NRL Release Number 00-1221.1-0443

Applying TAME to I/O Automata: A User's Perspective*

Elvinia Riccobene[1], Myla Archer[2], and Constance Heitmeyer[2]

[1] Dipartimento di Matematica, Università di Catania,
Viale A.Doria 6, I-95125 Catania (Italy)
riccobene@cs.unict.it

[2] Naval Research Laboratory (Code 5546), Washington DC 20375
{archer,heitmeyer}@itd.nrl.navy.mil

Abstract. Mechanical theorem provers have been shown to expose proof errors, some of them serious, that humans miss. Mechanical provers will be applied more widely if they are easier to use. The tool TAME (Timed Automata Modeling Environment) provides an interface to the prover PVS to simplify specifying and proving properties of automata models. Originally designed for reasoning about Lynch-Vaandrager (LV) timed automata, TAME has since been adapted to other automata models. This paper shows how TAME can be used to specify and verify properties of I/O automata, a class of untimed automata. It also describes the experiences of a new TAME user (the first author) who used TAME to check Lamport-style hand proofs of invariants for two applications: Romijn's solution to the RPC-Memory Problem [21, 20] and the verification by Devillers et al. of the tree identify phase of the IEEE 1394 bus protocol [9, 8]. For the latter application, the TAME mechanization of the hand proofs [8] is compared with the more direct PVS proofs [9]. Improvements to TAME in response to user feedback are discussed.

1 Introduction

When done by hand, even the most carefully crafted formal specifications and proofs may contain inconsistencies and other errors, some of them serious. Mechanically supported formal methods, such as mechanical provers, can expose many errors that humans miss (see, e.g., [5, 12]). Not only can mechanical checking of a formal specification and its properties confirm the correctness of the properties, it can also reduce the human effort needed to expose defects in the specification and in the statements of properties while the specification and proofs are under development [9].

A major barrier to more widespread use of mechanical proof methods in industry is the overhead and complexity of using a mechanical theorem prover. One must first encode a specification in the language of the prover and then must establish properties of the specification in the logic of the prover, using proof steps that often do not correspond to proof steps natural for a human. To date, most mechanical verification has been done by researchers with highly detailed knowledge of a mechanical prover, such as PVS [23]. The frequency of mechanical verification can be expected to increase if tools such as PVS are easier, and thus more cost-effective, to use.

The tool TAME (Timed Automata Modeling Environment) [3, 5, 4, 6] provides an interface that simplifies specifying and proving properties of Lynch-Vaandrager (LV) timed automata [18] using PVS. TAME is designed to make mechanically supported formal methods (such as PVS) easier to use by simplifying the encoding of an automaton specification, by supporting proofs of

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

properties through natural proof steps, and by presenting saved proofs in human-understandable form. A major goal of TAME is to facilitate the checking of hand proofs with PVS.

The TAME approach to mechanized theorem proving may be contrasted with the approach of Devillers, Griffioen, Romijn, and Vaandrager, whose PVS proofs for the properties in [9] do not follow the hand proofs given in [8], but were created independently. For this reason, Devillers et al. question the utility of the hand proofs. In contrast, we view the existence of hand proofs—or, at least, human-understandable mechanized proofs—as useful, because such proofs explain why a specification has a given property. A property that holds for unexpected reasons raises the question of whether the specification actually captures the intended behavior. Mechanized proofs that follow the structure of the hand proofs confirm not only the correctness of the properties but the correctness of the given reasons that the properties hold.

This paper makes four contributions. First, we demonstrate by example how TAME can be applied to I/O automata to check Lamport-style hand proofs. Originally designed to specify and verify properties of LV timed automata, TAME has been adapted to work with other automata models, including the (untimed) I/O automata model and the automaton model that underlies SCR [6]. Previous proofs checked with TAME were natural language (but not Lamport-style) hand proofs. Second, the paper describes the positive experience of a new user (the first author), who used TAME to check the Lamport-style proofs of invariant properties for two applications: Romijn's solution of the RPC-Memory Problem [21, 20], and the verification by Devillers et al. of the tree identify phase of the IEEE 1394 bus protocol [9, 8]. Third, the paper describes improvements to TAME that resulted from the first author's feedback. Finally, the paper compares two approaches to using PVS to prove properties of an I/O automata model: one approach uses TAME, while the second uses PVS directly.

Section 2 briefly reviews PVS, I/O automata, and TAME. Based on the first author's use of TAME and the use of PVS described in [9] to prove properties of the *TIP* (*Tree-Identify Protocol*) specification, Section 3 contrasts the TAME approach for specifying and proving properties of I/O automata with the direct use of PVS. Section 4 discusses how specifications of I/O automata and Lamport-style proofs of invariant properties can be translated into TAME and presents the results of the first author's application of TAME to *TIP* as well as to several I/O automata from [21]. Section 5 discusses the first author's experience with TAME and resulting improvements to TAME, Section 6 describes related work, and Section 7 presents some conclusions and our future plans.

2 Background

PVS. PVS [23] is a higher order logic specification and verification environment developed by SRI. Proof steps in PVS are either *primitive* steps or *strategies* defined using primitive steps, applicative Lisp code, and other strategies. Strategies may be built-in or user-defined. PVS's support for user-defined strategies allows the construction of specialized prover interfaces, such as TAME, on top of PVS. TAME exploits both the PVS support for user-defined strategies and recent enhancements to PVS, including 1) support for labeling formulae appearing in proof goals and for documenting proofs with comments, 2) some new finer-grained primitive steps, and 3) new access functions and documentation

that allow strategies to incorporate computations based on the internal data structures maintained by PVS.

One important PVS feature is its very rich strong type system. Because PVS permits the user to define subtypes using arbitrary predicates, the type correctness of PVS specifications is undecidable, and the PVS typechecker typically generates several type correctness conditions (TCCs) that must be verified before a specification is "type correct". PVS has a general strategy that proves many TCCs automatically; however, the user is sometimes obliged to supply a proof. When a TCC cannot be proved, usually some subtle inconsistency in the specification requires correction.

The I/O Automata Model. In the I/O automata model [17], a system is described as a set of I/O automata, interacting by means of common actions. For verification purposes, these interacting automata can be composed into a single automaton by combining corresponding output and input actions. Every I/O automaton is described by a set of states, some of which are initial states; a set of actions (input, output, and internal); and a transition relation coupling a state-action pair with another state. In a typical I/O automaton specification (see Appendix A), a state is an assignment of values to state variables. For deterministic automata, which include all the automata discussed in this paper, the transition relation can be described as a (partial) function that maps an action and an old state in which the action is enabled to a new state obtained by applying the effects of the action to the state variables of the old state. Three major classes of properties of automata are 1) state invariants, 2) simulation relations, and 3) properties of execution sequences. Proofs of both 1) and 2) have a standard structure, with a base case involving initial states and a case for each possible action, and hence are especially good targets for mechanization. The proof examples in this paper all involve state invariants and thus belong to class 1).

TAME. TAME provides a template for specifying automata, a set of standard theories, and a set of standard PVS strategies. The TAME template provides a standard structure for defining an automaton. Originally designed for specifying LV timed automata, this template is easily adapted to specifying I/O automata. To define either a timed or untimed automaton, the user provides the information indicated in Figure 1. The standard strategies of TAME are designed to support

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

Fig. 1. Information required in the TAME template.

mechanical reasoning about automata using proof steps that mimic human proof steps. These strategies are based on the set of standard theories, certain template conventions, and a set of special definitions, auxiliary local theories, and local strategies that can be generated from a template instantiation. Reference [2] describes the TAME user strategies in detail.

Specifications of I/O automata in the style used in [9] and [21] can be easily translated into TAME specifications. The definitions of the (non-time-passage) actions of the I/O automaton provide the names and argument types needed for their TAME declarations, preconditions and effects. 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. The initial state information for the I/O automaton is translated into the initial state predicate `start` of the TAME specification. Finally, any constants defined for the I/O automaton can be declared in the TAME specification, and any predicates relating the constants can be included in the TAME specification in the axiom `const_facts`. When an I/O automaton is defined as the composition of two or more other I/O automata (this happens with some of the automata in the RPC-Memory example), the information extracted from the individual automaton descriptions can be combined to produce a single TAME specification in a (usually) straightforward way.

Hand proofs of invariant properties of automata typically contain a limited variety of proof steps. Figure 2 shows the most common proof steps and their corresponding TAME strategies. TAME strategies also exist for many steps needed less frequently than the six listed in Figure 2.

Proof Step	TAME Strategy	Remarks
Break down into base case and induction (i.e., action) cases	AUTO_INDUCT	For starting an induction proof
Appeal to precondition of an action	APPLY_SPECIFIC_PRECOND	Used, when needed, in induction cases
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)

Fig. 2. Common proof steps for invariant proofs, and their TAME strategies.

3 Comparing TAME with PVS for I/O Automata

For the *TIP* example, reference [9] describes the direct use of PVS to mechanize the proofs of properties. Below, we contrast the TAME approach with this approach. For brevity, we refer to "TAME" specifications and proofs versus "PVS" specifications and proofs.

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 (eight or nine) levels. Moreover, the

organization of the import structure is at least partly problem-specific. In contrast, the TAME specification of *TIP* is contained in a single automaton-specific theory that imports instantiations of a small collection of generic theories, and is thus more easily understood as a whole. There are additional, automaton-specific theories associated with the TAME specification that supply rewrites to the generic TAME strategies. However, these theories can be derived in a standard, automatable way from the main theory for any given 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 some instances, some information from the precondition is needed in the definition of the effect for the definition to pass typechecking. However, when possible, separating the precondition and effect has an advantage: it allows one to determine just when the precondition is important in an induction step.

Because PVS lacks support for defining a general automaton type and for passing theory parameters to theories, a completely general definition of refinement is impossible 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, using the most convenient general form that can currently be provided with PVS[1], and in this respect, has an advantage over the TAME specification. The generic theories supporting the definition of refinement in the PVS specification could almost certainly be adapted for use with a new TAME "refinement" template. Instead, a future version of TAME will use the support for theory parameters to be provided in a future version of PVS [15] to support a generic refinement template.

The PVS encoding of state invariant lemmas, which is slightly different from the TAME encoding, has two lemmas associated with most invariants: the first states that the invariant holds in start states and is preserved by transitions and the second (usually 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 follows from other invariants—only the second form is given. The TAME encoding of state invariant lemmas uses only the second of the forms used in the PVS encoding. For proofs requiring induction, the strategy AUTO_INDUCT first reduces this form to the first PVS encoding form and then performs many of the standard initial proof steps.

The most dramatic difference between the PVS approach of [9] and the TAME approach is in the proofs of invariants. The TAME proofs are much shorter, and the significance of proof branches and individual proofs steps is much clearer. Moreover, the TAME proofs correspond in a very clear way to the hand proofs in [8] (see Section 4.1). This contrast between the PVS proofs and the TAME proofs is illustrated by Figures 3 and 4, which show corresponding TAME and PVS proofs of *TIP* Invariant I_5. While the TAME proof execution times in the *TIP* example average about three times as long as those of the corresponding PVS proofs (e.g., I_6, I_7, and I_8 combined took the longest time, 37 seconds for TAME vs 15 seconds for PVS[2]), the relative simplicity and

[1] This definition makes use of a parameterized automaton type defined in a theory parameterized by the action and state types.

[2] These times are for PVS 2.2 on an UltraSPARC-II.

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

;;; Proof lemma_5-like-hand for formula tip_invariants.lemma_5
(""
 (AUTO_INDUCT)
 (("1" ;;Case add_child(addE_action)
   (APPLY_SPECIFIC_PRECOND)
   (SUPPOSE "e_theorem = addE_action")
   (("1" ;;Suppose e_theorem = addE_action
     (TRY_SIMP))
    ("2" ;;Suppose not [e_theorem = addE_action]
     (TRY_SIMP))))
  ("2" ;;Case children_known(childV_action)
   (SUPPOSE "source(e_theorem) = childV_action")
   (("1" ;;Suppose source(e_theorem) = childV_action
     (APPLY_SPECIFIC_PRECOND)
     (APPLY_INV_LEMMA "2" "e_theorem")
     (TRY_SIMP))
    ("2" ;;Suppose not [source(e_theorem) = childV_action]
     (TRY_SIMP))))
  ("3" ;;Case ack(ackE_action)
   (SUPPOSE "e_theorem = ackE_action")
   (("1" ;;Suppose e_theorem = ackE_action
     (APPLY_SPECIFIC_PRECOND)
     (TRY_SIMP))
    ("2" ;;Suppose not [e_theorem = ackE_action]
     (TRY_SIMP))))))
```

Fig. 3. TAME Proof (nonverbose) of *TIP* Invariant I_5

```
INV_5((s: states)): bool = (FORALL (e: E): length(mq(s)(e)) <= 1)

;;; Proof INV_5_inv-1 for formula invariant.INV_5_inv

(""
(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)
 (("1"
  (SKOSIMP*)
  (EXPAND "steps")
  (EXPAND "A_C_step")
  (PROP)
  (REPLACE –2 :HIDE? T)
  (EXPAND "INV_5")
  (SKOSIMP*)
  (LIFT-IF)
  (PROP)
  (("1"
   (INST?)
   (REPLACE –1 :HIDE? T)
   (HIDE –1)
   (HIDE 2)
   (EXPAND "tl")
   (EXPAND "length")
   (ASSERT)
   (LIFT-IF)
   (PROP)
   (ASSERT)
   (EXPAND "length")
   (LIFT-IF)
   (PROP)
   (("1" (ASSERT))
    ("2" (ASSERT))
    ("3" (ASSERT))))
  ("2" (INST?))))
 ("2"
  (SKOSIMP*)
  (EXPAND "steps")
  (EXPAND "ACK_step")
  (PROP)
  (REPLACE –1 :HIDE? T)
  (EXPAND "INV_5")
  (SKOSIMP*)
  (LIFT-IF)
  (PROP)
  (("1"
   (INST?)
   (REPLACE –1 2 3)
   (EXPAND "tl")
   (EXPAND "length")
   (LIFT-IF)
   (PROP)
   (("1" (ASSERT))
    ("2" (ASSERT)
         (EXPAND "length")
         (LIFT-IF)
         (ASSERT))))
  ("2" (INST?))))
 ("3"
  (SKOSIMP*)
  (EXPAND "steps")
  (EXPAND "C_K_step")
  (PROP)
  (REPLACE –3 :HIDE? T)
  (EXPAND "INV_5")
  (SKOSIMP*)
  (LIFT-IF)
  (PROP)
  (("1"
   (INST?)
   (("1"
    (ASSERT)
    (USE "INV_2_reach")
    (EXPAND "INV_2")
    (INST?)
    (ASSERT)
    (INST?)
    (("1"
     (EXPAND "member")
     (EXPAND "froms")
     (ASSERT)
     (HIDE –2 –3 –4)
     (INST?)
     (EXPAND "append")
     (EXPAND "length" 1)
     (EXPAND "length")
     (ASSERT))
    ("2"
     (EXPAND "tos")
     (EXPAND "target")
     (EXPAND "inv")
     (EXPAND "member")
     (EXPAND "froms")
     (PROPAX))))
   ("2"
    (EXPAND "tos")
    (EXPAND "member")
    (EXPAND "froms")
    (EXPAND "inv")
    (EXPAND "target")
    (PROPAX))))
  ("2" (HIDE –2) (INST?))))
 ("4"
  (SKOSIMP*)
  (EXPAND "steps")
  (EXPAND "R_C_step")
  (PROP)
  (REPLACE –3 :HIDE? T)
  (EXPAND "INV_5")
  (PROPAX))
 ("5"
  (SKOSIMP*)
  (EXPAND "steps")
  (EXPAND "ROOT_step")
  (PROP)
  (REPLACE –2 :HIDE? T)
  (EXPAND "INV_5")
  (PROPAX))))))
```

Fig. 4. PVS Proof of *TIP* Invariant I_5 by Devillers et al.

clarity of the TAME proofs strongly suggests that the human time needed to construct the proofs with TAME is several times shorter than that needed to construct proofs with the PVS-based approach of [9]. The PVS proofs clearly have repeating patterns; the TAME strategies take advantage of such repeating patterns to produce higher-level proof steps.

Although the TAME proofs for *TIP* attempt to follow the hand proofs very closely, avoiding some of the case breakdowns in the hand proofs often produces shorter TAME proofs. In addition to checking hand proofs, TAME has proved helpful in proof exploration and can also be used, without any formal hand proof, to test the user's ideas of whether (or why) a property holds.

4 Example I/O Automata, Properties, and Proofs

This section describes the results obtained by the first author in using TAME to mechanize the specifications and Lamport-style hand proofs [14] of invariant properties of several I/O automata models. These model were from two sources: the *Tree-Identify Protocol Specification* [9, 8] and the *RPC-Memory Specification* [21, 20]. In each case, we show how TAME was used to check the hand proofs, with attention to particular techniques used and problems encountered in creating the TAME specifications and mechanizing the proofs. Because TAME does not yet have specification or proof support for simulation proofs, in this exercise, only proofs of invariants were checked.

4.1 The Tree-Identify Protocol

Reference [9] describes and analyzes the IEEE 1394 high performance serial multimedia bus protocol. The major goal of the analysis is to verify the leader election algorithm, the core of the tree identify phase of the physical layer of the protocol. An I/O automaton model for the leader election algorithm provides the mathematical basis for the hand proof of the main property: "For an arbitrary tree topology, exactly one leader is elected". The tree identify protocol is only applied to a graph with a particular type of topology: the graph must be an undirected digraph (i.e., if it contains an edge e, then it also contains its reverse edge $reverse(e)$) without self-loops and with a tree-like topology.[3] As the algorithm proceeds, particular links (directed edges) between adjacent nodes are added to a directed spanning tree until the tree is complete; its root is then the "leader". At any point during execution of the algorithm, those edges that have been added to the spanning tree are known as *child* edges.

In [9], the algorithm is specified in terms of an I/O automaton *TIP*. Appendix A contains the original specification of *TIP* from [9]. A number of invariant properties (see Appendix B for some examples) are established for *TIP* and used to prove that *TIP* is a refinement of a more abstract automaton *SPEC*, which captures the required behavior. We focused on *TIP* and its invariants, due to our interest in automating the hand proofs of the invariants. The TAME translation of this specification was easily obtained from the I/O automaton specification using TAME's standard template [3, 2].

The first fifteen invariants from [9], I_1 through I_{15}, form a sequence establishing that *at most* one leader is elected by the *TIP* algorithm. Devillers has developed detailed Lamport-style hand proofs for these invariants [8]. Each hand proof is one to two pages in length. The full set of invariants from [9] includes two extra invariants, I_{16} and I_{17}, used in the proof that *at least* one leader is elected. All but the invariant I_{15} were proved using TAME and no other mathematical superstructure, except a small set of auxiliary lemmas describing the relationship between a link and its inverse link (needed for invariants I_{10}, I_{11}, I_{12} and I_{14}). These auxiliary lemmas were used to translate those steps in the hand proofs whose justification was "math".

Figures 5 and 6 show the correspondence between steps from a Lamport-style proof and TAME steps for invariant I_4. Figure 5 shows only that part, a single branch, of the hand proof that TAME found to be nontrivial; Figure 6 shows the complete TAME proof of I_4. In the hand proof, the values s and t represent the prestate and poststate in the induction step, and the values f and g are, respectively, the Skolem constants for the quantified variables e and f in I_4, which are automatically named `e_theorem` and `f_theorem` by TAME. The appeal "by IH" to the inductive hypothesis at step $< 3.1 >$ in the hand proof is handled automatically by TAME's AUTO_INDUCT strategy whenever, as in this case, the correct instantiation of its variables 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 that f and g are of type $to(v)$ is handled by proving

[3] That is, for each pair of vertices v, w, there is a unique sequence of vertices $v_0, v_1, ..., v_n$ 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.

<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)

Fig. 5. Single Nontrivial Branch of Lamport-style Proof of Invariant I_4

```
;;; Proof lemma_4-3 for formula tip_invariants.lemma_4
("" (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)
        ;;        &
        ;;    (FORALL (e: Edges):
        ;;        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" (TRY_SIMP)) ("2" (TRY_SIMP))))     < 3.2.3.2 >, < 3.2.3.3 >
    ("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 >
```

Fig. 6. Complete TAME Proof (verbose) of Invariant I_4

a TCC generated by PVS when the INST is done—this is accomplished by the proof step TRY_SIMP at "2" in the line right after the INST step. The effect of the action, the appeal to previous proof steps, and setting up invariant I_4 in the poststate as a proof goal are all handled automatically by the TAME strategies AUTO_INDUCT and TRY_SIMP.

Invariant I_{15} is especially important because it is the only invariant used in the proof that *TIP* is a refinement of *SPEC* [9]. I_{15} is also the only invariant that requires knowledge of the graph topology of the tree identify protocol network. Both the PVS proof of I_{15} described by Devillers et al. in [9] and the TAME proof of I_{15} required formalizing some of this knowledge. The (natural language) hand proof in [9] is an informal proof by contradiction, which the TAME proof resembles at a high level. Devillers et al. needed a few days to construct the PVS proof, because they took the time to specify a general PVS theory for acyclic finite strongly-connected digraphs. Our approach, which benefited from their experience, was more economical. The hand proof of I_{15} and the way it uses invariant I_{14} suggested that to prove I_{15} in TAME, we needed to formalize only the fact that the graph topology is connected.[4] The TAME proof of invariant

[4] The fact that the graph topology is tree-like is used only in a proof that was not mechanized by Devillers et al. [9]: the proof that at least one leader is elected.

I_{15} then uses two auxiliary invariants: (i) *Any adjacent link of a child link is also a child link*; and (ii) *Any link connecting to the source of a child link via a path of adjacent links is also a child link.* A link f is *adjacent* to a link e if f is an incoming link to the source of e, i.e., $target(f) = source(e)$. Invariant (ii) is proved by induction on the length of the path, using Invariant (i), a consequence of Invariant I_{14}. The hand proof of I_{15} in [9] relies on Invariants I_{14} and I_{11} plus the existence of a unique cycle-free path between any two vertices. Our corresponding proof in TAME uses Invariants (ii) and I_{11} plus the connectedness axiom.

4.2 RPC-Memory Specification Problem

The RPC-Memory problem, posed in 1994 by Broy and Lamport at the Dagstuhl Workshop on Reactive Systems, concerns the specification of a memory component and a remote procedure call (RPC) component for a distributed system and the implementation of both. The I/O automata solution in [21] contains approximately twenty I/O automata and 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. Hand proofs of these properties are provided in [20]. Almost all proofs of state invariants are in the Lamport style. Since our goal was to automate these proofs, we 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, connected through an appropriate front end for the RPC; and *Imp*, which models an implementation of a lossy version of the RPC, with timing information added.[5]

Few problems arose in the TAME mechanization of the *Memory**, *MemoryImp*, and *Imp* specifications and the detailed proofs of their properties from [21, 20]. Nevertheless, the mechanization did expose some incompleteness and inconsistency in the specifications and some missing and incorrect details in the proofs. For example, 1) the intended types of certain constants are unclear, 2) there is a type inconsistency in the definition and use of one function, and 3) a missing detail in the proof of a *Memory** invariant required the identification and proof of an auxiliary invariant lemma in TAME.

Aspects of the TAME mechanization required some creativity. In particular, a few hand proofs were only sketched, so we needed to establish the details of the corresponding TAME proofs, including some needed auxiliary invariants. Further, the encoding of *MemoryImp* and *Imp* using the TAME specification template required renaming state variables to avoid name clashes and careful definition of the union types and subtype recognizers needed to define the composition of certain transitions from separate components.

Despite the above complications, the specification and proofs for *Memory**, *MemoryImp*, and *Imp* in TAME were straightforward. The hand proofs for the *Memory** invariants were easily checked by applying only four TAME steps: AUTO_INDUCT, APPLY_SPECIFIC_PRECOND, APPLY_INV_LEMMA, and TRY_SIMP. The example *MemoryImp* led to improvements in some existing TAME strategies, and the addition of two new ones: INST_IN and SKOLEM_IN

[5] Although *Imp* does involve timing information, this information is encoded using a set of independent clocks instead of a universal clock. Instead of using the time features of the timed automaton template of TAME, we treated the time step action for *Imp* as an ordinary I/O automaton action.

(see Section 5). Given these improvements, the TAME proofs for *MemoryImp* used only the four preceding proof steps plus SUPPOSE, DIRECT_PROOF, INST_IN, SKOLEM_IN, and the PVS proof step EXPAND. Finally, the TAME proofs for *Imp* were done using only the four preceding proof steps plus SUPPOSE, DIRECT_PROOF, and the PVS proof step INST.

5 Discussion

This section discusses the first author's experience in using TAME to specify and prove properties of I/O automata, including the time required, the problems that had to be overcome to extract TAME specifications from I/O automata models, and the effort required to prove state invariants with TAME. Our goal was to understand the difficulty of learning how to use TAME, for a user without any previous knowledge of PVS, I/O automata, or TAME. This section also discusses enhancements made to TAME as a result of the first author's feedback.

Specification in TAME. The first TAME specifications that the first author developed were of the I/O automata *Memory**, *MemoryImp*, and *Imp* from the RPC-Memory problem [21]. Understanding previous TAME specifications required about one week. Specifying *Memory** in TAME required two additional days. The built-in templates were useful for understanding what information about the model was needed and how to organize this information. Defining the invariants of the *Memory** model in TAME was easy, given the predefined template for specifying invariants. The definition of the auxiliary theories (which currently must be developed by hand) took extra time but was straightforward.

After this initial experience, the specification of TAME theories for *MemoryImp* required only a few days. The difficulty in this case was learning how to use TAME to combine the three component I/O automata (*RPC*, *ClerkR* and *RMemory′*) and their corresponding input and output actions into a single automaton specification. A complication was handling certain input actions having different definitions depending on the type of the parameter to the action. The parameters of such actions had to be represented as union types using the PVS datatype construct. The first author's initial specification of *MemoryImp* had some unprovable TCCs connected with types specified in [21] in terms of membership, subtyping, and distinction between members. Although the TCCs could have been proved by including axioms specifying these relationships, one lesson learned was that redefining such types in terms of a supertype of related types using the PVS datatype construct both permits the TCCs to be handled automatically by PVS and avoids the possible creation of inconsistent axioms.

The specification of *Imp* was straightforward and almost problem-free. The only difficulty was determining that the "for-do" statement in the definition of the time-step action TIME needed to be formalized using PVS's LAMBDA construct. Specification of the model and proofs of the invariants in TAME took approximately three days, with some of the time used to develop the auxiliary theories. After the experience with the more complex RPC-Memory specifications, representing the specification for *TIP* in TAME was very simple.

For a new user, the templates for the automaton specification and for the invariant lemma definitions proved extremely useful. The specification template provides a starting point for specifying an automaton in PVS without a deep knowledge of PVS; the user simply fills in the actions, the names and types of state variables, the action preconditions and action effects, and the start states, and provides a few auxiliary definitions of types and constants. The user

must also generate some auxiliary functions, strategies, and theories needed to support the TAME strategies. This process, which will eventually be automated, is tedious but straightforward to do by hand, once the specification template is filled in.

Proofs using TAME. Once the way to apply the TAME strategies and the nature of their correspondence with the steps in the hand proofs was understood, the TAME proofs of the three invariant properties for *Memory** (and the fourth, auxiliary property required to prove the first) were easily completed in a few hours. Proving the twelve invariants of *MemoryImp* was the most difficult part, for two reasons: 1) the lack, in some cases, such as the corollaries, of complete and precise hand proofs, and 2) the inability of the strategies to capture (or at least to simply capture) a few of the proof steps, such as the one requiring the application of an invariant property to a poststate. Once the TAME strategies were improved (see Section 4.2 and below), proving these invariants required a little over two weeks. As with specifying *Imp*, proving the four properties of *Imp* in TAME was straightforward and almost problem-free.

After experience with the more complex RPC-Memory examples, and because the formal and clear hand proofs in [8] were almost immediately reproducible in TAME, obtaining TAME proofs for *TIP* was very simple. The *TIP* specification and proofs of the first fourteen invariants for *TIP* took three days. Due to the need to discover and formalize the minimal knowledge needed about the graph structure, approximately two additional days were required to prove the last invariant, I_{15}.

In proving the invariants of the automata considered in this paper, TAME was used only twice (on invariants I_{16} and I_{17} of *TIP*) to construct proofs without any handwritten proofs as guidance.[6] The proof of I_{16} in TAME required no thought; we simply applied AUTO_INDUCT, APPLY_SPECIFIC_PRECOND, and TRY_SIMP. The proof of I_{17} used these steps, plus repeated applications of APPLY_INV_LEMMA and APPLY_LEMMA. Finding the right lemmas to apply required a few hours; nevertheless, TAME saved the user from micromanagement of the proof. TAME was also used to mechanize several simply sketched hand proofs, such as Corollary 30 of *MemoryImp*. After studying the hand proofs, we formulated an auxiliary invariant lemma that could be proved with TAME and was sufficient to support the TAME proof of Corollary 30. Thus, in all cases in which the proof of a lemma was incomplete, using TAME resulted in a structured proof for the lemma.

TAME Enhancements Due to User Feedback. Feedback from the first author led to improvements in TAME that include an improved template convention, improvements to existing strategies, and the addition of new strategies.

The strategy AUTO_INDUCT discharges the base (start state) case in an induction proof automatically in most cases, provided that the start state predicate `start(s)` from the TAME template is given as an equality defining `s`. The *Memory** example led to a new improved method of formulating `start` which in effect subsumes the old one. Previously, `start(s)` has been expressed as an equality between the argument `s` and an explicit record value (a single start state). In the new method, `start(s)` is an equality between `s` and `s` with its time-related components assigned the standard initial values and its `basic` com-

[6] The proofs given for I_{16} and I_{17} in [8] are simply "done in PVS".

ponent (which covers non-time-related state variables) partially or fully updated. This formulation matches the initialization of the I/O model for *Memory**, which defines the initial values of only three of the five state variables.

To facilitate a faithful translation of some of the proofs from the example *MemoryImp* into TAME, improvements and additions to the TAME strategies were needed. For example, in the proof of Lemma 35, an invariant lemma is applied in an induction step to the poststate, rather than, as is more common, to the prestate. TAME previously represented the poststate as `trans(a,prestate)`, where `a` is the action of the induction step, and recorded among the hypotheses that it is a reachable state, facilitating application of an invariant lemma to the poststate. However, applying an invariant lemma to `trans(a,prestate)` involves not only using a complex term (containing the current action `a` as an argument) but, once the invariant lemma is applied, expanding the function `trans`. Improvements to AUTO_INDUCT and APPLY_INV_LEMMA now hide this complexity from the user, replacing the term `trans(a,prestate)` with the name `poststate`, and allowing the user to simply invoke the invariant lemma on `poststate` plus any other arguments to the lemma.

The proofs of Lemmas 27 and 29 demonstrated the difficulty of following the steps in a hand proof when one cannot instantiate or skolemize with respect to embedded quantifiers in PVS. To address this problem, the strategies 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 normal PVS proof steps INST and SKOLEM. In some cases (as unfortunately happens for Lemma 35), this leads to some wasteful proof branching, but in many cases, this works well. The proof of Lemma 35 also inspired a general improvement that allows TRY_SIMP to cover even more "obvious" general reasoning steps.

6 Related Work

An increasing number of proof assistants, including assistants for the Duration Calculus [24], for the TRIO logic [1], and for proving invariant properties of DisCo specifications [13], use PVS as the underlying prover. The Duration Calculus and TRIO assistants support proofs using steps from particular 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; the 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 [9], reference [16] describes how the Larch theorem prover LP was used to prove properties of several protocols specified as LV timed automata, and reference [19] describes a verification environment for I/O automata based on Isabelle; like [9], both include simulation proofs as well as proofs of invariants. In addition, [19] 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 [19], 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 [11]. This toolset will eventually include a similar translator to PVS that is being developed by Devillers and Vaandrager; a prototype now exists [10]. TAME currently has a prototype translator from specifications in the SCR language to TAME specifications [6], and an automatic translator from IOA specifications is planned.

7 Conclusions and Future Work

The work described in this paper has provided valuable user feedback about the utility of the TAME templates and strategies and an opportunity to compare the application of PVS to a particular problem, the *TIP* verification, in two ways: directly, or by using TAME. User feedback helped us improve TAME by refining existing strategies, by adding new strategies, and by improving the default template for the start state predicate. The results of the *TIP* comparison clearly demonstrate the advantages of using TAME for specifying and proving invariant properties of I/O automata. These advantages of TAME have recently also been noted by another new TAME user [7].

This was the first time TAME was used to mechanize Lamport-style proofs. Constructing TAME proofs that very closely follow hand proofs presented in this style was generally straightforward. However, like Rudnicki and Trybulec [22], we found that Lamport-style proofs are still informal and thus may have incorrect or missing details. In addition, as illustrated in Figures 5 and 6, many details included in Lamport-style proofs need not be made explicit in TAME proofs; some micro-steps in proofs are directly managed by the TAME strategies, which employ the PVS decision procedures, along with some rewriting and forward chaining, to automatically handle most low-level proof steps. Some hand proofs can be shortened even further in TAME. In fact, the hand proofs mechanized in TAME usually make clear which facts are needed in proving each result. Use of the TAME step TRY_SIMP avoids many explicit uses of "case distinction", provided all relevant facts for the proof branches for the distinct cases are first provided. However, simplified TAME proofs can sometimes obscure details of human reasoning that are important to understanding a proof.

Future plans for TAME are to add support for proofs of forward simulation, backward simulation, and refinement relations between automata specified with the TAME specification template, to provide an interface for generating template instantiations and their auxiliary theories and strategies from minimal user input, and to generate natural language explanations of TAME proofs.

Acknowledgements

We thank Marco Devillers, Judi Romijn, and Oleg Cheiner for helpful discussions. We especially thank Judi Romijn for comments on an early version of this paper.

References

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

2. M. Archer. Tools for simplifying proofs of properties of timed automata: The TAME template, theories, and strategies. Technical Report NRL/MR/5540–99-8359, NRL, Wash., DC, 1999.
3. M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*. IEEE Computer Society Press, 1996.
4. Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In *Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 33–48. Springer-Verlag, 1997.
5. Myla Archer and Constance Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.
6. Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers 1998*, Eindhoven, Netherlands, July 1998. Eindhoven Univ. of Technology.
7. Oleg Cheiner. Private communication. February, 1999.
8. M. Devillers. Verification of a tree-identity protocol. See the URL `http://www.cs.kun.nl/~marcod/1394.html`, 1997.
9. M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to IEEE 1394. *Formal Methods in System Design*. To appear.
10. Marco Devillers. Private communication. January, 1999.
11. S. J. Garland and N. A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Draft. MIT Lab. for Computer Sci., August, 1998.
12. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), November 1998.
13. Pertti Kellomaki. *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, Finland, November 1997.
14. L. Lamport. How to write a proof. Technical report, Digital Equipment Corp., System Research Center, February 1993. Research Report 94.
15. Patrick Lincoln. Private communication. July, 1998.
16. Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, MIT, June 1995.
17. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.
18. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lect. Notes in Comp. Sci.*, pages 397–446. Springer-Verlag, 1991.
19. Olaf Mueller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universitaet Muenchen, September 1998.
20. J. Romijn. Tackling the RPC-Memory Specification Problem with I/O automata. Addendum. URL `http://www.cwi.nl/~judi/papers/dagstuhl_proofs.ps.gz`.
21. J. Romijn. Tackling the RPC-Memory Specification Problem with I/O automata. In M. Broy, S. Merz, and K. Spies, editors, *Formal Systems Specification — The RPC-Memory Specification Case*, volume 1169 of *Lect. Notes in Comp. Sci.*, pages 437–476. Springer-Verlag, 1996.
22. P. Rudnicki and A. Trybulec. A note on "How to Write a Proof". In *Proc. 1992 Workshop on Types and Proofs for Programs*, June 1996.
23. N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.
24. J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault-Tolerant Systems*, Lect. Notes in Comp. Sci. Springer-Verlag, 1994.

A The I/O Automaton *TIP* from [9]

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

B Some Invariants of *TIP* from [9][7]

4. *If a node has left the initial stage then all links, or all links but one, are child links.*
$I_4(e, f, v) \equiv \mathsf{target}(e) = \mathsf{target}(f) = v \wedge e \neq f \to init[v] \vee child[e] \vee child(f)$

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

6. *If a node is in the initial stage, then none of its neighbors is involved in root contention.*
$I_6(e) \equiv init[\mathsf{source}(e)] \to \neg contention[\mathsf{target}(e)]$

7. *Child links are empty.*
$I_7(e) \equiv child[e] \to mq[e] = \mathsf{empty}$

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

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

11. *Two nodes can never be children of each other.*
$I_{11}(e) \equiv child[e] \to \neg child[e^{-1}]$

14. *All incoming liks of the source of a child link, except for its inverse, are child links as well.*
$I_{14}(e, f) \equiv child[e] \wedge \mathsf{source}(e) = \mathsf{target}(f) \wedge e \neq f^{-1} \to child[f]$

15. *There is at most one node for which all incoming links are child links.*
$I_{15} \equiv (\exists v \forall e \in \mathsf{to}(v) : child[e]) \to (\exists! v \forall e \in \mathsf{to}(v) : child[e])$

[7] We have dropped the argument v to I_{15}.