Presented at STRATEGIES 2004, Cork, Ireland, July 4, 2004

Reusable PVS Proof Strategies for Proving Abstraction Properties of I/O Automata *

Sayan Mitra
MIT Comp. Sci. and AI Laboratory
32 Vassar Street
Cambridge, MA 02139
mitras@csail.mit.edu

Myla Archer
Naval Research Laboratory
Code 5546
Washington, DC 20375
archer@itd.nrl.navy.mil

Abstract

Recent modifications to PVS support a new technique for defining abstraction properties relating automata in a clean and uniform way. This definition technique employs specification templates that can support development of generic high level PVS strategies that set up the standard subgoals of these abstraction proofs and then execute the standard initial proof steps for these subgoals. In this paper, we describe an abstraction specification technique and associated abstraction proof strategies we are developing for I/O automata. The new strategies can be used together with existing strategies in the TAME (Timed Automata Modeling Environment) interface to PVS; thus, our new templates and strategies provide an extension to TAME for proofs of abstraction. We illustrate how the extended set of TAME templates and strategies can be used to prove example I/O automata abstraction properties taken from the literature.

1 Introduction

One approach to supporting strategies in tactic-based provers such as PVS is to adhere to specification templates that provide a uniform organization for specifications and properties upon which strategies can rely. This approach has been used in the TAME (Timed Automata Modeling Environment) interface to PVS [1, 2].

Until now, TAME proof support has been aimed at properties of a single automaton—mainly state and transition invariants for (both timed and untimed) I/O automata, though TAME does include minimal strategy support for proofs of properties of execution sequences of I/O automata. All of TAME's proof support is aimed at supplying "natural" proof steps that users can employ in checking high level hand proofs of properties of automata that are specified following the TAME automaton template.

One long standing goal for TAME has been to extend its proof support to include proofs of abstraction properties, such as refinement and simulation relations, involving two automata. This goal includes the ability to reuse established specifications and invariants of two automata in defining and proving an abstraction relation between them. A second goal is that the new proof support for abstraction properties should be generic in the same way as TAME support for invariant proofs: that is, there should be

*Funding for this research has been provided by ONR.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2004		**00-00-2004 to 00-00-2004**

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

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

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**14**	

a fixed set of TAME proof steps, supported by PVS strategies, that can be applied to proofs of abstraction properties without being tailored to a specific pair of automata.

The theory interpretation feature [10] in the latest version of PVS (PVS Version 3), combined with some recent enhancements to PVS, makes it possible to accomplish these goals. In previous work [8], we outlined our plan for taking advantage of these new PVS features in specifying abstraction properties and developing uniform PVS strategies for proofs of these properties. In this paper, we describe how specification and proofs of abstraction relations between two I/O automata can now in fact be accomplished in TAME, and illustrate these new capabilities on examples.

This paper is organized as follows. Section 2 reviews TAME's support for invariant proofs and utility of abstraction in verification of I/O automata. Section 3 discusses the past problem with designing TAME support for abstraction proofs, and shows how with PVS 3.2, methods similar to those used in TAME support for invariant proofs can now be used to provide TAME support for abstraction proofs. Section 4 discusses some verification examples from the literature along with the formalization of the relevant abstraction properties in TAME. Section 5 presents in detail a new TAME strategy for proving weak refinement, and describes its usage with examples. This section also discusses progress towards developing an analogous strategy for forward simulation. Finally, Section 6 discusses some related work, and Section 7 presents our conclusions and plans for future work.

2 Background

2.1 I/O Automata model

The formal model underlying TAME is the MMT timed automaton [7], which subsumes the class of untimed I/O automata. In this paper, we refer to MMT timed automata simply as (timed) I/O automata. The main components of an I/O automaton are its set of states, determined by the values of a set of state variables; its set of (usually parameterized) actions that trigger transitions; and its set of start states. The actions are partitioned into *visible* and *invisible* actions. The visible actions, in turn, partition into *input* and *output* actions. For systems involving timing, a special time passage action records passage of time.

An *execution* of an I/O automaton A is an alternating sequence of states and actions of A in which the first state is an initial state of A and each action in the sequence transforms its predecessor state into its successor state. The *trace*, or externally visible behavior of A, corresponding to a given execution α is the sequence of visible actions in α. For timed I/O automata, there are analogous notions of timed executions and timed traces. Further details can be found in [5, 6].

2.2 TAME support for invariant proofs

State (or transition) invariants of an I/O automaton are properties that hold for all of its reachable states (or reachable transitions). To support proofs of invariants of an I/O automaton, TAME provides a template for specifying a (timed or untimed) I/O automaton, a set of standard PVS theories, and a set of strategies that embody the natural high-level steps typically needed in hand proofs of invariants. The standard

PVS theories include generic theories such as `machine` that establishes the principle of induction over reachable states, and special-purpose theories that can be generated from the DATATYPE declarations in an instantiation of the TAME automaton template. A sample of typical TAME steps for invariant proofs is shown in Figure 1.

Proof Step	TAME Strategy	Use
Get base and induction cases and do standard initial steps	**AUTO_INDUCT**	Start an induction proof
Appeal to precondition of an action	**APPLY_SPECIFIC_PRECOND**	Demonstrate need to use precondition
Apply the inductive hypothesis to non-default argument(s)	**APPLY_IND_HYP**	Supplement **AUTO_INDUCT**'s use of default arguments
Apply an auxiliary invariant lemma	**APPLY_INV_LEMMA**	Needed in proving "non-inductive" invariants
Break down into cases based on a predicate	**SUPPOSE**	Add proof comments and labels to PVS' **CASE**
Apply "obvious" reasoning, e.g., propositional, equational, datatype	**TRY_SIMP**	Finish proof branch once facts have been introduced
Use a fact from the mathematical theory for a state variable type	**APPLY_LEMMA**	Perform special mathematical reasoning
Instantiate embedded quantifier	**INST_IN**	Instantiate but don't split first
Skolemize embedded quantifier	**SKOLEM_IN**	Skolemize but don't split first

Figure 1: A sample of TAME steps for I/O automata invariant proofs

2.3 Abstraction properties and trace inclusion for I/O automata

Quite often it is not natural to formulate and prove a desired property P of an automaton A as an invariant property, but is instead natural to think of P as being represented by the behavior of another, more abstract automaton B. In this case, one can show that A satisfies P by showing that every trace of A is a possible trace of B. Since abstraction relations imply trace inclusion, by the careful choice of a specification automaton B for P, the verification that P holds for A can be reduced to proving an abstraction relation between A and B.

Possible abstraction relations between two automata include homomorphism, refinement, weak refinement, forward simulation, backward simulation, and so on. Forward-and-backward simulation relations are both sound and complete with respect to trace inclusion of I/O automata [6], and therefore they constitute a powerful set of tools for automata-based verification.

3 Formalizing abstraction properties for I/O automata

In this section, we describe the hurdles we encountered earlier in developing TAME support for abstraction proofs and how we take advantage of theory interpretations and other new PVS features in adding abstraction proof support to TAME.

3.1 Previous barriers to TAME support for abstraction proofs

Abstraction properties involve a pair of automata, and hence to express them generally, one needs a way to represent abstract automaton objects in PVS.

The most convenient way to represent abstract automaton objects would be to make them instances of a type `automaton`. But, there are barriers to doing this in PVS. An I/O automaton in TAME is determined by instantiations of two types (`actions` and `states`), a set of start states, and a transition relation. Abstractly, these elements can be thought of as fields in a record, and an abstract automaton object can be thought of as an instance of the corresponding record type. However, record fields in PVS are not permitted to have type "type". An alternative way to express a type of automata is to use parametric polymorphism, as in [9]. However, unlike Isabelle/HOL, which was used in [9], PVS does not support parametric polymorphism.

Because no general automaton type can be defined in PVS, I/O automaton objects have been defined in TAME as theories obtained by instantiating the TAME automaton template. Invariants for I/O automata are based on the definitions in these theories.

One can define an abstraction property between two automata defined by instantiating the TAME template theory by creating a new template that imports the template instantiations (together with their associated invariants), and then tailoring the details of a definition of the abstraction property to match the details of the template instantiations. However, this approach is very awkward for the user, who must tailor fine points of complex definitions to specific cases and be particularly careful about PVS naming conventions. It is also awkward for the strategy-writer, whose strategies would need to make multiple probes in a standard definition structure to find specific names. Further, this scheme relies on following a property template to permit a strategy to be reused in different instantiations of the property.

3.2 A new design for defining and proving abstraction in TAME

With the theory instantiation feature of PVS, together with other new PVS features, we have been able to design support for defining abstraction relations between two I/O automata that is both straightforward for a TAME user and clean from the point of view of the strategy developer. This support relies on (1) new TAME supporting theories `automaton` and `timed_automaton`, (2) a library of *property theories*, and (3) new TAME templates for stating abstraction properties as theorems.

Figure 2 shows the theory `automaton`, which can be instantiated by an automaton declaration by concretely defining the components `actions`, `visible`, `states`, etc. A

```
automaton: THEORY

 BEGIN
  actions : TYPE+;
  visible (a:actions) : bool;
  states : TYPE+;
  start (s:states) : bool;
  enabled (a:actions, s:states) : bool;
  trans (a:actions, s:states) : states;
  reachable (s:states) : bool;
  converts (s1, s2: states) : bool;
 END automaton
```

Figure 2: The new TAME supporting theory `automaton`

```
weak_refinement[ A, C : THEORY automaton,
                 actmap: [C.actions -> A.actions],
                 r: [C.states -> A.states] ] : THEORY
  BEGIN
    weak_refinement_base: bool =
      FORALL(s_C:C.states): (C.start(s_C) => A.start(r(s_C)));

    weak_refinement_step : bool =
      FORALL(s_C:C.states, a_C:C.actions):
             C.reachable(s_C) AND C.enabled(a_C,s_C) =>
               (C.visible(a_C) =>
                 (A.enabled(actmap(a_C),r(s_C)) AND
                  r(C.trans(a_C,s_C))= A.trans(actmap(a_C),r(s_C)))) AND
               (NOT C.visible(a_C) =>
                 ((r(s_C) = r(C.trans(a_C,s_C)))
                  OR (r(C.trans(a_C,s_C))= A.trans(actmap(a_C),r(s_C)))))

    weak_refinement: bool = weak_refinement_base & weak_refinement_step;

  END weak_refinement
```

Figure 3: The new TAME property theory `weak_refinement`

```
forward_simulation[C, A : THEORY timed_automaton,
                   amap: [C.actions-> A.actions],
                   r: [C.states, A.states -> bool]] : THEORY
BEGIN
  f_simulation_base:bool = FORALL (s_C: C.states):
    (C.start(s_C) => EXISTS(s_A: A.states): A.start(s_A) AND r(s_C,s_A));

  f_simulation_step:bool =
    FORALL (s_C,s1_C: C.statesl, s_A: A.states, a_C: A.states):
     A.reachable(s_C) AND reachable(s_A) AND r(s_C,s_A) AND
     A.enabled(a_C,s_C) AND s1_C = A.trans(a_C,s_C) =>
          (A.visible(a_C) AND (NOT A.nu?(a_C)) =>
           EXISTS (s1_A,s2_A,s3_A: A.states):
            converts(s_A,s1_A) AND converts(s2_A,s3_A) AND r(s1_C,s3_A) AND
            A.enabled(amap(a_C),s1_A) AND A.trans(amap(a_C),s1_A) = s2_A)
      AND (A.nu?(a_C) => EXISTS (s3_A: A.states):
                           A.t_converts(s_A,s3_A,A.timeof(a_C)) AND r(s1_C,s3_A))
      AND (NOT A.visible(a_C) => EXISTS (s3_A: A.states):
                                   converts(s_A,s3_A) AND r(s1_C,s3_A));

   forward_simulation: bool = f_simulation_base & f_simulation_step;

  END forward_simulation
```

Figure 4: The new TAME property theory `forward_simulation`

new PVS feature allows the use of syntax matching to automatically extract the concrete definitions, thus simplifying the instantiation of `automaton` from a TAME automaton specification. Because `states` and `actions` are both declared as `TYPE+`, i.e., nonempty types, instantiating `automaton` results in two TCCs (type correctness conditions) requiring these types to be nonempty. The theory `timed_automaton` is similar, but includes components related to current time, time passage, and so on.

Examples of property theories for weak refinement and forward simulation are shown in Figures 3 and 4. We are building a library of property theories which include other commonly used abstraction relations such as refinement, backward simulation, etc.

```
tip_abstraction: THEORY
  BEGIN
    IMPORTING tip_invariants
    IMPORTING tip_spec_invariants
    MC  : THEORY = automaton :-> tip_decls
    MA  : THEORY = automaton :-> tip_spec_decls
    amap(a_C: MC.actions): MA.actions  =
            CASES a_C OF
               nu(t): nu(t),
               add_child(e): noop,
               children_known(c): noop,
               ack(a): noop,
               resolve_contention(r): noop,
               root(v): root(v),
               noop:noop
            ENDCASES
    ref(s_C: MC.states): MA.states =
       (# basic := (# done := EXISTS (v:Vertices): root(v,s_C) #),
          now := now(s_C),
          first := (LAMBDA(a:MA.actions): zero),
          last := (LAMBDA(a:MA.actions): infinity) #)
    IMPORTING weak_refinement[MA, MC, amap, ref]
    tip_refinement_thm: THEOREM refinement
  END tip_abstraction
```

Figure 5: Instantiating the `weak_refinement` template for *TIP*

```
timeout_abstraction:THEORY
  BEGIN
    IMPORTING timeout_invariants
    IMPORTING periodic_send_channel_timeout_invariants
    MC  : THEORY = timed_automaton :-> periodic_send_channel_timeout_decls
    MA  : THEORY = timed_automaton :-> timeout_decls
    amap(a_C: MC.actions): MA.actions  =
      CASES a_C OF
        nu(t): nu(t),
        send(m): noop,
        receive(m): noop,
        fail: fail,
        timeout: timeout,
        noop:noop
      ENDCASES
    ref(s_C: MA.states, s_A: MA.states):bool =
      failed(s_C) = failed(s_A) AND
      suspected(s_C) = suspected(s_A) AND
      FORALL (a_C:MC.actions): visible(a_C) =>
             (first(s_C)(a_C) = first(s_A)(amap(a_C))
              AND last(s_C)(a_C) = last(s_A)(amap(a_C))) AND
      now(s_C) = now(s_A))
    IMPORTING forward_simulation[MC,MA, amap, ref]
    timeout_fw_simulation_thm: THEOREM forward_simulation
  END timeout_abstraction
```

Figure 6: Instantiating the `forward_simulation` template for *PCT*

Particular instantiations of the TAME template for stating abstraction properties as theorems are given in Figures 5 and 6, which show the instantiations corresponding to examples below in Sections 4.1 and 4.3, respectively. Each template instantiation instantiates two copies—one for each of the abstract and the concrete automata—of either the `automaton` or the `timed_automaton` theory (as appropriate), defines the action and state mappings between the two automata, and imports the relevant property theory with all the above as parameters.

4 Examples

In this section, we illustrate with examples how to use the theories and template introduced in the previous section to state an abstraction property between a pair of I/O automata as a theorem (to be proved).

4.1 Leader Election Protocol

Our first exercise in using the extended templates and strategies of TAME was to formalize a weak-refinement proof for a simple spanning-tree based leader election protocol. Figure 5 shows our template instantiated with the automata *TIP* (representing the leader election protocol) and *SPEC* from [3]. The TAME specifications of these two automata are the theories `tip_decls` and `tip_spec_decls`. A set of invariants proved for *TIP* (both by the authors of [3] and in TAME [2]) establishes that at any given point in the execution of the algorithm, at most one leader has been chosen. The automaton *SPEC* has only one visible action (excluding the time passage action `nu`): namely, `root`. A weak refinement from *TIP* to *SPEC* is used to establish that all traces of *TIP* are included in the set of traces of *SPEC*, thus ensuring that the choice of a leader—the `root` action—occurs at most once in any execution of *TIP*.

The `tip_abstraction` theory in Figure 5 imports the library theory `weak_refinement` (Figure 3) with four parameters. The parameters `MA` and `MC` are instantiations of the `automaton` theory corresponding to the *TIP* and the *SPEC* automata; `amap` is a map from the actions of *TIP* to the actions of *SPEC, and* `ref` *is the refinement function from the states of* TIP *to the states of* SPEC. As a result of this importing the `weak_refinement` relation between *TIP* and *SPEC* is defined, and hence the corresponding refinement theorem `tip_refinement_thm` can be stated.

4.2 Failure Prone Memory Component

Our second case study concerns the specification and implementation of the memory component of a remote procedure call (RPC) module taken from [11]. A failure prone memory component *MEM* and a reliable memory component *REL_MEM* are modeled as I/O automata, and the requirement is to show that every trace of *REL_MEM* is a trace of *MEM*. The *MEM* and *REL_MEM* automata are almost identical, except that the `failure` action in *MEM* is absent in *REL_MEM*. Owing to this similarity, the refinement function `ref` is a bijection and the action map `amap` is an injection. As noted in [11], once again, a weak refinement from *REL_MEM* to *MEM*, suffices to establish trace inclusion, and we state this weak refinement property in the same way as in the previous example.

4.3 Periodic Send-Timeout Process

The final example concerns the composition of a periodically sending process P, a timed channel C, and a timeout process T, taken from [4]. The process P periodically sends messages, every u_1 time until an externally controlled `failure` action occurs. C enqueues each message with a deadline for its delivery, which is at most b time from its sending time. An enqueued message is received by T sometime before its delivery deadline. If no message is received by T over an interval longer than u_2, then it performs a `timeout` action and *suspects* P (to be failed). If $u_2 > u_1 + b$, then T suspects P only if P has really failed. The external behavior of the composed automaton PCT is captured by a simple abstract automaton ABS in which a `failure` action is always followed by a `timeout` action, within $u_2 + b$ time.

Taking `failure`, `timeout`, and time passage actions to be visible, we have proved a forward simulation relation from PCT to ABS in PVS, thus establishing that every trace of PCT is a trace of ABS. The forward simulation property is stated using a template similar to that in Figure 5. The only differences are that `ref` is now a *relation* instead of a function and the property theory imported is `forward_simulation` from Figure 4.

5 Strategies for abstraction proofs

In this section, we discuss two TAME strategies for abstraction proofs: **PROVE_REFINEMENT**, for which we have a fairly polished prototype, and **PROVE_FWD_SIM** which we are beginning to develop. Basing the strategies on generic property theories allows the strategies themselves to be generic i.e., they can be applied towards proving their associated abstraction properties for arbitrary automaton pairs. As an illustration, we show the results we obtained from applying **PROVE_REFINEMENT** in the *TIP* and *RPC* case studies.

5.1 The PROVE_REFINEMENT strategy

PROVE_REFINEMENT, our generic strategy for weak refinement proofs, is based on the generic `weak_refinement` property defined in the theory `weak_refinement` shown in Figure 3 on page 5. **PROVE_REFINEMENT** is designed to be invoked on a theorem which, like `tip_refinement_thm` in Figure 5 on page 6, asserts `weak_refinement`. An overview of **PROVE_REFINEMENT** showing its heirarchy of substrategies is shown in Figure 7. **PROVE_REFINEMENT** undertakes to prove the weak-refinement theorem inductively by exploiting its known structure. First, **PROVE_REFINEMENT** splits a weak refinement theorem into its base case and induction case (corresponding to `weak_refinement_base` and `weak_refinement_step` in Figure 3). Next, **PROVE_REFINEMENT** delegates the base case to a substrategy called **SETUP_REF_BASE**, probes to see if it can discharge the base case trivially, and then applies **SETUP_REF_INDUCT_CASES** to split up the induction case into branches for the individual action types of the `actions` datatype of the concrete automaton. The substrategy **SETUP_REF_BASE** performs the standard steps needed in the base case: skolemization, applying PVS's EXPAND to the definitions of `start` and `ref`, and some minor simplifications.

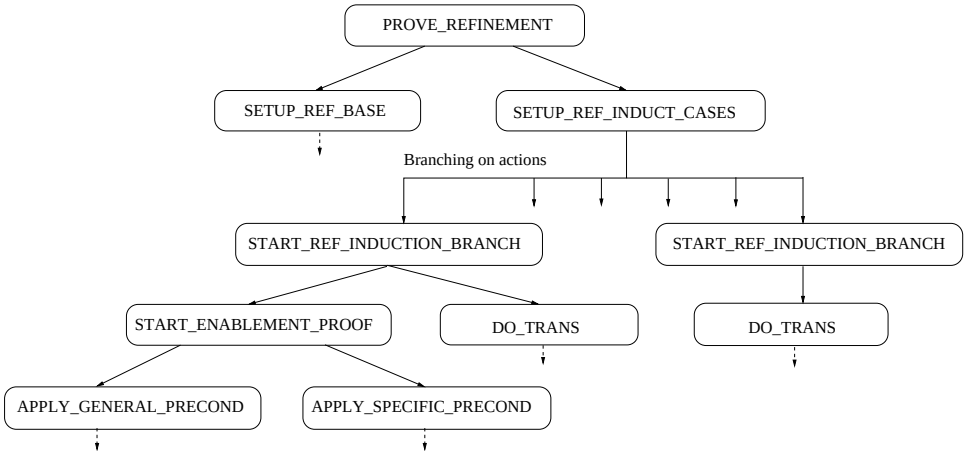

Figure 7: The **PROVE_REFINEMENT** strategy and related substrategies.

After the substrategy **SETUP_REF_INDUCT_CASES** splits the induction case on the actions, each subgoal is handed off to **START_REF_INDUCTION_BRANCH**, which performs skolemization and expands the definition of `visible`. This yields different sets of subgoals for visible and invisible actions. For each invisible action, a single congruence subgoal is generated from the condition in lines 8-9 in the `weak_refinement_step` definition in Figure 3. For each visible action, two new subgoals, an enablement subgoal and a congruence subgoal, are generated from lines 5 and 6 in `weak_refinement_step`.

Congruence subgoals concern the correspondence of poststates, and **START_REF_INDUCTION_BRANCH** applies the substrategy **DO_TRANS** to them; this strategy just expands the transition definition and repeatedly simplifies. **START_REF_INDUCTION_BRANCH** handles the first (enablement) subgoal for visible actions by applying the **START_ENABLEMENT_PROOF** strategy. **START_ENABLEMENT_PROOF** splits the enablement goal into subgoals for the general (timeliness) precondition and the specific precondition of the action, which it respectively handles by **APPLY_GENERAL_PRECOND** followed by a probe to see if this subgoal can be trivially discharged, and **APPLY_SPECIFIC_PRECOND.**

PROVE_REFINEMENT resolves most of the subgoals for simple base and action cases of refinement proofs. For the subgoals that are not resolved, the user must interact with PVS, using steps such as TAME's **APPLY_INV_LEMMA**, **INST_IN**, **SKOLEM_IN**, **TRY_SIMP**, and so on.

In the interaction of **PROVE_REFINEMENT** and its substrategies, significant use is made of formula labels, both for deciding which action to take based on the presence or absence of a formula with a given label, and to focus computation on formulae with specific labels. The labels are designed to be informative: for example, the label `A.specific-precondition` on line 4 of the proof in Figure 12 belongs to the specific precondition of the `root` action of the abstract automaton (in this case, *SPEC*). This is so that when an unresolved subgoal is returned to the user, its content is as informative as possible. For the same reason, **PROVE_REFINEMENT** and its substrategies attach comments to any subgoals they create that denote their significance. The comment `;; root(rootV_C_action) specific enablement` that appears on line 2 in Figure 12

indicates that this subgoal is the specific enablement subgoal for the action `root`. The argument to `root`, `rootV_C_action`, is a skolem constant (automatically generated by **PROVE_REFINEMENT**) for the formal parameter `rootV` of `root`; its suffix `_C_action` indicates that it is generated from an action of the concrete automaton (in this case, *TIP*). Thus our new strategy **PROVE_REFINEMENT** adheres to the same design principles as the earlier TAME strategies (see [1]).

To illustrate some of the above features of the supporting strategies for **PROVE_REFINEMENT** as well as the nature of strategy-writing in PVS, we show the actual strategy definition for the substrategy **START_ENABLEMENT_PROOF** in Figure 8. Some details that can be seen from this code are that **START_ENABLEMENT_PROOF** is passed a `comment-string` as an argument, that it contains Lisp code to compute two commands (`gencomcmd` and `speccomcmd`) that apply comments computed from `comment-string` (which is normally the name of an action), and that it both uses labels (`enabled C.action` and `enabled A.action`) and applies labels (`C.general-precondition`, etc.). Further, it splits the current subgoal into general enablement and specific enablement subgoals. The call to **TIME_SIMP_PROBE** is the probe that tries to prove the general enablement subgoal automatically and backtracks on failure.

```
(defstep start_enablement_proof (comment-string)
  (let ((gencomcmd
         `(comment
           ,(format nil "~a~a" comment-string " general enablement")))
        (speccomcmd
         `(comment
           ,(format nil "~a~a" comment-string " specific enablement"))))
  (then (expand "enabled" "enabled C.action" :assert? NONE)
        (with-labels (flatten-disjunct)
                     (("C.general-precondition"
                       "C.specific-precondition")))
        (expand "enabled" "enabled A.action" :assert? NONE)
        (branch
          (with-labels (split)
                       (("A.general-precondition")
                        ("A.specific-precondition")))
          ((then (apply_general_precond)
                 gencomcmd
                 (time_simp_probe)
                 (postpone))
           (then (apply_specific_precond)
                 speccomcmd
                 (postpone))))))
  "" "Starting proof of the enablement branch")
```

Figure 8: Code for the strategy **START_ENABLEMENT_PROOF.**

5.2 Applying PROVE_REFINEMENT

We can illustrate the behavior of **PROVE_REFINEMENT** by showing some details of how it works when applied to the *TIP/SPEC* example. First, **PROVE_REFINEMENT** breaks up the `tip_refinement_thm` into the base case and the induction case. The base case sequent as seen by **SETUP_REF_BASE** is shown in Figure 9, in which `tip_decls.start` and `tip_spec_decls.start` are the start predicates of *TIP* and *SPEC*,

```
   ;;; Base case
     |-------
   {1}   FORALL (s_C: tip_decls.states):
           (tip_decls.start(s_C) => tip_spec_decls.start(ref(s_C)))
```

Figure 9: Initial base case sequent for tip_abstraction.

```
[-1,(reachable C.prestate)]
      reachable(sC_theorem)
[-2,(enabled C.action)]
      enabled(nu(timeofC_action), sC_theorem)
  |-------
{1,(enabled A.action)}
      tip_spec_decls.enabled
          (nu(timeofC_action),
           (# basic :=
               (# done := EXISTS (v: Vertices): root(v, sC_theorem) #),
              now := now(sC_theorem),
              first := (LAMBDA (a: MA.actions): zero),
              last := (LAMBDA (a: MA.actions): infinity) #))
```

Figure 10: Initial enablement sequent for the action nu in *TIP*.

```
[-1,(reachable C.prestate)]
      reachable(sC_theorem)
[-2,(enabled C.action)]
      enabled(nu(timeofC_action), sC_theorem)
  |-------
{1,(congruence)}
      ((# basic :=
           (# done:= EXISTS (v: Vertices):
                          root(v, trans(nu(timeofC_acton), sC_theorem)) #),
          now := now(trans(nu(timeofC_action), sC_theorem)),
          first := (LAMBDA (a: MA.actions): zero),
          last := (LAMBDA (a: MA.actions): infinity) #) =
      tip_spec_decls.trans(nu(timeofC_action),
                           (# basic :=
                              (# done:= EXISTS (v: Vertices):
                                              root(v, sC_theorem) #),
                            now := now(sC_theorem),
                            first := (LAMBDA (a: MA.actions): zero),
                            last := (LAMBDA (a: MA.actions): infinity) #)))
```

Figure 11: Initial congruence sequent for the action nu in *TIP*.

respectively. Next, the induction case (not shown) is split up by **SETUP_REF_INDUCT_CASES** into six branches, one for each of the six actions in the tip_decls.actions type. As noted in Section 5.1, the handling of each induction goal depends on whether the corresponding action is visible. Because nu is a visible action in *TIP*, two subgoals—for enablement and congruence—are generated. These are shown in Figures 10 and 11. In general, the enablement goal is then split into a general enablement goal and a specific enablement goal. For the nu action, the enablement goal is the same as the specific enablement goal. For *TIP*/*SPEC*, all the general enablement subgoals

```
(""
 (prove_refinement)
 (("1" ;; Case root(rootV_C_action) specific enablement
   (skolem_in "A.specific-precondition" "v_1")
   (apply_inv_lemma "15" "s_C_theorem")
   ;; Applying the lemma
   ;; (EXISTS (v: Vertices): FORALL (e: tov(v)): child(e, s_C_theorem)) =>
   ;;  ((EXISTS (v: Vertices): FORALL (e: tov(v)): child(e, s_C_theorem)) &
   ;;    (FORALL (v, w: Vertices):
   ;;       (((FORALL (e: tov(v)): child(e, s_C_theorem)) &
   ;;          (FORALL (e: tov(w)): child(e, s_C_theorem)))
   ;;         => v = w)))
   (inst_in "lemma_15" "rootV_C_action")
   (inst_in "lemma_15" "v_1" "rootV_C_action")
   (skolem_in "lemma_15" "e_1")
   (apply_inv_lemma "13" "s_C_theorem" "e_1")
   ;; Applying the lemma
   ;; FORALL (e: Edges): root(target(e), s_C_theorem) => child(e, s_C_theorem)
   (try_simp))
  ("2" ;; Case root(rootV_C_action) congruence
   (inst "congruence" "rootV_C_action")
   (try_simp))))
```

Figure 12: TAME refinement proof for *TIP/SPEC*.

are discharged as trivial.

The saved proof for the *TIP/SPEC* case study (Figure 12) shows that all but two parts of the inductive goal for the `root` action—the specific enablement subgoal and the congruence subgoal—are resolved by **PROVE_REFINEMENT** automatically. Proving the `root` specific enablement goal, which is shown in Figure 13, requires using two invariant properties (invariants 13 and 15 from [3]) *TIP*, proved earlier with TAME. The `root` congruence goal requires **INST_IN**. Both subgoals require the TAME "it is now trivial" step **TRY_SIMP** (see Figure 1) to complete.

For the failure-prone memory case study, because of the simple nature of the refinement, the entire proof is resolved automatically by **PROVE_REFINEMENT**, and we therefore do not show the proof.

5.3 The PROVE_FWD_SIM strategy

We have proved in PVS the forward simulation relation for the periodic send-timeout process described in Section 4.3, and we are currently in the process of developing a generic TAME strategy **PROVE_FWD_SIM** for forward simulation proofs. The initial steps of **PROVE_FWD_SIM** are similar to those in **PROVE_REFINEMENT**: splitting the simulation theorem into base and induction cases, and then splitting the induction case into subcases for the individual actions. The greater complexity of the definition of forward_simulation means that our ultimate **PROVE_FWD_SIM** will be more complex than **PROVE_REFINEMENT**, and that we may need to design additional TAME proof steps appropriate for completing branches of forward simulation proofs interactively.

```
tip_refinement_thm.1 :
;;; Case root(rootV_C_action) specific enablement

{-1,(A.specific-precondition)}
      EXISTS (v: Vertices): root(basic(s_C_theorem))(v)
{-2,(reachable C.prestate)}
      reachable(s_C_theorem)
{-3,(C.general-precondition)}
      enabled_general(root(rootV_C_action), s_C_theorem)
{-4,(C.specific-precondition_part_4 C.specific-precondition)}
      FORALL (e: tov(rootV_C_action)): child(basic(s_C_theorem))(e)
  |-------
{1,(C.specific-precondition_part_1 C.specific-precondition)}
      init(basic(s_C_theorem))(rootV_C_action)
{2,(C.specific-precondition_part_2 C.specific-precondition)}
      contention(basic(s_C_theorem))(rootV_C_action)
{3,(C.specific-precondition_part_3 C.specific-precondition)}
      root(basic(s_C_theorem))(rootV_C_action)
```

Figure 13: First subgoal returned by **PROVE_REFINEMENT** in Figure 12

6 Related work

A metatheory for I/O automata, based on which generic definitions of invariant and abstraction properties are possible, has been developed in Isabelle by Müller [9], who also developed an associated verification framework. Example proofs of (e.g.) forward simulation have been done for at least simple examples using this framework; it is not clear to what extent uniform Isabelle tactics are employed. PVS has been used by others to do abstraction proofs, and in fact a refinement proof for *TIP* and *SPEC* was mechanized by Devillers et al. [3]. However, to our knowledge, no one has developed "generic" PVS strategies to support proving abstraction properties with PVS.

7 Conclusions and future work

Currently, we have developed supporting PVS theories and templates for abstraction proof strategies, and added them to TAME. The supporting theories include a set of abstraction property theories that are being collected into a library, and generic automaton theories that serve as theory types for theory parameters to our property theories. For each abstraction property theory, there is a template to allow the abstraction property to be instantiated as a (proposed) theorem that relates two particular automata. Building on this structure, we have (1) added a reusable PVS weak refinement strategy to TAME, and applied it to examples, and (2) developed an example forward simulation proof, which we are using as a guide to the development of a reusable forward simulation strategy. In the example weak refinement proofs we have done so far, previously existing TAME strategies provide sufficient proof steps for interactively completing the refinement proofs.

Our approach to developing strategies for abstraction proofs is geared towards theorem provers that support tactic-style interactive proving. In theorem proving systems that allow a definition of an automaton type, an approach to developing such strategies that is not based on templates may be possible. Because we cannot expect to develop strategies that will do arbitrary abstraction proofs fully automatically, a major goal

for us is to design our strategies to support user-friendly interactive proving. A PVS feature that facilitates making both interaction with the prover and understanding the significance of saved proofs easier is support for comments and formula labels. Thus, a challenge for other theorem proving systems is to find ways to support ease of understanding during and after the proof process equivalent to what we provide using PVS.

For the future, we plan to complete the development of the forward simulation strategy and add similar proof support for other abstraction properties to TAME. We also expect to add further TAME steps, if needed, for (interactively) completing proofs of action cases. Eventually, we expect to add our enhanced version of TAME to the set of tools being developed to support specification and verification of timed I/O automata (TIOA) [4].

Acknowledgements

We thank Sam Owre and Natarajan Shankar of SRI International for adding the features to PVS that have made our work possible.

References

[1] Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Math. and Artif. Intel.*, 29(1-4):139–181, 2000.

[2] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

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

[4] D. K. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Draft. MIT Laboratory for Computer Science, Apr., 2004.

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

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

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

[8] S. Mitra and M. Archer. Developing strategies for specialized theorem proving about untimed, timed, and hybrid I/O automata. In *Proc. 1st Int'l Wkshop on Design and Appl. of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, Rome, Italy, Sept. 8 2003.

[9] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universität München, Sept. 1998.

[10] S. Owre and N. Shankar. Theory Interpretations in PVS. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, April 2001. Draft.

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