Automated Software Engineering, 9, 201–232, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Proving Invariants of I/O Automata with TAME*

MYLA ARCHER archer@itd.nrl.navy.mil
CONSTANCE HEITMEYER heitmeyer@itd.nrl.navy.mil
Naval Research Laboratory, Code 5546, Washington, DC 20375, USA

ELVINIA RICCOBENE riccobene@dmi.unict.it
Dipartimento di Matematica e Informatica, Università di Catania, Viale A. Doria 6, I-95125 Catania, Italy

Abstract. This paper describes a specialized interface to PVS called TAME (Timed Automata Modeling Environment) which provides automated support for proving properties of I/O automata. A major goal of TAME is to allow a software developer to use PVS to specify and prove properties of an I/O automaton efficiently and without first becoming a PVS expert. To accomplish this goal, TAME provides a template that the user completes to specify an I/O automaton and a set of proof steps natural for humans to use for proving properties of automata. Each proof step is implemented by a PVS strategy and possibly some auxiliary theories that support that strategy. We have used the results of two recent formal methods studies as a basis for two case studies to evaluate TAME. In the first formal methods study, Romijn used I/O automata to specify and verify memory and remote procedure call components of a concurrent system. In the second formal methods study, Devillers et al. specified a tree identify protocol (TIP), part of the IEEE 1394 bus protocol, and provided hand proofs of TIP properties. Devillers also used PVS to specify TIP and to check proofs of TIP properties. In our first case study, the third author, a new TAME user with no previous PVS experience, used TAME to create PVS specifications of the I/O automata formulated by Romijn and Devillers et al. and to check their hand proofs. In our second case study, the TAME approach to verification was compared with an alternate approach by Devillers which uses PVS directly.

Keywords: software engineering, software requirements analysis, formal methods, proof checking, verification, theorem proving

1. Introduction

A number of authors (Miller and Srivas, 1995; Butler et al., 1995; Crow and Di Vito, 1996; Butler, 1996) have found that the PVS specification language (Owre et al., 1999) and similar strongly typed, higher-order logic languages are well suited to the formalization of system specifications. All report that practitioners, such as hardware and software designers and requirements analysts, can understand appropriately structured PVS specifications. Further, Miller states that engineers at Collins Aviation learned to use PVS to prove properties of a microprocessor design (Miller and Srivas, 1995).

Yet, significant barriers exist to widespread industrial use of mechanical provers such as PVS in software practice. In fact, Miller and Srivas (1995), Butler et al. (1995), and Crow and Di Vito (1996, 1998) all concede that in general practitioners are unwilling or unable to create formal specifications or to perform analysis of the specifications using the PVS proof checker. Further, Butler et al. (1995) observe that one barrier to transferring formal

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

methods technology to industry is that "Training the industrial experts to use the formal techniques, especially to develop skill in verification, [is costly]."

The high cost of training practitioners is not the sole barrier to more general use of PVS (and other theorem provers) to formalize and prove properties of specifications. First, the user must construct an appropriate representation of the system of interest in the language of the theorem prover. Formal proofs of system properties will be influenced by the details of this representation, the manner in which properties are formalized, and the set of reasoning steps available in the theorem prover. The effort required is therefore substantial even for those who have mastered the prover. Moreover, direct use of a general-purpose theorem prover can encourage users to adapt both specifications and proofs to the needs of the prover rather than to the standard conventions of the problem domain. As a result, practitioners often find it difficult to relate the formal representation and formal proofs of properties to their intuitive notions of the system behavior and its properties. We agree with Butler et al. (1995) who argue that:

> [T]he formal methods researchers must be willing to adapt their methods to the problem domain rather than fight to change the existing methodologies to conform to their needs.

TAME (Timed Automata Modeling Environment) (Archer and Heitmeyer, 1996, 1997b; Archer et al., 1998, 2000; Archer, 2000) is a specialized interface to PVS designed to reduce the barriers to more general use of theorem proving for verifying automata models. With TAME, users can create PVS specifications of three different automata models: Lynch-Vaandrager (LV) timed automata (Lynch and Vaandrager, 1996), I/O automata (Lynch and Tuttle, 1989), and the automata model that underlies specifications in the SCR (Software Cost Reduction) tabular notation (Heitmeyer et al., 1998). Users can also formulate properties of automata in standard logic and prove these properties mechanically using reasoning "natural" to humans. A major goal of TAME is to provide explicit support for specifying and proving properties of automata. To make it easy to create PVS specifications from automaton descriptions, TAME provides templates for specifying the behavior of automata. To simplify the development of mechanized proofs, TAME provides a set of PVS strategies supporting the kinds of proof steps normally found in hand proofs of invariant properties of automata. In particular, TAME is designed to support straightforward construction of a mechanized proof of an invariant property from a high-level hand proof that expresses a domain expert's explanation of why the property holds. Further, TAME is designed to produce saved proofs that can be easily translated into natural language proofs (i.e., explanations).

In two recent formal methods studies, two example systems were specified and verified using the I/O automata model. In the first study, Romijn specified and verified a solution to the RPC-Memory (Remote Procedure Call) problem, a problem posed by Broy and Lamport at the 1994 Dagstuhl Seminar on Reactive Systems (Romijn, 1996). In the second study, Devillers et al. specified and verified an I/O automaton called *TIP* (Tree Identify Protocol), the leader election algorithm of the IEEE 1394 multimedia bus protocol. These studies, together with a volunteer TAME user with no previous PVS experience (the third author), gave us the opportunity to evaluate TAME. Both Romijn (1996) and Devillers et al. (2000) provide specifications of I/O automata and their properties. In addition, they provide hand proofs (Romijn, 1996; Devillers, 1997) of most of the properties in Romijn (1996)

and Devillers et al. (2000) in varying degrees of detail. Thus, the third author had several example specifications and proofs to which she could apply TAME. In addition, Devillers et al. (2000) describe the PVS specification and proofs of the I/O automaton *TIP* and provide a pointer to the specification and proofs on the web. This gave us the opportunity to compare the use of PVS through TAME with the direct use of PVS by Devillers et al.

Section 2 of this paper briefly reviews I/O automata and then provides an overview of the *TIP* and RPC-Memory examples. Section 3 provides a high-level description of TAME, its templates, and its strategies. Section 4 compares a direct PVS approach to specification and proof for *TIP* (Devillers et al., 2000) to the TAME approach to the same problem. Although this study of *TIP* was done after the study of *TIP* discussed in Section 5, we describe it first because it clarifies the difference between using PVS through TAME and using PVS directly. Section 5 describes the experience of the third author in applying TAME to *TIP* and the RPC-Memory Problem (Romijn, 1996; Devillers et al., 2000; Devillers, 1997), with particular attention to (1) the time and effort required and (2) the adequacy of the TAME proof steps for mechanizing the proofs of invariants. Section 6 discusses the results described in Sections 4 and 5 and some improvements in TAME that resulted from the case study described in Section 5. Section 6 also provides an example of the automatic translation of a TAME proof into a natural language explanation. Finally, Section 7 describes related work, and Section 8 presents our conclusions.

2. The I/O automata model: Two examples

The I/O automata model is a very general model for (possibly infinite) discrete state transition systems. Lynch and Tuttle (1989) define an I/O automaton as a set of *states* with a subset of *initial states*, a set of *actions*, and a *transition relation*. A state is an assignment of values to state variables. The transition relation is usually described in terms of the "precondition" and "effect" of each action. The actions are partitioned into *input*, *output*, and *internal* actions to control the way I/O automata can be composed: an output action of one automaton may be composed with compatible input actions of one or more other automata; no other composition of actions is permitted. No composition of automata occurs in the *TIP* example, but composition is used in the RPC-Memory example to connect a remote procedure call component with a memory component.

The *TIP* study (Devillers et al., 2000) produced a specification and proofs for the leader election algorithm, the core of the tree identify phase of the physical layer of the IEEE 1394 bus protocol. The tree identify protocol is applied to a special kind of graph, namely, an *undirected digraph* (if the graph contains an edge e, then it also contains its reverse edge *reverse(e)*) without self-loops. Moreover, the graph topology must be tree-like, meaning that for each pair of vertices v, w in the graph, there is a unique sequence of vertices $v_0, v_1, \ldots, v_n$ such that (1) $v_0 = v$; (2) $v_n = w$; (3) for all $0 \leq i \leq n-1, (v_i, v_{i+1}) \in$ `Edges`; and (4) no vertex occurs more than once in the sequence. The algorithm identifies a spanning tree of the digraph. As the algorithm proceeds, particular links (directed edges) between adjacent nodes are added to a directed spanning tree. At any point during execution of the algorithm, those edges that have been added to the spanning tree are known as *child* edges. The algorithm terminates when the spanning tree is complete; its root is then the

"leader". The goal of the analysis by Devillers et al. (2000) is to establish the property: "For an arbitrary tree topology, exactly one leader is elected." This property is proved in Devillers (1997) with the aid of 17 invariants.

Appendix A contains the specification of *TIP* in terms of an I/O automaton (Devillers et al., 2000). The first two lines of this specification declare the four internal actions and the single output action (there are no input actions). The next five lines declare the state variables and define the initial state. All five state variables of *TIP* have a complex (function) type rather than a simple type. For example, the declaration that *init* has type $\mathbf{V} \rightarrow \mathbf{Bool}$ says that the state variable *init* is a function from **V** to **Bool**. The types **V** and **E**, which represent the vertices and edges of the graph, were defined outside of the specification. The type $\mathbf{Bool}*$ represents lists of booleans with associated operations hd, tl, empty, and append. The remainder of the specification specifies the parameters, preconditions, and effects of the five actions. Appendix B contains the TAME representation of the *TIP* specification. Appendix C provides a sample of the 17 *TIP* invariants from Devillers et al. (2000) and their representations in TAME. Section 5 describes how the third author constructed the TAME translation for *TIP*.

The RPC-Memory problem concerns the interaction of a memory component and a remote procedure call (RPC) component of a distributed system. Romijn's solution (Romijn, 1996) contains approximately twenty I/O automata and hand proofs of many kinds of properties—relative safety, liveness, deadlock-freeness, properties of quiescent states, implementation (based on weak simulation or weak refinement properties), and state invariants. Since the third author's goal was to automate these proofs, she focused on three automata for which invariants were proved: *Memory**, which models one version of the memory; *MemoryImp*, which models the combination of a "reliable" version of the memory with the RPC; and *Imp*, which models an implementation of a lossy version of the RPC, with timing information added.[1]

Nearly all of the hand proofs of the state invariants for both the *TIP* and RPC-Memory examples are in the Lamport style (Lamport, 1993). The TAME specifications and proofs for both examples can be found at the URL http://chacs.nrl.navy.mil/projects/tame.

3. TAME

In proving properties of automata models, model checking (Clarke et al., 1986) is often viewed as more practical and easier to use than theorem proving. While clearly an important technique for detecting certain classes of errors in specifications, model checking has some limitations when compared to theorem proving. For example, it is often claimed that model checking is automatic and that it requires little expertise from the user. However, due to the state explosion problem, the user usually model checks an abstract model of a given system rather than the complete system specification. Finding the appropriate abstract model often relies on user ingenuity and creativity. Even when abstraction is used, the state explosion problem may prevent the model checker from running to completion. Hence, model checking may be unable to establish the correctness of a property. Moreover, when system models contain parameters, model checking can only check correctness for specific (usually small), rather than arbitrary, parameter values. The protocols from Romijn (1996) and Devillers et al.

(2000) contain parameters as well as another feature—complex data types. Both features are problematic for a model checker. For the verification of these and similar examples, theorem proving is more effective.

However, direct use of a mechanical theorem prover to establish a selected property, even a simple property, typically requires significant and tedious human effort. See, for example, the PVS proof in figure 5 in Section 4. To reduce the human effort needed to establish properties of automata using theorem proving, TAME provides a special theorem prover interface (Archer and Heitmeyer, 1996, 1997b; Archer et al., 1998, 2000; Archer, 2000). This interface consists of a set of templates for specifying automata, a set of standard PVS theories, and a set of standard proof steps, with each proof step implemented as a PVS strategy. A PVS theory is an object, sometimes parameterized, containing a set of related definitions and theorems. A PVS strategy is a pattern of inference steps that can adapt to different circumstances by using conditionals and backtracking.

Several considerations led to our decision to base TAME on PVS rather than a different theorem prover:

- PVS has a relatively user-friendly interface and supports a standard logic that allows users to formalize properties to be proved in a natural way.
- The PVS decision procedures handle many of the low-level details in proofs efficiently.
- PVS saves rerunnable proof scripts and thus provides the basis for creating human readable proofs.
- PVS has a rich type system.

The rich type system supported by PVS has one drawback: it makes type checking undecidable. As a result, the type checker may generate type correctness conditions (TCCs) that must be proved before any proofs of properties are considered valid. TCCs may also occur as extra subgoals in PVS proofs. Unprovable TCCs may be the result of specification errors; these can be eliminated by correcting the specification. Any remaining TCCs can often be proved by PVS's automatic theorem proving strategy GRIND. The user occasionally must supply more proof detail. For the examples in this paper, example elements of certain sets had to be supplied to prove some of the TCCs generated by PVS.

Below, we describe TAME's templates, theories, and strategies, and how they are related. We also discuss the major goals which have guided the design of the TAME strategies.

3.1. TAME templates

As stated in Section 1, TAME currently provides templates for three classes of automata: LV timed automata, I/O automata, and SCR automata. Each template provides a standard structure for defining an automaton. Because LV timed automata are essentially I/O automata with time added, the template originally designed for specifying LV timed automata was easily adapted to specifying I/O automata by assigning standard default definitions to time-related quantities. To define an automaton of either of these two classes, the user provides the information shown in figure 1.

Template Part	User Fills In	Remarks
`actions`	Declarations of non-time-passage actions	The nondefault cases of the `actions` datatype
`MMTstates`	Type of the "basic state" representing the state variables	Usually a record type
`OKstate?`	A state predicate constraining the set of states	Default is `true`
`enabled_specific`	Preconditions for all the non-time-passage actions	`enabled_specific(a,s)` = specific precondition of action `a` in state `s`
`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	Default is `true`

Figure 1. Information in the TAME template.

As illustrated in Section 5, specifications of I/O automata in the style used by Devillers et al. (2000) and Romijn (1996) can easily be translated into TAME specifications. Although TAME does not provide automated support for composing automata or reasoning directly about an automaton defined as a composition, when an I/O automaton is defined as the composition of two or more other I/O automata, the user can combine the information extracted from the individual automaton descriptions to produce a single TAME specification in a (usually) straightforward way. For an indication of how the composition procedure for I/O automata specifications may sometimes require human insight, see the discussion in Section 5 of composition in the RPC-Memory Problem (Romijn, 1996).

3.2. *TAME proof steps*

TAME's standard strategies are designed to support mechanical reasoning about automata using proof steps that mimic human proof steps. These strategies are based on type and name conventions enforced by the templates, the TAME standard theories, and additional special definitions, auxiliary local theories, and local strategies that can be generated from a particular template instantiation. For example, lemmas in a standard theory called `machine` support the (automaton) induction strategy **AUTO_INDUCT** (see figure 2). The auxiliary local theories contain lemmas which support the rewriting and forward chaining needed in "obvious" reasoning about a particular application.

The TAME user strategies implement proof steps typically used in hand proofs of automaton properties. Hand proofs of invariant properties typically contain only proof steps from a limited set. Figure 2 lists the most common proof steps used in invariant proofs and identifies the TAME strategies that implement them. TAME strategies also exist for several

Proof Step	TAME Strategy	Remarks
Break down into base case and induction (i.e., action) cases, and do standard first steps of each case	**AUTO_INDUCT**	For starting an induction proof
Appeal to precondition of an action	**APPLY_SPECIFIC_PRECOND**	Used when needed in induction cases
Apply the inductive hypothesis to argument(s) other than the (default) skolem constant(s)	**APPLY_IND_HYP**	Used when needed in induction cases; needs argument(s)
Perform the standard initial steps in the direct proof of an invariant	**DIRECT_PROOF**	For starting a non-induction proof
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)

Figure 2. Common steps for invariant proofs and their TAME strategies.[2]

steps needed less frequently than those listed in figure 2. For more information about the TAME user strategies, their effects, and how they are implemented, see Archer (2000).

3.3. Major goals of the TAME proof steps

One major goal of the TAME proof steps is to save the user from the tedium typical of proofs done directly in PVS. One technique for achieving this, used in almost all of the TAME strategies, is to incorporate repeated patterns of steps into high-level steps. For example, the TAME strategy **AUTO_INDUCT** incorporates several repeated patterns into high-level steps. In some cases, a repeated pattern becomes a single proof step: for example, given the appropriate arguments, the TAME strategy **APPLY_INV_LEMMA** introduces and instantiates the desired invariant lemma, expands the invariant, and discharges the reachability condition that is the hypothesis of the lemma.

A second technique for reducing tedium is used in the TAME strategy **TRY_SIMP**. **TRY_SIMP** automates certain inferences which are "obvious" to humans but which, in PVS, require detailed user guidance together with knowledge of the behavior of PVS and some of its more obscure proof steps. Several such inferences concern the PVS DATATYPE construct. 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` because by definition `des(a)` returns the element `x` such that `a` = `con(x)`. To establish this needed fact in a proof, the PVS user must apply the PVS step APPLY-EXTENSIONALITY. Establishing other simple facts about data types in PVS can also be tedious, requiring the PVS steps REPLACE and CASE to do explicit substitution and judicious case splitting. The auxiliary local theories that can be generated from a template instantiation provide

the conditional rewrite rules and lemmas used by **TRY_SIMP** to make such inferences automatic when **TRY_SIMP** is invoked.

Easterbrook and Callahan (1997) report that they abandoned early experiments with PVS to prove properties of SCR specifications because "when a proof failed, it took too long to discover the problem." A second major goal of the TAME proof steps is to give the user better feedback both after a proof or proof attempt and while a proof is being constructed.

To achieve better feedback after a full or partial proof has been completed, the TAME proof steps are designed to make saved PVS proofs understandable without executing them in PVS. Saved TAME proofs have a clear structure: the meanings of the proof branches are indicated by comments automatically generated by TAME. The meaning of an individual TAME proof step can be inferred from its name and its arguments. In verbose mode, TAME prints extended comments showing the exact facts introduced, so that the reader of a proof need not look up particular lemmas, preconditions, etc. (See Sections 4 and 6 for example TAME proofs.) Thus, a complete proof can be inspected to see if it succeeded for the expected reasons; examination of a partial proof can pinpoint where in the argument the proof failed or is incomplete.

TAME provides users improved feedback from PVS during a proof by attaching labels to formulae that indicate their origin, and hence their significance. For example, the strategy **AUTO_INDUCT** labels formulae in subgoals corresponding to induction cases according to whether they come from the precondition, inductive hypothesis, or inductive conclusion, or express the fact that the prestate (or poststate) of an induction step is reachable. When the TAME step **SUPPOSE** is applied to an assertion argument `P`, the current subgoal is split into two subgoals, one with the hypothesis `P` labeled `Suppose` and the other with the hypothesis `not(P)` labeled `Suppose not`. Facts introduced with **APPLY_INV_LEMMA** or **APPLY_LEMMA** are given a label `lemma <lemma-name>` derived from the name of the lemma. In later subgoals, the descendants of a labeled formula retain the labels of their parent; the inherited labels thus indicate the reason for the presence of new formulae. The ability to label formulae, a recently added feature of PVS, is important in the execution of, as well as the feedback from, the TAME strategies.

4. Applying TAME to *TIP*

Devillers et al. (2000) describe the direct use of PVS to mechanize the proofs of properties of the automaton *TIP*. Below, we contrast the TAME approach with the direct use of PVS, drawing from the third author's results in applying TAME to *TIP*. For brevity, we refer to "TAME" specifications and proofs and "PVS" specifications and proofs.

4.1. Comparing proofs of invariants

The *TIP* property of interest—for an arbitrary tree topology, exactly one leader is elected—is established by proving two simpler properties: "at most one leader is elected" and "at least one leader is elected". Fourteen invariants, I_1 through I_{14}, are used to prove that at most one leader is elected (invariant I_{15}). Two additional invariants, I_{16} and I_{17}, are used in the proof that at least one leader is elected.

The most dramatic difference between the PVS approach of Devillers et al. (2000) and the TAME approach lies in the proofs of these invariants. The TAME proofs are much shorter, and the significance of proof branches and individual proof steps is much clearer. Moreover, the TAME proofs correspond in a clear way to the hand proofs in Devillers (1997). In fact, the TAME proofs for those *TIP* invariants for which hand proofs were available were done by referring to the hand proofs. For an example, see the discussion of the proof of Invariant I_4 in Section 6. This method differs from the method used by the authors of Devillers et al. (2000), whose PVS proofs followed the subgoal-driven proof style directly supported by PVS. In this style, the next PVS step to use is chosen by looking at the current proof subgoal. These authors found the hand proofs of invariants of little use in guiding the invariant proofs in PVS and hence did not try to follow the reasoning used in the hand proofs.

Proofs in general have a natural tree structure. Branching occurs when the proof breaks into cases or when extra proof obligations are created by a proof step. When a user creates a proof interactively in PVS, PVS saves an executable script of the proof, recording both the proof steps invoked by the user and the branching structure of the proof. In the example TAME and PVS proofs in this paper, the proof steps supplied by the user appear in Roman font, while the names of TAME strategies invoked by the user appear in bold. The parts of the proof scripts created by PVS appear 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 also include comments, in italics and preceded by semicolons, automatically generated by the TAME strategies.

The resemblance of TAME proofs to hand proofs is illustrated by the natural language proof of *TIP* Invariant I_5 in figure 3. This proof was obtained by hand translating the TAME

To prove: For every reachable state `s`, for every edge `e`,
`length(mq(e,s)) <= 1.`

Proof: The proof is by induction. The assertion is trivial for the base case, i.e., when `s` is an initial state. For each action of *TIP*, it remains to prove the corresponding induction case, i.e., that the assertion is preserved by that action. Only three of the induction cases are nontrivial.

Case 1: The action `add_child(addE)`. Let the value of the parameter `addE` of `add_child` be `addE_action`, and consider the edge `e_theorem`. First, apply the specific precondition of `add_child(addE_action)`. Then, consider separately the cases `e_theorem = addE_action` and `e_theorem ≠ addE_action`. In each of these cases, the proof is now obvious.

Case 2: The action `children_known(childV)`. Let the value of the parameter `childV` of children_known be `childV_action`, and consider the edge `e_theorem`. Consider separately two cases. Suppose first that `source(e_theorem) = childV_action`. In this case, first appeal to the specific precondition of `children_known(childV_action)` and then apply Invariant I_2 in the prestate to `e_theorem`. The remainder of the proof in this case is now obvious. The proof in the case `source(e_theorem) ≠ childV_action` is obvious.

Case 3: The action `ack(ackE)`. Let the value of the parameter `ackE` of ack be `ackE_action`, and consider the edge `e_theorem`. Consider separately two cases. Suppose first that `e_theorem = ackE_action`. Appeal to the specific precondition of `ack(ackE_action)`. The proof in this case is now obvious. The proof in the case `e_theorem ≠ ackE_action` is obvious.

Figure 3. Natural language proof of Invariant I_5.

```
Inv_5(s:states): bool = (FORALL (e:Edges): length(mq(e,s)) <= 1);
    (""
    (AUTO_INDUCT)
    (("1" ;;Case add_child(addE_action)
            (APPLY_SPECIFIC_PRECOND)
            (SUPPOSE "e_theorem = addE_action")
            (("1.1" ;;Suppose e_theorem = addE_action
                    (TRY_SIMP))
             ("1.2" ;;Suppose not [e_theorem = addE_action]
                    (TRY_SIMP))))
     ("2" ;;Case children_known(childV_action)
            (SUPPOSE "source(e_theorem) = childV_action")
            (("2.1" ;;Suppose source(e_theorem) = childV_action
                    (APPLY_SPECIFIC_PRECOND)
→                   (APPLY_INV_LEMMA "2" "e_theorem")
                    (TRY_SIMP))
             ("2.2" ;;Suppose not [source(e_theorem) = childV_action]
                    (TRY_SIMP))))
     ("3" ;;Case ack(ackE_action)
            (SUPPOSE "e_theorem = ackE_action")
            (("3.1" ;;Suppose e_theorem = ackE_action
                    (APPLY_SPECIFIC_PRECOND)
                    (TRY_SIMP))
             ("3.2" ;;Suppose not [e_theorem = ackE_action]
                    (TRY_SIMP))))))
```

Figure 4. TAME proof of Invariant I_5. The TAME strategies supplied by the user are in **bold**, and the parts supplied by PVS are in *italics*. The PVS parts include comments automatically generated by the TAME strategies; these comments are preceded by semicolons.

proof of I_5 in figure 4 in a straightforward way. Although the hand proof from which the TAME proof of I_5 was derived is a Lamport-style proof rather than a natural language proof, TAME proofs can also be (and have been) derived from natural language proofs providing the level of detail of the proof in figure 3.

Figure 5 presents the PVS proof of *TIP* Invariant I_5 developed by Devillers et al. (2000). As the translation in figure 3 of the TAME proof of Invariant I_5 illustrates, a TAME proof can be understood by referring only to the specification of the automaton and its invariants; the user need not rerun the TAME proof through the PVS proof checker. PVS proofs in general do not have this property. For example, one must step through the proof in figure 5 with the PVS proof checker to determine the contributions of many of the steps, such as (PROP), (SKOSIMP*), (HIDE −1), (INST?), (REPLACE −2 :HIDE? T), and (LIFT-IF) in the first column.

The PVS encoding of state invariant lemmas is slightly different from the TAME encoding. In the PVS encoding, most invariants—those proved by induction—have two associated lemmas: the first lemma states that the invariant holds in start states and is preserved by transitions, and the second (proved trivially from the first) states that the invariant holds for all reachable states. When induction is not required in the proof—i.e., when the invariant

```
INV_5((s: states)): bool = (FORALL (e: E): length(mq(s)(e)) <= 1)
(""
 (EXPAND "invariant?")
 (PROP)
 ((("1"
  (SKOSIMP*)
  (EXPAND "INV_5")
  (SKOLEM!)
  (EXPAND "Init")
  (PROP)
  (HIDE -1)
  (INST?)
  (PROP)
  (HIDE 1)
  (EXPAND "length")
  (ASSERT))
 ("2"
  (SKOLEM 1 (S _ T))
  (INDUCT "a" 1)
  ((("2.1"
   (SKOSIMP*)
   (EXPAND "steps")
   (EXPAND "A_C_step")
   (PROP)
   (REPLACE -2 :HIDE? T)
   (EXPAND "INV_5")
   (SKOSIMP*)
   (LIFT-IF)
   (PROP)
   ((("2.1.1"
    (INST?)
    (REPLACE -1 :HIDE? T)
    (HIDE -1)
    (HIDE 2)
    (EXPAND "tl")
    (EXPAND "length")
    (ASSERT)
    (LIFT-IF)
    (PROP)
    (ASSERT)
    (EXPAND "length")
    (LIFT-IF)
    (PROP)
    ((("2.1.1.1" (ASSERT))
     ("2.1.1.2" (ASSERT))
     ("2.1.1.3" (ASSERT))))
    ("2.1.2" (INST?))))
   ("2.2"
    (SKOSIMP*)
    (EXPAND "steps")
    (EXPAND "ACK_step")
    (PROP)
    (REPLACE -1 :HIDE? T)
    (EXPAND "INV_5")
    (SKOSIMP*)
    (LIFT-IF)
    (PROP)
    ((("2.2.1"
     (INST?)
     (ASSERT)
     (HIDE -1 2 3)
     (EXPAND "tl")
     (EXPAND "length")
     (LIFT-IF)
     (PROP)
     ((("2.2.1.1" (ASSERT))
      ("2.2.1.2" (ASSERT)
       (EXPAND "length")
       (LIFT-IF)
       (ASSERT))))
     ("2.2.2" (INST?))))
   ("2.3"
    (SKOSIMP*)
    (EXPAND "steps")
    (EXPAND "C_K_step")
    (PROP)
    (REPLACE -3 :HIDE? T)
    (EXPAND "INV_5")
    (SKOSIMP*)
    (LIFT-IF)
    (PROP)
    ((("2.3.1"
     (INST?)
     ((("2.3.1.1"
      (ASSERT)
      → (USE "INV_2_reach")
      → (EXPAND "INV_2")
      → (INST?)
      (ASSERT)
      (INST?)
      ((("2.3.1.1.1"
       (EXPAND "member")
       (EXPAND "froms")
       (ASSERT)
       (HIDE -2 -3 -4)
       (INST?)
       (EXPAND "append")
       (EXPAND "length" 1)
       (EXPAND "length")
       (ASSERT))
      ("2.3.1.1.2"
       (EXPAND "tos")
       (EXPAND "target")
       (EXPAND "inv")
       (EXPAND "member")
       (EXPAND "froms")
       (PROPAX))))
      ("2.3.1.2"
       (EXPAND "tos")
       (EXPAND "member")
       (EXPAND "froms")
       (EXPAND "inv")
       (EXPAND "target")
       (PROPAX))))
     ("2.3.2" (HIDE -2) (INST?))))
   ("2.4" (SKOSIMP*)
    (EXPAND "steps")
    (EXPAND "R_C_step")
    (PROP)
    (REPLACE -3 :HIDE? T)
    (EXPAND "INV_5")
    (PROPAX))
   ("2.5"
    (SKOSIMP*)
    (EXPAND "steps")
    (EXPAND "ROOT_step")
    (PROP)
    (REPLACE -2 :HIDE? T)
    (EXPAND "INV_5")
    (PROPAX))))))
```

Figure 5. PVS proof of Invariant I_5 by Devillers et al. The proof steps supplied by the user are in Roman font, and the parts supplied by PVS e.g. tree addresses and PROPAX are in *italics*.

follows from other invariants—only the second lemma is needed. The TAME encoding of every state invariant lemma is equivalent to the second PVS lemma. For proofs requiring induction, the strategy **AUTO_INDUCT** first reduces this lemma to the equivalent of the first PVS lemma before performing many of the standard initial proof steps. Thus, the TAME

proofs by induction of invariants correspond to the PVS proofs of the first associated lemma in the PVS encoding.

The difference between corresponding TAME and PVS proofs is illustrated by the TAME and PVS proofs for I_5 in figures 4 and 5. Both proofs were created interactively with the PVS prover. The user supplied all of the information in the proofs except the parts in italics. An obvious difference between the proofs is the number of proof steps: the TAME proof requires the user to supply only 14 steps, while the PVS proof requires the user to supply 112 steps. The two proofs also have different structures. The PVS proof tree (see figure 5) has two branches at the top level, with the second dividing into five parts. The first branch corresponds to the base case of the induction proof, while the five parts into which the second branch divides correspond to the five induction cases—one for each of the five actions of *TIP* (see Appendix A). In contrast, the TAME proof tree (see figure 4) has three top level branches.

Although the two proofs have different structures, some relationships can be found. The PVS proof clearly has repeating patterns; the TAME strategies take advantage of such repeating patterns to produce higher-level proof steps. One pattern involves the application of an invariant lemma. The three arrows at the bottom of the middle column of the PVS proof in figure 5 mark the three steps that apply Invariant I_2 to the current state (before the action) and the skolem constant for the edge e of Invariant I_5. (A skolem constant is a generic instance of a universally quantified variable.) In the TAME proof in figure 4, this is accomplished by the proof step (**APPLY_INV_LEMMA** "2" "e_theorem") in the second proof branch (also marked with an arrow). Most of the repeating patterns are handled by either **AUTO_INDUCT** or **TRY_SIMP**. For example, the base case and last two induction cases, whose proofs are "obvious" to a human, are done automatically by **AUTO_INDUCT**. Hence the three top-level branches in the TAME proof, which represent the three nontrivial action cases, correspond to the branches *"2.1"*, *"2.2"*, and *"2.3"* of the PVS proof (but not in that order).

Figure 6 lists the number of steps and relative execution times of the TAME proofs and the PVS proofs for all 17 *TIP* invariants. The two numbers in parentheses correspond to slightly altered versions of invariants used in the PVS proofs. Invariant I_{15} did not have a PVS proof. The second, improved set of TAME statistics for Invariant I_{17} are for an improved TAME proof created by using as hints the commands to expand invariants in the PVS proof. These commands indicated which invariants were used, but not the case in which they were used and not always the arguments to which they were applied. Thus, the TAME proof could not be directly derived from the PVS proof.

As figure 6 shows, the lengths of the TAME proofs of invariants I_1 through I_{14} are on the average more than five times shorter than the corresponding PVS proofs. Because high-level strategies often use some trial-and-error, the proof execution times in the *TIP* example average about twice as long for the TAME proofs as for their corresponding PVS proofs (the maximum ratio—for the proof of the three invariants I_6, I_7, and I_8 combined—was 24 seconds for TAME vs 9 seconds for PVS). However, the relative simplicity and clarity of the TAME proofs strongly suggests that the human time needed to construct the proofs with TAME is considerably shorter than that needed to construct proofs with the direct PVS approach of Devillers et al. (2000).

Invariant	Number of steps		Time (seconds)	
	PVS	*TAME*	*PVS*	*TAME*
Inv 1	51	5	2.38	3.30
Inv 2	69	1	2.85	2.94
Inv 3	63	10	2.86	4.20
Inv 4 (4s)	(60)	8	(3.78)	4.53
Inv 5	112	14	3.75	4.98
Invs 6, 7, & 8	218	54	9.05	23.98
Inv 9	80	12	4.15	8.89
Inv 10	115	20	5.59	12.66
Invs 11 & 12	181	55	6.32	12.79
Inv 13	57	6	2.35	3.34
Inv 14	25	5	0.85	0.76
Invs 1-14	***1031***	***190***	***43.93***	***82.37***
Inv 15	—	10	—	1.50
Inv 16 (16r)	(76)	3 (6)	3.92	4.27 (4.90)
Inv 17	132	22 → 16	6.26	17.50 → 10.78

Figure 6. Statistics for TAME vs PVS proofs. Execution times are for PVS 2.3 on an UltraSPARC-II.

4.2. Comparing specifications

As expected of two independent encodings of a problem, the PVS and TAME specifications have rather different structures. The PVS specification of the automaton *TIP* involves a large set of automaton-specific theories with a complex import structure having several (around nine) levels. Moreover, the organization of the import structure is at least partly problem-specific. Thus, how one would use the same methodology to specify a different I/O automaton in PVS is not obvious. In contrast, the TAME specification of *TIP* is essentially a single automaton-specific theory that imports instantiations of a small collection of generic theories. Only one of these generic theories—the theory `states`, which defines the state type `states` by combining the automaton-specific "basic" state type defined in the TAME template with a standard part associated with time values—involves the automaton definition; the others are used in theorem proving support. Hence, unlike the PVS specification, the TAME specification of the automaton involves almost no layering of definitions. As a result, the TAME specification is more easily grasped as a whole, and its correspondence to the original I/O automaton description is easier to understand. There are additional, automaton-specific theories associated with the TAME specification that can be derived in a standard, automatable way from the automaton specification. These theories supply lemmas to the generic TAME strategies.

The PVS and TAME specifications of *TIP* also differ in the way they capture the transitions of the automaton. In the PVS specification, each transition is described using the combined

information from the precondition and effect of each action. In TAME, the preconditions and effects of actions are defined separately. In a few instances, some information from the precondition is needed as a guard in the definition of the effect for the definition to pass type checking. Experience with many examples has shown that in practice, this rarely happens. When possible, separating the precondition and effect of an action provides an advantage in creating understandable induction proofs: in the induction step for each action, one can first reason about the effect of the action and then determine whether or not the precondition is also needed in the justification. Thus, it allows one to determine whether a specification property might be affected when a precondition is changed.

4.3. Beyond invariants: Simulation and refinement

Because PVS lacks support for defining a general automaton type and for passing theory parameters to theories, completely general definitions of simulation and the closely related notion of refinement (as defined in (Lynch and Vaandrager, 1995)) are awkward to express in PVS. For this reason, TAME does not yet include specialized support for proving simulations or refinements. However, the PVS specification of *TIP* does include a definition of the refinement relation in the most convenient general form that can currently be provided with PVS. The definition makes use of a theory that expects as parameters a type of actions and a type of states, and defines an automaton type limited by these parameters. In addition, the PVS proofs for *TIP* include a proof that *TIP* is a refinement of another automaton called *SPEC*. In this respect, the PVS specification and proofs have an advantage over the TAME specification and proofs. Theories of the form supporting the definition of refinement in the PVS specification can almost certainly be adapted for use with a new TAME "refinement" template. Rather than use this approach, however, a future version of TAME will use PVS support for theory parameters to theories (to be provided in a future version of PVS (Lincoln, 1998)) to provide generic definitions of simulation and refinement with associated proof strategies.

5. Experience of a new TAME user

This section describes the experience of the third author, who had no previous experience with TAME or PVS, in applying TAME to the examples described in Section 2. The third author first applied TAME to the examples from Romijn's solution to the RPC-Memory Problem (Romijn, 1996), and then to *TIP* and its invariants (Devillers et al., 2000; Devillers, 1997). For the RPC-Memory example, this required specifying the three I/O automata *Memory**, *MemoryImp*, and *Imp* and proving 24 associated invariants. For *TIP*, it required using TAME to specify the single automaton and to check the proofs of the 15 invariants for which hand proofs were supplied. No hand proofs were supplied for the two additional *TIP* invariants; these were later proved by the first author using TAME. Below, we first illustrate the specification support provided by TAME by describing how the third author completed the TAME template using the *TIP* specification in Appendix A. We then discuss the time she required to apply TAME to the examples, special problems she encountered and solved,

the extent to which the TAME strategies were sufficient for mechanizing the proofs, and some errors in the specifications and proofs she discovered during the mechanization.

5.1. Specifying TIP in TAME

The form of the *TIP* specification in Appendix A was used for all of the I/O automata studied by the third author. Appendix B shows the TAME template instantiation derived for *TIP*. The third author was able to instantiate the TAME template for each I/O automaton specification in a largely straightforward way as follows. The definitions of the actions of the I/O automaton provide the names and argument types needed for their TAME declarations, preconditions and effects. Thus, for the *TIP* action *CHILDREN_KNOWN*, she added the constructor `children_known(childV:Vertices)` to the `actions` datatype, and appropriate `children_known(v)` cases to the definitions of `enabled_specific` and `trans`, which respectively describe the preconditions and effects of actions. The definitions of the state variables and their types in the I/O automaton specification provide the information needed to define the type of the basic state as well as any needed auxiliary type definitions in the TAME specification. Thus, the declaration $init : \mathbf{V} \rightarrow \mathbf{Bool}$ in the *TIP* specification becomes the component `init:[Vertices->Bool]` in the definition of the basic state type `MMTstates` in TAME. The initial state information for the I/O automaton is translated into the initial state predicate `start` of the TAME specification. Because the state variables all have function type, the third author used `LAMBDA` expressions (i.e., expressions with function values) to express their initial values in the `basic` component of `start`. Finally, any constants and predicates relating constants defined for the I/O automaton can be represented in the TAME specification using constant declarations and the axiom `const_facts`. Because there were no predicates relating constants connected with the *TIP* specification, the axiom `const_facts` retains its default value `true`. PVS declarations of the types and certain function constants such as `source` and `target` used in the *TIP* specification are given at the beginning of the TAME specification. Instead of defining the list functions and constants hd, tail, empty, and append used in the *TIP* specification, the third author chose to use PVS's built-in Lisp versions, i.e., `car`, `cdr`, `null`, and `cons`.

5.2. Time required

Figure 7 summarizes the time required by the third author to apply TAME to the *RPC* and *TIP* examples. She needed approximately a week to read and understand earlier TAME specifications. These specifications include auxiliary theories, which are derived from a template instantiation and currently must be generated by hand. In addition, she needed about a day to learn how to use TAME to obtain a proof.

Once this initial learning period was complete, specifying *Memory** in TAME and creating its auxiliary theories required about two days, and the proofs of its three invariants, plus a fourth auxiliary invariant, required only a few hours (see figure 7). Some of this time was used to discover the need for, and the formulation of, the auxiliary invariant. As shown in figure 7, specifying *MemoryImp* in TAME required only a few days, whereas proving its 12 invariants required about ten days. The time required to prove these 12 invariants

Automaton Name	Time (days) to specify in *TAME*	Number of invariants	Time (days) to prove in *TAME*	Remarks
Memory*	2	3	0.5	+ 6 days learning *TAME*. 1 aux. invariant needed.
MemoryImp	3	12	10	1 aux. invariant needed.
Imp	2	7	1	—
TIP	2.5	15	2.5	1 aux. axiom used. 4 aux. lemmas proved. 2 aux. invariants proved.
Total	9.5	37	14	—

Figure 7. Time required to apply TAME to *RPC* and *TIP*.

was longer for a combination of reasons. First, the proofs of these invariants were more complex than the proofs of the *Memory** invariants. Because the proof of one invariant was only loosely sketched, some time was required to determine all of the facts, including an additional invariant lemma, required in its proof. Trying to understand the scopes of the quantifiers in one of the invariants led to a weaker initial formulation of the invariant that was insufficient for the proofs of later invariants, and this had to be discovered and rectified. Finally, the third author encountered some situations in which the TAME strategies had to be supplemented by special knowledge and the direct use of PVS. Once appropriate improvements were made to TAME (see Section 6.3), she was able to quickly complete the proofs. After her experience with *Memory** and *MemoryImp*, specifying *Imp* and proving its seven invariants took only three days, and specifying *TIP* and proving its 15 invariants took only five days.

5.3. Special problems

While translating I/O automata specifications into the TAME template is largely straightforward, creativity was needed for some aspects of the translation. These aspects usually concerned type definitions. For example, the specification of *MemoryImp* required the composition of several I/O automata specifications into a single TAME specification, thus requiring that the output actions of certain automata be combined with input actions of others. For two of the six combined actions, creativity was required in defining the action parameter types to make an output action and an input action compatible. In addition, several state variable and action parameter types in the RPC-Memory automaton had complex subtype relationships. The third author's original definitions of these relationships in TAME led to several unprovable TCCs (type correctness conditions generated by the PVS type checker). One approach to making the TCCs provable is to include axioms describing the subtype relationships in the specification. Instead, the third author defined the types as appropriate subtypes of a PVS datatype. Doing this permits the TCCs to be proved automatically in PVS and avoids the possible introduction of inconsistent axioms.

Both *Imp* and *TIP* have a step whose definition uses a `for` construct to simultaneously update a set of variables whose indices satisfy a certain predicate. Some ingenuity was

required to solve the problem of representing `for` in TAME, since PVS does not have such a construct. The third author's solution was to use the PVS LAMBDA construct to represent each use of `for` by a function from variables to their values in the new state.

In addition to the information required in the TAME template, auxiliary information is sometimes needed. For the RPC-Memory automata, a few auxiliary functions and predicates defined in the original I/O automata specifications were also included in the TAME specifications. For *TIP*, a few auxiliary results about data structures were also required. A small set of lemmas about the relationship between edges and their reverse edges was needed to mechanize steps in hand proofs whose justification in Devillers (1997) was "math". These were simple enough to prove using GRIND, PVS's general automatic proof step.[3] In addition, a subset of the theory of tree-structured digraphs was needed in the proof of Invariant I_{15}. Rather than using the full theory developed by Devillers et al., the third author simply determined the essential fact needed about such digraphs—that they are connected—and included it as an axiom. Using this axiom, she proved several auxiliary invariants needed in the proof of I_{15}.

5.4. *Sufficiency of the TAME strategies*

Once improvements to the TAME strategies (due to feedback from the third author) were complete, the strategies listed in figure 2 were almost sufficient to obtain all of the proofs for the RPC-Memory example. **APPLY_IND_HYP** and **APPLY_LEMMA** were not needed. In a few places, new TAME strategies (**INST_IN** and **SKOLEM_IN**, which are described in Section 6.3) and the PVS steps EXPAND and INST were used.

The proofs of the *TIP* invariants used all of the strategies in figure 2, together with **INST_IN**, **SKOLEM_IN**, EXPAND, and INST; in addition, the PVS step SPLIT was used to separate threads in the combined proofs of several lemmas, and two additional TAME steps, **COMPUTE_POSTSTATE** and **DIRECT_INDUCTION** were required. The step **COMPUTE_POSTSTATE** was needed to introduce facts about the poststate required in a proof in which it was natural to refer to the poststate in a supposition introduced with **SUPPOSE**. The proof of one auxiliary invariant lemma for *TIP* introduced by the third author required induction over the natural numbers, but not over the reachable states of *TIP*. She mechanized this proof using variants of the PVS SKOLEM command, PVS's EXPAND, INDUCT, and INST commands, and TAME's **APPLY_INV_LEMMA** strategy. The step **DIRECT_INDUCTION** was developed to prove invariants whose proof requires mathematical induction followed by direct (non-induction) proofs of the branches. With the aid of **DIRECT_INDUCTION**, the proof can be mechanized using PVS's EXPAND, together with the TAME strategies **APPLY_INV_LEMMA**, **SKOLEM_IN**, and **APPLY_IND_HYP** (used to apply the mathematical induction hypothesis). Although *TIP* was the first TAME application requiring **DIRECT_INDUCTION**, this strategy has since proved useful in the verification of a multicast stream authentication protocol (Archer, 2002).

5.5. *Specification and proof errors discovered*

The specifications and proofs of both the RPC-Memory and *TIP* examples were very carefully done. Thus, the third author uncovered only a few errors. Specifying the RPC-Memory

automata in TAME and applying the PVS type checker exposed a few cases of incompleteness and inconsistency in the specifications. For example, the intended types of certain constants were unclear, and there was a type inconsistency in the definition and use of one function. No specification errors were found in the *TIP* example, which is not surprising, since the specification and invariants had already been checked in PVS. But the PVS proofs for *TIP* were not derived from the hand proofs, so although the *TIP* invariants had been checked, their hand proofs had not been checked. Using TAME, the third author discovered a few cases of incorrect inferences or justifications in both the hand proofs for *TIP* and the RPC-Memory proofs. She was able to correct all of these problems in the TAME proofs, usually in a very straightforward way, and in one case by identifying and proving an auxiliary invariant. Thus, like Rudnicki and Trybulec (1996), she found that Lamport-style proofs, though very structured and detailed, are still informal and as a result may contain incorrect or missing details. Her results led to corrections by Romijn and Devillers et al. in both the specifications and proofs in Romijn (1996) and Devillers (1997).

6. Discussion

This section presents several observations resulting from our case studies. First, as noted in Section 4, TAME proofs are readily constructed from hand proofs that give sufficient detail. A hand proof that indicates which facts were used on which proof branches and which subcases need to be considered usually provides sufficient detail; details of inferences drawn from the facts are normally not required. In previous applications (e.g. (Archer and Heitmeyer, 1996, 1997b)), the hand proofs mechanized with TAME were English language proofs. As indicated in Section 2, the majority of the proofs mechanized by the third author using TAME were Lamport-style proofs. These proved to be as straightforward as English language proofs to mechanize in TAME. Section 6.1 gives an example of the correspondence between a Lamport-style proof and the TAME proof derived from it. Second, as noted in Section 4, TAME proofs are intended to be understandable without reference to the PVS proof checker. In Section 6.2, we describe how TAME proofs can actually be interpreted as English language proofs, using the TAME proof from Section 6.1 as an example. Finally, several improvements were made to TAME as a result of the third author's experience. Section 6.3 discusses these improvements and some issues they raise.

6.1. Constructing TAME proofs

The proof in figure 8 (provided to the reader as an aid) describes in English the complete TAME proof (see figure 9) of *TIP* Invariant I_4. As an illustration of how a TAME proof can be obtained from a Lamport-style proof, figure 9 shows the correspondence between the TAME steps and the relevant steps from a Lamport-style proof of Invariant I_4. These relevant steps are shown in figure 10, which contains the single branch of the hand proof that TAME found to be nontrivial; the remainder of the Lamport-style proof was done automatically by TAME.

To understand the relationship between the two kinds of proofs, we can compare the details of the proofs of I_4 in figures 10 and 9. In the Lamport-style proof (figure 10), the values s and

To prove: For every reachable state `s`,
for all edges `e` and `f` and every vertex `v`,
if `target(e) = target(f) = v` and `e ≠ f`,
then `init(v)` or `child(e)` or `child(f)`.

Proof: The proof is by induction. The assertion is trivial for the base case, i.e., when `s` is an initial state. For each action of *TIP*, it remains to prove the corresponding induction case, i.e., that the assertion is preserved by that action. Only one of the induction cases is nontrivial: the case of the action `children_known(childV)`.

Thus, consider the case when the action is `children_known(childV)`. Let the value of the parameter `childV` of `children_known` be `childV_action`. Let the state before this action be `prestate`. Let the edges `e` and `f` and the vertex `v` be `e_theorem`, `f_theorem`, and `v_theorem`. Consider separately two cases. Suppose first that `v_theorem = childV_action`. In this case, introduce the fact that

```
init(childV_action, prestate) ∧
∀ e:tov(childV_action), f:tov(childV_action).
    child(e,prestate) ∨ child(f,prestate) ∨ e = f
```

by appealing to the specific precondition of `children_known(childV_action)` in the state `prestate`. Instantiating the second part of this precondition with `e_theorem` and `f_theorem`, the remainder of the proof in this case is now obvious. The fact that `e_theorem` and `f_theorem` belong to the subtype `tov(childV_action)` of the type `Edges` is also obvious. The proof in the case `v_theorem ≠ childV_action` is obvious.

Figure 8. English translation of TAME proof of I_4.

t represent the prestate and poststate in the induction step, and the values f, g, and v' are, respectively, the skolem constants for the quantified variables e, f, and v in I_4, which TAME automatically names `e_theorem`, `f_theorem`, and `v_theorem`. The action *C_KNOWN*(v) in the Lamport proof corresponds to the action `children_known(childV_action)` in the TAME proof; the name `childV_action` is constructed automatically by TAME from the name of the formal parameter `childV` of `children_known`. We added annotations (see the right-hand column of figure 9) to the TAME proof to show, for each of its steps or branches, the step of the Lamport proof containing a corresponding inference or justification. For example, the appeal "by IH" to the inductive hypothesis at step <3.1> in the Lamport-style proof is handled automatically by TAME's **AUTO_INDUCT** strategy since, for this proof, the correct instantiation of the variables in the inductive hypothesis is the skolem constants. The only steps the TAME user must supply, besides **TRY_SIMP**, are the **SUPPOSE** for the case distinction at step <3.2.2> and the **APPLY_SPECIFIC_PRECOND** and INST corresponding to application of the precondition to f and g at step <3.2.3.1>. Checking in TAME that f and g are of type $to(v)$ at step <3.2.3.1> is handled by proving the TCCs generated by PVS as the result of the instantiation step INST—this is accomplished by the proof steps **TRY_SIMP** at *"1.2"* and *"1.3"* in the TAME proof. Introducing the effect of the action and setting up Invariant I_4 in the poststate as a proof goal are both handled automatically in the TAME proof by **AUTO_INDUCT**, and appeals to previous proof steps are handled automatically in the TAME proof by the final **TRY_SIMP**.

```
Inv_4(s:states): bool =
    (FORALL (e,f:Edges, v:Vertices):
            (target(e)=v AND target(f)=v AND NOT(e=f))
                ⇒ (init(v,s) OR child(e,s) OR child(f,s)))

("" (AUTO_INDUCT)
   ;; Case children_known(childV_action)          < 3 >, < 3.1 >, < 3.2 >,
                                                   < 3.2.1 >
  (SUPPOSE "v_theorem = childV_action")            < 3.2.2 >
  (("1" ;; Suppose v_theorem = childV_action       < 3.2.3 >
      (APPLY_SPECIFIC_PRECOND)                     < 3.2.3.1 >
      ;; Applying the precondition
      ;; init(childV_action, prestate)
      ;;        AND
      ;;     (FORALL (e: tov(childV_action)):
      ;;     FORALL (f: tov(childV_action)):
      ;;      child(e, prestate) OR
      ;;      child(f, prestate) OR e = f)
      (INST "specific-precondition_part_2"
            "e_theorem" "f_theorem")
      (("1.1" (TRY_SIMP))                          < 3.2.3.2 >, < 3.2.3.3 >
       ("1.2" (TRY_SIMP))                          < 3.2.3.1 >
       ("1.3" (TRY_SIMP))))                        < 3.2.3.1 >
   ("2" ;; Suppose not [v_theorem = childV_action] < 3.2.4 >
      (TRY_SIMP))))                                < 3.2.4.1 >, < 3.2.4.2 >
                                                   < 3.2.5 >, < 3.3 >
```

Figure 9. Complete TAME proof (verbose) of I_4.

$$I_4 = \forall_{e,f,v} target(e) = target(f) = v \wedge e \neq f \Rightarrow init(v) \vee child[e] \vee child[f]$$

<3>	Assume $a = C_KNOWN(v)$, $v \in \mathbf{V}$	
<3.1>	. $s \models I_4$	(by IH)
<3.2>	. Take arbitrary f, g, v' such that $target(f) = target(g) = v \wedge g \neq f$	
<3.2.1>	. . $s \models init(v') \vee child[f] \vee child[g]$	
<3.2.2>	. . Case distinction on $v' = v$	
<3.2.3>	. . Assume $v' = v$	
<3.2.3.1>	. . . $s \models child[f] \vee child[g]$	(pre. $C_KNOWN(v)$ and $f, g \in to(v)$)
<3.2.3.2>	. . . $t \models child[f] \vee child[g]$	(eff. $C_KNOWN(v)$ does not change $child$)
<3.2.3.3>	. . . $t \models init(v) \vee child[f] \vee child[g]$	
<3.2.4>	. . Assume $\neg(v' = v)$	
<3.2.4.1>	. . . $s \models init(v') \vee child[f] \vee child[g]$	(by <3.2.1>)
<3.2.4.2>	. . . $t \models init(v') \vee child[f] \vee child[g]$	(eff. $C_KNOWN(v)$ does not change $child$ or $init[v']$ by <3.2.4>)
<3.2.5>	. . $t \models init(v') \vee child[f] \vee child[g]$	
<3.3>	. $t \models I_4$	(def. I_4)

Figure 10. Nontrivial branch of Lamport-style proof of I_4.

6.2. *Explaining TAME proofs*

Because the meanings of TAME proof steps are essentially independent of the proof state current at the time they are executed by the PVS proof checker, TAME proofs can be understood from their saved scripts by referring to the original specification and by knowing the conventions TAME uses for skolemization and instantiation. Thus, given the TAME proof in figure 9, it is fairly straightforward to derive the equivalent English language proof in figure 8. Knowledge of TAME's conventions about skolemization is used in specializing the action parameter `childV` to `childV_action`, and `s`, `e`, `f`, and `v` in the theorem to `prestate`, `e_theorem`, `f_theorem`, and `v_theorem`. One convention about instantiation is used in the INST command in the TAME proof: a precondition in the form of a conjunction is broken down into "_part_1", "_part_2", and so on, in order. A second convention about instantiation is reflected in the English translation in figure 3 of the **APPLY_INV_LEMMA** step in the TAME proof in figure 4: unless a state argument is given, the invariant lemma is applied to the state `prestate`. One additional question that arises in interpreting the TAME proof in figure 9 is why the INST step in the proof results in three subgoals instead of one. When extra subgoals from an INST occur, PVS has generated one or more TCCs as extra proof branches, requiring the user to show that values used in the instantiation have the correct types.

Aside from the problem of identifying TCCs, the derivation of an English language proof from a TAME proof is straightforward enough to be automated, and hence we have recently implemented a prototype translator of saved TAME proofs. Applying the prototype translator to the TAME proof in figure 9 yields the translation in figure 11. Note that the TAME proof in figure 9 is a verbose TAME proof, in contrast to the non-verbose TAME proof in figure 4. Thus, details such as the actual fact introduced by **APPLY_SPECIFIC_PRECOND** in figure 9 could be incorporated into the English version (although the prototype translator does not yet do this). Had the proof in figure 4 been verbose, the actual fact introduced by **APPLY_INV_LEMMA** would have been displayed in the TAME proof (as well as the facts introduced by each of the three uses of **APPLY_SPECIFIC_PRECOND**). An alternative to

---

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

• **Consider the action** `children_known(childV_action)`. The proof in this case is as follows. Suppose first that `v_theorem = childV_action`. Apply the precondition of the action `children_known(childV_action)`. Instantiate specific-precondition_part_2 with the values `e_theorem` and `f_theorem`. There are 2 type correctness conditions associated with the instantiation. Assume first that the type correctness conditions are satisfied. The rest of the proof in this case is obvious. The proofs of the type correctness conditions are as follows. Condition 1: The rest of the proof in this case is obvious. Condition 2: The rest of the proof in this case is obvious. Therefore, the instantiations were type correct. This completes the proof when `v_theorem = childV_action`. Suppose, on the other hand, that it is not true that `v_theorem = childV_action`. The rest of the proof in this case is obvious. This completes the proof for the action `children_known(childV_action)`. □

Figure 11. Automated translation of the TAME proof in figure 9.

translating TAME proofs from their saved scripts to obtain an English language version is to create an English language version simultaneously with the TAME version. Implementing this technique would allow even more detail to be incorporated in the English version, if desired, and would better facilitate interpreting extra TCC subgoals in English.

6.3. Improvements made to TAME

Several improvements were made to the template, the strategies, and the supporting theories of TAME as the result of feedback from the third author. The first improvement generalizes the template. Improvements to the strategies have made TAME more user-friendly by reducing the amount of low-level reasoning associated with certain proof steps. Improvements to the supporting theories extend the scope of the high-level reasoning supported in TAME. We discuss these improvements below, along with some issues they raise.

6.3.1. Improving the template. The base case of induction proofs corresponding to the start states is usually trivial to prove. The strategy **AUTO_INDUCT** is designed to prove the base case automatically, when possible. Although none of the TAME templates enforce any condition on the form of the start state predicate `start`, the automatic proof of the base case by **AUTO_INDUCT** works best if `start(s)` is expressed as an equality `s = <start-state>`, where `<start-state>` is a record value. In previous applications of TAME, each automaton had a single start state, and thus the convention was that `<start-state>` was an explicit record giving the initial values of all state variables.

In the RPC-Memory example, the third author encountered an automaton in which the start state was not unique: initial values were given for only some of the basic state variables. She therefore developed a new template convention for the start state, in which `<start-state>` is a record with its time-related components assigned the standard initial values and its `basic` component (representing the basic state variables) assigned its "old" value updated with values for those variables whose initial values are specified. This is easily done using the PVS construct WITH for updating records and functions and has the effect of leaving the non-updated variables of the basic state unspecified, as desired. Moreover, the strategy **AUTO_INDUCT** works just as well for proving base cases with the new conventional form for `start(s)` as it did with the old one.

6.3.2. Improving the strategies. As noted in Section 5, the third author had difficulty translating a few of the steps from hand proofs into TAME. One such step was the application of an invariant lemma to the poststate of a transition in an induction step. The default used by the TAME step **APPLY_INV_LEMMA** is to apply the lemma to `prestate`; applying the lemma to a different state requires the state in question to be passed as an explicit argument. TAME previously represented the poststate as `trans(<action>,prestate)`, where `<action>` is the action of the induction step, and maintained among the hypotheses the fact that `trans(<action>,prestate)` is reachable, to facilitate application of invariant lemmas to the poststate. However, this representation of the poststate complicated applying an invariant lemma. Not only did the user have to supply `trans(<action>,prestate)` as an argument to **APPLY_INV_LEMMA**, where `<action>` itself could be an expression

with parameters, but after doing this, the user had to explicitly expand the transition function `trans`. The third author's difficulties inspired improvements to **AUTO_INDUCT** and **APPLY_INV_LEMMA** that hide this complexity from the user. The term `trans(<action>,prestate)` is now represented simply as `poststate`, and to apply an invariant lemma to the poststate, the user simply applies **APPLY_INV_LEMMA** to the argument `poststate` and any other arguments to the lemma.

As noted in Archer and Heitmeyer (1997a) the inability of the user to instantiate or skolemize with respect to embedded quantifiers in PVS sometimes makes it difficult to follow the structure of a hand proof using PVS. The third author encountered this problem in some proofs of the RPC-Memory example. To address the problem, two new strategies, called **INST_IN** and **SKOLEM_IN**, were added to TAME to approximate internal instantiation and skolemization. These strategies perform automated simplification in an attempt to handle the non-quantified parts of a formula, and then use the standard PVS proof steps INST and SKOLEM. Although some wasteful proof branching can result (this happened with one RPC-Memory lemma), in many cases, this approach handles embedded quantifiers efficiently.

6.3.3. Improving the supporting theories. The state variables used in I/O automata specifications do not always have simple types. For example, some automata from the RPC-Memory example use state variables that must be represented as "datatypes" using the PVS DATATYPE construct. As noted in Section 3, TAME supports "obvious" reasoning about datatypes using auxiliary theories generated from a template instantiation. Previous to the third author's use of TAME, these auxiliary theories contained only lemmas to support reasoning about the datatype `actions`. Because of the additional datatypes used in the RPC-Memory automata, the auxiliary theories now include lemmas for *all* datatypes in a template instantiation.

In the specification of the automaton *TIP* from Devillers et al. (2000) (see Appendix A), the type of the state variable `mq(e)`, where `e` is an edge, is defined to be `Bool*`, that is, a list of Booleans. Several of the *TIP* invariant lemmas involving `mq(e)` require reasoning about lengths of lists. Because PVS has a built-in type `list[T]`, where `T` is a type parameter, it is reasonable to add auxiliary lemmas to support reasoning about lengths of lists. Figure 12 shows the set of lemmas used as rewrite rules for lists in TAME. Those rules with an asterisk were actually applied by TAME in proving the *TIP* invariants. The PVS proof in figure 5 contains several instances of the PVS command '(EXPAND "length")'; these mark places

* **1.** length(emptylist) = 0

2. $\forall$ L:list. length(L) = 0 $\Rightarrow$ L = emptylist

3. $\forall$ n:nat, e:element, L:list. length(L) = n $\Rightarrow$ length(cons(e,L)) = n+1

* **4.** $\forall$ e:element, L:list. L = emptylist $\Rightarrow$ length(cons(e,L)) = 1

* **5.** $\forall$ n:nat, e:element, L:list. L $\neq$ emptylist $\wedge$ length(L) = n $\Rightarrow$ length(cdr(L)) = n−1

* **6.** $\forall$ n:nat, e:element, L:list. L $\neq$ emptylist $\wedge$ length(L) $\leq$ n $\Rightarrow$ length(cdr(L)) $\leq$ n−1

7. $\forall$ L:list. (length(L) < 0) = FALSE

Figure 12. Rewrite rules for lists used by TAME.

where simple reasoning about length is occurring. With the rules in figure 12, the TAME user does not need to guide PVS through this reasoning.

While one might argue that rewrite rules for lists should be standard in TAME because `list[T]` is a standard type in PVS, there are many examples in which state variables have other complex types, as we discovered in applying TAME to some of the invariant lemmas of Fekete et al. (1997). For reasons of proof efficiency, the number of rewrite rules that are always active should be limited to those that are relevant. Thus, a practical approach for handling "obvious" reasoning about complex types is to use a library of PVS theories containing the lemmas needed to support such reasoning. This would allow the theories to be imported only when required. Such theories need to be developed with care; we do not guarantee the theory in figure 12 to be the best theory to support obvious reasoning about lengths of lists. The extent to which existing libraries developed by other members of the PVS user community would be useful in TAME is an open question.

7. Related work

An increasing number of proof assistants, including assistants for the Duration Calculus (Skakkebaek and Shankar, 1994), for the TRIO logic (Alborghetti et al., 1997; Gargantini and Morzenti, 2001), and for proving invariant properties of DisCo specifications (Kellomaki, 1997), use PVS as the underlying prover. Both the Duration Calculus and TRIO assistants support proofs using steps from particular logics, and use specialized pretty-printers to display PVS sequents in the languages of these logics. The DisCo assistant supports proofs of properties of DisCo specifications, using Lamport's Temporal Logic of Actions, with specialized PVS strategies generated by a compiler. These strategies, though uniform in concept, are specific to each given application. A similar approach was used in an earlier version of TAME; PVS enhancements, especially the documentation of the internal structure of PVS sequents, have allowed us to make the TAME strategies more generic.

Several researchers have applied mechanical theorem provers to LV timed automata or I/O automata. In addition to the application of PVS described in Devillers et al. (2000), Luchangco (1995) describes how the Larch theorem prover LP was used to prove properties of several protocols specified as LV timed automata, and Müller (1998) describes a verification environment for I/O automata based on Isabelle; like Devillers et al. (2000), both include simulation proofs as well as proofs of invariants. In addition, Müller (1998) develops a detailed metatheory for I/O automata. TAME has an advantage over Larch and Isabelle: it produces compact, informative proof scripts. Although Larch provides detailed proof scripts with some information on the content of a proof, Larch does not support the matching of high level natural proof steps with user-defined strategies, nor the automatic documentation of a proof through comments provided by TAME. While Isabelle tactics perform some of the services of the TAME strategies (Müller, 1998), Isabelle does not save proof scripts for completed proofs.

A toolset has been developed that provides an automatic translator from the IOA language for I/O automata to Larch specifications and an interface to the Larch theorem prover LP (Garland and Lynch, 1998). This toolset will eventually include a similar translator to PVS that is being developed by Devillers and Vaandrager; a prototype now exists (Devillers, 1999). TAME currently has a prototype translator from specifications in the

SCR language to TAME specifications (Archer et al., 1998), and an automatic translator from IOA specifications is planned.

8. Conclusion

Miller (1998) discusses several major problems encountered in the AAMP5 study, in which PVS was used to prove the correctness of a set of microcode instructions. Two problems were how to organize the specification and how to structure complex proofs. He also notes that the learning curve in this project was very steep, that many supporting theories had to be developed, and that the robustness of proofs became a concern when specifications were modified.

Within its domain of application, TAME solves most of these problems. In particular, it provides templates to organize specifications of automata, high level proof steps designed to make proof structures more understandable, and supporting theories appropriate to the domain. The third author's experience with TAME supports our belief that the learning curve for TAME is much less steep than that for the direct use of PVS. TAME proofs tend to be fairly robust because they use high level proof steps that do not depend on details of the sequent in the current proof goal. (By contrast, such dependence on details is present in several places in the PVS proof in the *TIP* case study in Section 4.) In addition, TAME proofs are usually easy to modify when a change in a specification requires some changes in a proof.

Miller (1998) also notes that productivity in the AAMP5 project required the same individuals to serve as both domain experts and PVS experts. Because TAME proofs can be understood separately from PVS, we believe that TAME can provide a way to allow domain experts to understand the results of a verification without becoming PVS experts, and to communicate high-level proof outlines to PVS (or TAME) experts that can be easily checked in TAME. This has been demonstrated to some extent by our previous experience with TAME and by the third author's experience with the RPC-Memory example.

Thus, we believe that specialized interfaces such as TAME can solve many of the problems associated with introducing the use of theorem proving into industrial practice. This is consistent with the point of view of Crow and Di Vito (1998), who state:

> Applying formal methods "right out of the box" is difficult. Tailoring the methods to the application at hand is both necessary and desirable.

As noted in Section 3, TAME is based on template specifications for given system models, standard supporting theories, and special strategies to implement reasoning steps appropriate to the models. By providing an appropriate automatic specification translator, a specialized interface can also allow developers to create specifications in an environment familiar to them. For TAME, such a translator has been developed for specifications created in the SCR toolset (Archer et al., 1998). We believe that the methods used to create TAME can be followed to create specialized interfaces in other application domains. In fact, similar methods were used to some extent in the other PVS-based proof assistants discussed in Section 7. An open question is whether specialized interfaces analogous to TAME can be developed that will address the needs of practitioners.

Appendix A. The I/O automaton *TIP* from Devillers et al. (2000)

TIP

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

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** $mq[e] := \mathsf{append}(child[e^{-1}], mq[e])$

Appendix B. *TIP* specified using the TAME template: The theory tip_decls[4]

tip_**decls: THEORY**

BEGIN
timed_auto_lib: LIBRARY = "../timed_auto_lib"
IMPORTING timed_auto_lib@time_thy
IMPORTING timed_auto_lib@bool_rewrites
IMPORTING timed_auto_lib@list_rewrites
Vertices: TYPE+;
Link: TYPE=[Vertices, Vertices];
Edges: TYPE = {l:Link | NOT(proj_1(l)=proj_2(l))};

```
Bool : TYPE = boolean;
BoolStar: TYPE = list[Bool];
source(e:Edges): Vertices = proj_1(e);
target(e:Edges): Vertices = proj_2(e);
reverse_edge(e:Edges): Edges = (target(e), source(e));
fromv(v:Vertices): TYPE = {e:Edges | source(e)=v};
tov(v:Vertices): TYPE = {e:Edges | target(e)=v};
adj(e,f:Edges): Bool = (source(e)=target(f) & NOT(reverse_edge(e)=f) );
path(e,f:Edges,n:nat): RECURSIVE Bool=
  IF n=0 THEN e=f
      ELSE (EXISTS (l:Edges): adj(e,l) & path(l,f,n − 1)) ENDIF
    measure n

const_facts: AXIOM = true;

actions: DATATYPE
 BEGIN
  nu(timeof:(fintime?)): nu?
  add_child(addE: Edges): add_child?
  children_known(childV: Vertices): children_known?
  ack(ackE: Edges): ack?
  resolve_contention(resE: Edges): resolve_contention?
  root(rootV: Vertices): root?
 END actions

MMTstates: TYPE = [# init: [Vertices − > Bool],
                     contention: [Vertices − > Bool],
                     root: [Vertices − > Bool],
                     child: [Edges − > Bool],
                     mq:[Edges − > BoolStar] #]

init(v:Vertices, s:states): Bool = init(basic(s))(v)
contention(v:Vertices, s:states): Bool = contention(basic(s))(v)
root(v:Vertices, s:states): Bool = root(basic(s))(v)
child(e:Edges, s:states): Bool = child(basic(s))(e)
mq(e:Edges, s:states): BoolStar = mq(basic(s))(e)

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

OKstate?(s:states): bool = true
enabled_general (a:actions, s:states):bool =
      now(s) >= first(s)(a) & now(s) <= last(s)(a);
enabled_specific (a:actions, s:states):bool =
   CASES a OF
      nu(delta_t): delta_t > zero
```

```
                & FORALL (a0:actions):
                      NOT(nu?(a0)) => now(s) + delta_t <= last(a0,s),
    add_child(e): init(target(e),s) & NOT(mq(e,s) = null),
    children_known(v): init(v,s) &
                  (FORALL (e:Edges)(f:tov(v)): child(e,s) OR child(f,s) OR e = f),
    ack(e): NOT(init(target(e),s)) & NOT(mq(e,s) = null),
    resolve_contention(e): contention(source(e),s) & contention(target(e),s),
    root(v): NOT(init(v,s)) & NOT(contention(v,s)) &
          NOT(root(v,s)) & (FORALL (e:tov(v)): child(e,s))
  ENDCASES

trans (a:actions, s:states):states =
  CASES a OF
    nu(delta_t): s WITH [now := now(s) + delta_t],
    add_child(e): IF NOT(mq(e,s) = null)
                THEN s WITH [basic := basic(s) WITH
                        [child := child(basic(s)) WITH [(e) := true],
                        mq := mq(basic(s)) WITH [(e) := cdr(mq(e,s))]]]
                ELSE s ENDIF,
    children_known(v): s WITH [basic := basic(s) WITH
                       [init := init(basic(s)) WITH [(v) := false],
                        mq := (LAMBDA (e:Edges):
                              (IF (source(e) = v)
                               THEN cons(child(reverse_edge(e),s),mq(e,s))
                               ELSE mq(e,s) ENDIF ))]],
    ack(e): IF NOT(mq(e,s) = null)
          THEN s WITH [basic := basic(s) WITH
                    [contention := contention(basic(s))
                                 WITH [(target(e)) := NOT(car(mq(e,s)))],
                     mq := mq(basic(s)) WITH [(e) := cdr(mq(e,s))]]]
          ELSE s ENDIF,
    resolve_contention(e): s WITH [basic := basic(s) WITH
                           [child := child(basic(s)) WITH [(e) := true],
                            contention := contention(basic(s)) WITH
                                [(target(e)) := false, (source(e)) := false]]],
    root(v): s WITH [basic := basic(s) WITH
              [root := root(basic(s)) WITH [(v) := true]]]
  ENDCASES
enabled (a:actions, s:states):bool =
 enabled_general(a,s) & enabled_specific(a,s) & OKstate?(trans(a,s));
```

```
start (s:states):bool =
   s = (# basic := basic(s) WITH [init:= LAMBDA(v:Vertices): true,
                                  contention:= LAMBDA(v:Vertices): false,
                                  root:= LAMBDA(v:Vertices): false,
                                  child:= LAMBDA(e:Edges): false,
                                  mq:= LAMBDA(e:Edges): null],
         now := zero,
         first := (LAMBDA (a:actions): zero),
         last := (LAMBDA (a:actions): infinity) #);

   IMPORTING timed_auto_lib@machine[states, actions, enabled, trans, start]
END tip_decls
```

Appendix C. Some *TIP* invariants from Devillers et al. (2000), and in TAME

2. *If a node is in the initial stage then its outgoing links are empty.*

$I_2(e) \equiv init[\mathsf{source}(e)] \rightarrow mq[e]{=}\mathsf{empty}$

```
Inv_2(s:states): bool =
      (FORALL (e:Edges): init(source(e),s) ⇒ mq(e,s) = null);
```

4. *If a node has left the initial stage then all links, or all links but one, are child links.*

$I_4(e, f, v) \equiv$

$\mathsf{target}(e){=}\mathsf{target}(f){=}v \wedge e \neq f \rightarrow init[v] \vee child[e] \vee child(f)$

```
Inv_4(s:states): bool =
      (FORALL (e,f:Edges, v:Vertices):
          (target(e) = v AND target(f) = v AND NOT(e = f))
             ⇒ (init(v,s) OR child(e,s) OR child(f,s)));
```

5. *Each link contains at most one message at a time.*

$I_5(e) \equiv \mathsf{length}(mq[e]) \leq 1$

```
Inv_5(s:states): bool = (FORALL (e:Edges): length(mq(e,s)) ≤ 1);
```

6. *If a node is in the initial stage, then none of its neighbors is involved in root contention.*

$I_6(e) \equiv init[\mathsf{source}(e)] \rightarrow \neg contention[\mathsf{target}(e)]$

```
Inv_6(s:states): bool =
      (FORALL (e:Edges):
          init(source(e),s) ⇒ NOT(contention(target(e),s)));
```

7. *Child links are empty.*

$I_7(e) \equiv child[e] \rightarrow mq[e] = \mathsf{empty}$

```
Inv_7(s:states): bool =
      (FORALL (e:Edges): child(e,s) ⇒ mq(e,s) = null);
```

8. *If a node is involved in root contention, then all its incoming links are empty.*

$I_8(e) \equiv contention[\mathsf{target}(e)] \rightarrow mq[e] = \mathsf{empty}$

```
Inv_8(s:states): bool =
      (FORALL (e:Edges): contention(target(e),s) ⇒ mq(e,s) = null);
```

15.[5] *There is at most one node for which all incoming links are child links.*

$$I_{15} \equiv (\exists v \forall e \in \mathsf{to}(v) : \mathit{child}[e]) \rightarrow (\exists! v \forall e \in \mathsf{to}(v) : \mathit{child}[e])$$

```
Inv_15(s:states): bool =
    (EXISTS (v:Vertices): (FORALL (e:tov(v)):child(e,s))) ⇒
      ((EXISTS (v:Vertices): (FORALL (e:tov(v)):child(e,s)))
        AND
       (FORALL (v,w:Vertices):
                 ((FORALL (e:tov(v)):child(e,s)) AND
                  (FORALL (e:tov(w)):child(e,s)) ⇒ v = w)))
```

Acknowledgments

We thank Marco Devillers, Judi Romijn, and Oleg Cheiner for helpful discussions. We especially thank Judi Romijn for insightful comments on an earlier version of this paper. We also thank the anonymous referees and our colleagues Ramesh Bharadwaj, Ralph Jeffords, James Kirby, and Elizabeth Leonard for insightful comments that helped us improve the paper.

Notes

1. Although *Imp* does involve timing information, the time step action is not the global time step used in LV timed automata. Therefore, we modeled *Imp* as an I/O automaton and its time step as just another I/O automaton action.
2. In figure 2 and elsewhere, a name in bold capital letters denotes a TAME strategy.
3. GRIND fails to terminate on one of the proofs and needs to be helped by APPLY-EXTENSIONALITY in another. These complications are probably due to a PVS bug.
4. The fixed parts of the TAME template are shown in bold.
5. We have dropped the argument v to I_{15}.

References

Alborghetti, A., Gargantini, A., and Morzenti, A. 1997. 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., pp. 211–226. Springer-Verlag.

Archer, M. 2000. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):139–181.

Archer, M. 2002. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Informal Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, OR.

Archer, M. and Heitmeyer, C. 1996. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp.(RTAS'96)*, pp. 192–203. IEEE Computer Society Press.

Archer, M. and Heitmeyer, C. 1997a. 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., pp. 33–48. Springer-Verlag.

Archer, M. and Heitmeyer, C. 1997b. 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., pp. 171–185. Springer-Verlag.

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

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

Butler, R.W. 1996. An introduction to requirements capture using PVS: Specification of a simple autopilot, NASA Technical Memorandum 110255, NASA Langley Research Center.

Butler, R.W., Caldwell, J.L., Carreño, V.A., Holloway, C.M., Miner, P.S., and Di Vito, B.L. 1995. NASA Langley's research and technology-transfer program in formal methods. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS'95)*, Gaithersburg, MD, pp. 135–149. IEEE Computer Society Press.

Clarke, E.M., Emerson, E.A., and Sistla, A.P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263.

Crow, J. and Di Vito, B.L. 1996. Formalizing space shuttle software requirements. In *Proc. First ACM Workshop on Formal Methods in Software Practice (FMSP'96)*, San Diego, CA, pp. 40–48.

Crow, J. and Di Vito, B.L. 1998. Formalizing space shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332.

Devillers, M. 1997. Verification of a tree-identify protocol. Available at URL http://www.cs.kun.nl/~marcod/1394.html.

Devillers, M. 1999. Private communication.

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

Easterbrook, S. and Callahan, J. 1997. Formal methods for V & V of partial specifications. In *Proc. 3rd Intern. Symposium on Requirements Engineering (RE '97)*, Annapolis, MD.

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

Gargantini, A. and Morzenti, A. 2001. Automated deductive requirements analysis of critical systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):255–307.

Garland, S.J. and Lynch, N.A. 1998. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Draft. MIT Laboratory for Computer Science.

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

Kellomaki, P. 1997. Mechanical verification of invariant properties of DisCo specifications. Ph.D. Thesis, Tampere University of Technology, Finland.

Lamport, L. 1993. How to write a proof. Technical Report, Digital Equipment Corp., System Research Center, Research Report 94.

Lincoln, P. 1998. Private communication.

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

Lynch, N. and Tuttle, M. 1989. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

Lynch, N. and Vaandrager, F. 1995. Forward and backward simulations—Part I: Untimed systems. *Information and Computation*, 121(2):214–233.

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

Miller, S. 1998. The industrial use of formal methods: Was Darwin right? In *Proc. 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, pp. 74–82.

Miller, S. and Srivas, M. 1995. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL.

Müller, O. 1998. A verification environment for I/O automata based on formalized meta-theory. Ph.D. Thesis, Technische Universität München.

Owre, S., Shankar, N., Rushby, J.M., and Stringer-Calvert, D.W.J. 1999. *PVS System Guide*. Menlo Park, CA, Computer Science Laboratory, SRI International.

Romijn, J. 1996. 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., pp. 437–476. Springer-Verlag. Addendum. (URL http://www.cs.kun.nl/~judi/papers/dagstuhl_proofs.ps.gz).

Rudnicki, P. and Trybulec, A. 1996. A note on "How to Write a Proof". Technical Report, University of Alberta, Number TR96-08.

Skakkebaek, J. and Shankar, N. 1994. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of. Lect. Notes in Comp. Sci., Springer-Verlag.