon University, Private communication. February, 1999.

[14] M. Devillers. Verification of a tree-identity protocol. URL `http://www.cs.kun.nl/~marcod/1394.html`, 1997.

[15] M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.

[16] S. Easterbrook and J. Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 1997.

[17] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, Sept. 1992.

[18] 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.

[19] S. J. Garland and N. A. Lynch. The IOA language and toolset: Support for mathematics-based distributed programming. Submitted for publication.

[20] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS'95)*, Gaithersburg, MD, June 1995. IEEE Computer Society Press.

[21] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11):927–948, Nov. 1998.

[22] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *10th Intl. Conf. on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 526–531. Springer-Verlag, 1998.

[23] 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, Dec. 1994.

[24] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-51, Lab. for Comp. Sci., MIT, Cambridge, MA, 1994. Also Technical Report 7619, NRL, Wash., DC 1994.

[25] 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.

[26] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[27] D. Hutter. Annotated reasoning. In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Proceedings of the FLoC'99 Workshop on Strategies in Automated Deduction (STRATEGIES'99), Trento, Italy*, pages 37–50, July 1999.

[28] James Kirby, Jr., M. Archer, and C. Heitmeyer. Applying formal methods to an information security device: An experience report. In *Proc. 4th IEEE International Symposium on High Assurance Systems Engineering (HASE '99)*. IEEE Comp. Soc. Press, November 1999.

[29] James Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proc. 15th Annual Computer Security Applications Conference (ACSAC '99)*. IEEE Comp. Soc. Press, December 1999.

[30] R. Jeffords. Private communication. NRL, 1998.

[31] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, FL, November 1998.

[32] S. Kalvala. Annotations in formal specifications and proofs. *Formal Methods in System Design*, 5(1/2), 1994.

[33] P. KelloMaki. *Mechanical Verification of Invariant Properties of DisCo Specifications.* PhD thesis, Tampere University of Technology, Finland, November 1997.

[34] L. Lamport. How to write a proof. Technical report, Digital Equipment Corp., System Research Center, February 1993. Research Report 94.

[35] G. Leeb and N. Lynch. Proving safety properties of the Steam Boiler Controller: Formal methods for industrial applications: A case study. In J.-R. Abrial, et al., eds., *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, vol. 1165 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1996.

[36] P. Lincoln. Private communication. July, 1998.

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

[38] R. R. Lutz and H.-Y. Shaw. Applying the SCR* requirements toolset to DS-1 fault protection. Technical Report JPL-D15198, Jet Propulsion Lab., Pasadena, CA, Dec. 1997.

[39] 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.

[40] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[41] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[42] O. Mueller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universitaet Muenchen, September 1998.

[43] D. L. Parnas, G. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April–June 1991.

[44] E. Riccobene, M. Archer, and C. Heitmeyer. Applying TAME to I/O automata: A user's perspective. Technical Report NRL/MR/5540–00-8448, NRL, Wash., DC, 2000.

[45] J. Richardson and A. Bundy. Proof planning methods as schemas. *Journal of Symbolic Computation*, 11, 1999.

[46] J. Romijn. Tackling the RPC-Memory Specification Problem with I/O automata. Addendum. URL `http://www.cwi.nl/~judi/papers/ dagstuhl_proofs.ps.gz`.

[47] J. Romijn. Tackling the RPC-Memory Specification Problem with I/O automata. In M. Broy, S. Merz, and K. Spies, editors, *Formal Systems Specification — The RPC-Memory Specification Case*, volume 1169 of *Lect. Notes in Comp. Sci.*, pages 437–476. Springer-Verlag, 1996.

[48] 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.

[49] 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.

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

Appendix

A. The I/O Automaton *TIP*

Below is the specification of the I/O automaton *TIP* from [15]. It first lists the actions, classifying them as to whether they are input, output, or internal actions (*TIP* has no input actions). It then lists the state variables, giving their types, and describes the set of initial states. Finally, it describes all the actions in terms of their preconditions and effects.

Internal: *ADD_CHILD, CHILDREN_KNOWN, RESOLVE_CONTENTION, ACK*
Output: *ROOT*
State Variables: $init : \mathbf{V} \rightarrow \mathbf{Bool}$
$contention : \mathbf{V} \rightarrow \mathbf{Bool}$
$root : \mathbf{V} \rightarrow \mathbf{Bool}$
$child : \mathbf{E} \rightarrow \mathbf{Bool}$
$mq : \mathbf{E} \rightarrow \mathbf{Bool}*$

Init: $\forall v, e : init[v]$
$\wedge \neg contention[v]$
$\wedge \neg root[v]$
$\wedge \neg child[e]$
$\wedge mq[e] = \mathsf{empty}$

Actions:

$ADD\_CHILD(e : \mathbf{E})$
Precondition :
$\wedge init[\mathsf{target}(e)]$
$\wedge mq[e] \neq \mathsf{empty}$
Effect :
$child[e] := 1$
$mq[e] := \mathsf{tl}(mq[e])$

$ACK(e : \mathbf{E})$
Precondition :
$\wedge \neg init[\mathsf{target}(e)]$
$\wedge mq(e) \neq \mathsf{empty}$
Effect :
$contention[\mathsf{target}(e)] := \neg \mathsf{hd}(mq[e])$
$mq[e] := \mathsf{tl}(mq[e])$

$RESOLVE\_CONTENTION(e : \mathbf{E})$
Precondition :
$\wedge contention[\mathsf{source}(e)]$
$\wedge contention[\mathsf{target}(e)]$
Effect :
$child[e] := 1$
$contention[\mathsf{source}(e)] := 0$
$contention[\mathsf{target}(e)] := 0$

$ROOT(v : \mathbf{V})$
Precondition :
$\wedge \neg init[v]$
$\wedge \neg contention[v]$
$\wedge \neg root[v]$
$\wedge \forall e \in \mathsf{to}(v) : child[e]$
Effect :
$root[v] := 1$

$CHILDREN\_KNOWN(v : \mathbf{V})$
Precondition :
$\wedge init[v]$
$\wedge \forall e, f \in \mathsf{to}(v) : child[e] \vee child[f] \vee e = f$
Effect :
$init[v] := 0$
for $e \in \mathsf{from}(v)$ **do** $mq[e] := \mathsf{append}(child[e^{-1}], mq[e])$

The type **E** in the above specification corresponds to the type Edges in the TAME formulations of Invariants I_{10} and I_{13}. The type $\mathbf{Bool}*$ in the specification is lists of booleans; thus, the variable mq maps an edge to a list of booleans.

The properties from [15] referenced in the lemmas and proofs in Section 4.1 are shown below. In TAME, representing the value of a state variable in a given state is done by using the state as an additional argument to the variable. Thus, $mq[e]$ in the properties below is represented as `mq(e,s)` in TAME. Further, empty, hd, and e^{-1} are represented as `null`, `car`, and `reverse_edge(e)` in TAME.

Invariant I_2. *If a node is in the initial stage then its outgoing links are empty.*
$I_2(e) \equiv init[\mathsf{source}(e)] \rightarrow mq[e] = \mathsf{empty}$

Invariant I_5. *Each link contains at most one message at a time.*
$I_5(e) \equiv \mathsf{length}(mq[e]) \leq 1$

Invariant I_8. *If a node is involved in root contention, then all its incoming links are empty.*
$I_8(e) \equiv contention[\mathsf{target}(e)] \rightarrow mq[e] = \mathsf{empty}$

Invariant I_{10}. *A node never sends a parent request to its children.*
$I_{10}(e) \equiv mq[e] \neq \mathsf{empty} \wedge \neg \mathsf{hd}(mq[e]) \rightarrow \neg child[e^{-1}]$

Invariant I_{13}. *All the incoming links of a root node are child links.*
$I_{13}(e) \equiv root[\mathsf{target}(e)] \rightarrow child[e]$

B. TAME Strategies

TAME currently provides more than twenty proof steps for proving properties of LV timed automata, I/O automata, and SCR automata through PVS strategies. Figure 13 lists the major TAME strategies and describes, for each strategy, the proof step it implements and when this proof step is appropriate in a proof.

Proof Step	TAME Strategy	Remarks
Break down into base case and induction (i.e., action) cases	**AUTO_INDUCT**	For starting an induction proof; only nontrivial cases produce subgoals
Appeal to precondition of an action	**APPLY_SPECIFIC_PRECOND**	Used, when needed, in induction cases
Apply the inductive hypothesis to values other than skolem constants of inductive conclusion	**APPLY_IND_HYP**	Used, when needed, in induction cases; needs arguments
Perform the usual first steps of a non-induction proof	**DIRECT_PROOF**	For starting a direct proof of an invariant
Apply an auxiliary invariant lemma	**APPLY_INV_LEMMA**	Used in any proof; needs argument(s)
Break down into cases based on a predicate	**SUPPOSE**	Used in any proof; needs boolean argument
Apply "obvious" reasoning, e.g., propositional, equational, datatype	**TRY_SIMP**	Used for "it is now obvious " in any proof
Use a fact from the mathematical theory for a state variable type	**APPLY_LEMMA**	Used in any proof; needs argument(s)
Introduce the relationships among specification constants	**CONST_FACTS**	Introduces the facts from the axiom **`const_facts`**
Attempt to skolemize a quantified formula ''in place''	**SKOLEM_IN**	Used in any proof; needs arguments
Attempt to instantiate a quantified formula ''in place''	**INST_IN**	Used in any proof; needs arguments
Apply the Hilbert **ε** axiom to a predicate	**USE_EPSILON**	For proofs referring to **ε** expressions
Instantiate the existentially quantified formula created by **USE_EPSILON**	**EPSILON_WITNESS**	For the companion proof branch created by **USE_EPSILON**
Attempt to prove a state or transition invariant automatically	**SCR_INV_PROOF**	Specialized for SCR automata; tries using generated invariants to finish proofs
Attempt to prove a state invariant automatically	**SCR_INDUCT_PROOF**	Specialized for SCR automata
Attempt to prove a transition invariant automatically	**SCR_DIRECT_PROOF**	Specialized for SCR automata
Display details of the state transition in a proof dead-end	**ANALYZE**	Specialized for SCR automata
Introduce and name the last event **π** before state-occurence **s** satisfying property **P(π,s)**	**NAME_LAST_EVENT**	For timed executions; needs arguments; creates existence proof obligation
Introduce and name the first event **π** before state-occurence **s** satisfying property **P(π,s)**	**NAME_FIRST_EVENT**	For timed executions; needs arguments; creates existence proof obligation

Figure 13. The major TAME strategies.

C. Some PVS Commands

A sample of major PVS strategy-building commands is shown in Figure 14. Figure 15 shows those standard PVS proof commands referred to in this paper.

PVS Command	Arguments	Effect
APPLY	A strategy	Turns a strategy into a defined rule
THEN	A list of proof commands	Applies the proof commands in order down all proof branches
TRY	Three proof commands	Tries the first proof command; if it succeeds, TRY applies the second; if not, TRY applies the third.
IF	A condition and two proof commands	Evaluates the condition; if the result is **`true`**, IF applies the first command; if **`false`**, IF applies the second
BRANCH	A proof command and a list of proof commands	Applies the command and spreads application of the list of commands over the new subgoals; the last command in the list is repeated on any extra subgoals
REPEAT	A proof command	Iterates the command down the main proof branch
REPEAT*	A proof command	Iterates the command down all proof branches
WITH-LABELS	A proof command and a list of lists of labels	Applies the command, and labels the new formulae in each generated subgoal with its corresponding list of labels

Figure 14. Strategy-building commands ("strategicals") in PVS.

PVS Command	Arguments	Effect
SIMPLIFY	Optional arguments control kinds and location of simplification	Applies Shostak decision procedures and and any additional simplifications requested
ASSERT	A subset of the arguments to SIMPLIFY	SIMPLIFY, with rewriting and recording of facts for future use
INST	Formula number and instantiations	Instantiates antecedent universal and consequent existental formulae
EXPAND	function name and (optional) formula number(s)	Expands a function definition and may do simplifications
FLATTEN	Formula number(s) (optional)	Separates antecedent conjunctions and consequent disjunctions or implications into distinct formulae
SPLIT	Formula numbers(s) (optional)	Separates antecedent disjunctions and implications and consequent conjunctions, yielding new subgoals
PROP	None	Applies FLATTEN and SPLIT to the extent possible
LIFT-IF	Formula number(s) (optional)	Lifts an embedded IF_THEN_ELSE to the top level
FAIL	None	Causes the current proof branch to fail
GRIND	None	Undertake an automatic proof

Figure 15. PVS proof commands referred to in this paper.