PVS Strategies for Proving Abstraction Properties of Automata

Sayan Mitra [1]

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, USA

Myla Archer [2]

Code 5546, Naval Research Laboratory,
Washington, DC 20375, USA

Abstract

Abstractions are important in specifying and proving properties of complex systems. To prove that a given automaton implements an abstract specification automaton, one must first find the correct abstraction relation between the states of the automata, and then show that this relation is preserved by all corresponding action sequences of the two automata. This paper describes tool support based on the PVS theorem prover that can help users accomplish the second task, in other words, in proving a candidate abstraction relation correct. This tool support relies on a clean and uniform technique for defining abstraction properties relating automata that uses library theories for defining abstraction relations and templates for specifying automata and abstraction theorems. The paper then describes how the templates and theories allow development of generic, high level PVS strategies that aid in the mechanization of abstraction proofs. These strategies first set up the standard subgoals for the abstraction proofs and then execute the standard initial proof steps for these subgoals, thus making the process of proving abstraction properties in PVS more automated. With suitable supplementary strategies to implement the "natural" proof steps needed to complete the proofs of any of the standard subgoals remaining to be proved, the abstraction proof strategies can form part of a set of mechanized proof steps that can be used interactively to translate high level proof sketches into PVS proofs. Using timed I/O automata examples taken from the literature, this paper illustrates use of the templates, theories, and strategies described to specify and prove two types of abstraction property: refinement and forward simulation.

Key words: Mechanical Theorem Proving, Strategies, I/O Automata, Abstraction, Refinement, Forward Simulation.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Abstractions are essential for applying formal methods to verify certain classes of properties of complex systems, such as properties of system executions. Given an automaton C, suppose we wish to verify that every visible behavior—i.e., trace—of C satisfies some property P. An effective way of doing this is to model the property P itself as the set of traces of an abstract automaton A, and then to show that the set of traces of C is a subset of the set of traces of A. This method generalizes to using not just a single abstraction, as above, but many levels of abstraction between the concrete, possibly complex implementation of a system and its abstract specification. A systematic way of showing trace inclusion—that every trace of automaton C is included in the set of traces of another automaton A—is to show that there exists an abstraction relation between the states of the two automata. Thus the creation of a specification automaton A for the property P can reduce the problem of verifying that P holds for C to proving an abstraction relation between C and A.

There are several possible abstraction relations between two automata, homomorphism, refinement, forward simulation, backward simulation, and so on. Forward-and-backward simulation relations are complete with respect to trace properties of I/O automata [9], and therefore they are powerful tools for automata-based verification. In this paper, we present a clean and uniform way of specifying abstraction properties relating pairs of automata in the PVS theorem prover [17] and describe how our specifications allow us to provide a set of generic strategies that aid users in proving abstraction properties while minimizing the necessary interaction with the prover.

One approach to supporting generic 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 TAME (Timed Automata Modeling Environment) [2,3], an interface which is designed to simplify proving properties of automata in PVS. 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 longstanding goal for TAME has been to extend its proof support to include proofs of refinement, simulation, and other abstraction properties 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 part of this goal is that the new

[1] Email: `mitras@theory.csail.mit.edu`
[2] Email: `archer@itd.nrl.navy.mil`

proof support for abstraction properties should be generic in the same way as TAME support for invariant proofs: that is, there should be 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. Finally, this goal includes making the new TAME proof steps "natural"—that is, they should provide a straightforward representation in PVS of the high level proof steps used in hand proofs of abstraction properties. The theory interpretation feature [15] in the latest version of PVS (PVS Version 3), combined with some recent enhancements in PVS 3.2, makes it possible to accomplish these goals.

In previous work [12], 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 automata can now in fact be accomplished in TAME, and illustrate these new capabilities on examples. Section 2 reviews automata models and TAME's support for invariant proofs; 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 3 describes the strategies we have developed for proving refinement and illustrates their usage on examples. Section 4 does the same for forward simulation. Finally, Section 5 discusses some related work, and Section 6 presents our conclusions and future plans.

2 Background

2.1 I/O Automata model

The formal model underlying TAME is the MMT automaton [11]. A general theory of Timed Input/Output Automata (TIOA) [8] for systems involving both discrete and continuous behavior has evolved since the inception of the MMT automaton. The TIOA model subsumes many other automata models, including the (untimed) I/O automaton model suitable for describing systems with only discrete events, the MMT automaton model, and the Alur-Dill timed automaton model [1]. Although the model used for the original development of the TAME was the MMT automaton, it will be possible to make the TAME templates and strategies work for a large and useful class of timed I/O automata with minor changes (see [6] for a report on ongoing work in this direction). Therefore, in this paper, we refer to MMT timed automata simply as (timed) I/O automata. In the following paragraph we give a very brief overview of the I/O automaton framework. For a complete description of the TIOA model and the related results, see [8].

The main elements of an I/O automaton A are its set of states, determined by the values of a set of state variables; its set of (usually parameterized) actions that trigger state transitions; and its set of start states. An *execution* of an 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. For systems involving continuous evolution, a special time passage action records the changes in the continuous variables after an interval of time. The set of all possible behavior of A, then, is the set of all its executions. To define the notion of visible behavior, the actions of A are partitioned into *visible* and *invisible* actions. The *trace*, or the externally visible behavior of A, corresponding to a given execution α is the sequence of visible actions in α. In order to define the parallel composition operator on automata in a meaningful way, the visible actions are further partitioned into *input* and *output* actions. Since in the rest of this paper we reason only about individual automata (simple or explicitly composed) and not about component automata that are composed using the composition operator, for simplicity, we do not partition visible actions into input and output subsets.

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`, which 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

Fig. 1. A sample of TAME steps for I/O automata invariant proofs

2.3 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, such a record type is not possible in PVS, because record fields in PVS are not permitted to have type "type". An alternative way to express a type of automata would be to use create a polymorphic type `automaton`$[\alpha, \sigma]$ analogous to the polymorphic type `(`α`,`σ`)ioa` in [14]. However, unlike Isabelle/HOL, which was used in [14], PVS does not support parametric polymorphism.

Because no general automaton type can be defined in PVS, I/O automata are represented in TAME as theories obtained by instantiating the TAME automaton template. Invariants for I/O automata are based on the definitions in these theories. We will refer to instantiations of the TAME automaton template as *TAME automata*.

One possible way to support the definition of abstraction properties between two TAME automata is to create abstraction property templates that import two TAME automata (together with their associated invariants), and then require the user to tailor certain details of a definition of the abstraction property to match the details of the TAME automata. 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 property-definition structure to find details of the tailored definitions. Further, this scheme relies on the user adhering properly to a property template to permit a strategy to be reused in different instantiations of the property.

What is really needed is some means to define abstraction properties *generically* in PVS. The design described in Section 2.4 provides such a means.

2.4 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) a new TAME supporting theory `automaton`, (2) a library of *property theories*, and (3) new TAME templates for stating abstraction properties as theorems.

```
automaton:THEORY
  BEGIN
    actions: TYPE+;
    states: TYPE+;
    start(s:states):bool;
    visible(a:actions):bool;
    enabled(a:actions, s:states): bool;
    trans(a:actions, s:states):states;

    reachable(s:states):bool
    invisible_seq_trans(s1, s2:states):bool;
    time_seq_trans(s1,s2:states, t:real):bool;
  END automaton
```

Fig. 2. The new TAME supporting theory `automaton`

Figure 2 shows the theory `automaton`, which is an abstract declaration of the elements that specify an automaton. Other than the pattern of type restrictions, theory `automaton` makes no restrictions on the elements of its instantiations. In TAME, `automaton` is instantiated only with TAME automata. The names and types of elements in `automaton` match exactly the names and types of elements in the TAME automaton template. A new PVS feature allows the use of syntax matching to automatically extract the elements from a TAME automaton specification that instantiate elements of `automaton`, allowing the user to simply refer to the theory name of the TAME automaton as the instantiation for `automaton`; this relieves the user from the tedious effort of explicitly listing element instantiations. 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.

Instantiation of `automaton` by a TAME automaton provides concrete definitions of the `automaton`'s elements. The concrete definitions for the first six of these elements are written by the user when filling in the TAME automaton template to define a TAME automaton. The remaining three elements are the same for all TAME automata and are defined in the `time_machine` theory which is a part of the TAME library. The user does not have to redefine them for specific automata, because an appropriate instance of `time_machine` is imported in the TAME automaton template. In every instantiation of `automaton` by a TAME automaton, the elements of `automaton` have the following properties: The `visible` and `start` predicates define the set of externally visible actions and the set of start states, respectively. The `enabled(a,s)` function returns `true` if action `a` is enabled in state `s`; it returns `false` otherwise. The transition function `trans(a,s)` gives the poststate that results from applying action `a` on state `s`. The `reachable` predicate recursively defines the set of reachable states of the automaton. The predicate `invisible_seq_trans(s1,s2)` is `true` if and only if there exists a sequence of invisible actions that takes state `s1` to `s2`. And `time_seq_trans(s1,s2,t)` is `true` if and only if there exists a sequence of invisible and time passage actions that takes `s1` to `s2` with a total time passage of `t` units.

Examples of property theories for weak refinement and forward simulation are shown in Figures 3 and 4. As an aid to understanding the PVS notation, we will define in English the more complex of these two properties: forward simulation. A relation `r` between the states of a concrete automaton `C` and an abstract automaton `A` is a *forward simulation relation* if (1) for every start state of `C`, there exists a start state of `A` such that the two states are related by `r`, and (2) for every reachable transition `(s_C, a, s1_C)` of `C`, if a state `s_A` is related to `s_C`, then there exists an action sequence σ of `A` that takes `s_A` to `s3_A` such that: (a) `r(s1_C, s3_A)`, and (b) the visible part of σ is same as of `a`. In the definition of `f_simulation_action` in Figure 4 part (2) of this definition is broken into the three cases—`a` is a visible non-time-passage action, `a` is a time-passage action, and `a` is an invisible action—to simplify strategy development. We are building a library of property theories which include other commonly used abstraction relations such as homomorphism, refinement, backward simulation, etc.

A particular instance of the TAME template for stating abstraction properties as theorems is the theory `tip_abstraction` shown in Figure 5. (Note that in PVS notation, lists of assignments between the end markers `(#` and `#)` denote a record value. In TAME, both states and the `basic` components of states are record values.) The theory `tip_abstraction` instantiates two copies—one for each of the abstract and the concrete automata—of the `automaton` theory, defines the action and state mappings between the two automata, and imports the relevant property theory with all the above as parameters.

```
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
```

Fig. 3. The new TAME property theory `weak_refinement`

```
forward_simulation[C, A : THEORY timed_automaton,
                   actmap: [C.actions -> A.actions],
                   r: [C.states, A.states -> bool]] : THEORY
 BEGIN
  f_simulation_base: bool = FORALL(s_C):
     (C.start(s_C) =>  EXISTS(s_A): A.start(s_A) AND r(s_C,s_A));
  f_simulation_action: bool =  FORALL(s_C, s1_C, s_A, a_C):
     (C.reachable(s_C) AND A.reachable(s_A) AND r(s_C,s_A) AND
      C.enabled(a_C,s_C) AND s1_C = C.trans(a_C,s_C)) =>
       (C.visible(a_C) AND (NOT C.nu?(a_C)) =>
          EXISTS (s1_A, s2_A, s3_A):
             A.invisible_seq_trans(s_A,s1_A) AND
             A.invisible_seq_trans(s2_A,s3_A) AND
             r(s1_C, s3_A) AND A.enabled(actmap(a_C),s1_A) AND
             A.trans(actmap(a_C),s1_A) = s2_A) AND
       (C.nu?(a_C) =>
          EXISTS (s3_A):
             A.time_seq_trans(s_A,s3_A,timeof(a_C)) AND
             r(s1_C, s3_A)) AND
       (NOT C.visible(a_C) =>
          EXISTS (s3_A):
             A.invisible_seq_trans(s_A, s3_A) AND
             r(s1_C, s3_A));
  forward_simulation: bool = f_simulation_base & f_simulation_action;
 END forward_simulation
```

Fig. 4. The new TAME property theory `forward_simulation`

```
 tip_abstraction: THEORY
  BEGIN
    IMPORTING TIP_invariants
    IMPORTING SPEC_invariants
    MC  : THEORY = automaton :-> TIP_decls
    MA  : THEORY = automaton :-> 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),
            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 weak_refinement
  END tip_abstraction
```

Fig. 5. Instantiating the `weak_refinement` template for *TIP*

3 Strategies for refinement proofs

In this section, we discuss strategies we have developed for proving weak refinements for timed and untimed I/O automata. Since having a weak refinement relation between the states of two automata is equivalent to having a refinement relation between the reachable states of the two automata, we will often simply refer to "refinement" in what follows. We illustrate the utility of these strategies by sketching the proof of the correctness of a tree based leader election protocol and a failure prone memory in a remote procedure call (RPC) module.

3.1 Design of the refinement strategy

Our main strategy for proving refinements, **PROVE_REFINEMENT**, is based on the `weak_refinement` property shown in Figure 3. The generic nature of the definition of the `weak_refinement` property allows us to define **PROVE_REFINEMENT** in such a way that it can be applied to an arbitrary refinement proof between any given pair of automata. This strategy is designed to perform much if not all of the work, for an arbitrary instantiation of the `weak_refinement` template, of proving by induction that the mapping `ref` from the states of concrete automaton `MC` to the states of abstract automaton `MA` is a refinement. The overall structure of **PROVE_REFINEMENT** in terms of substrategies is shown in Figure 6. First, **PROVE_REFINEMENT** splits a refinement theorem into its base case and induction case (corresponding to `weak_refinement_base` and `weak_refinement_step` in Figure 3). The base case is delegated to a substrategy called **SETUP_REF_BASE**, which performs the standard steps needed in the base case, including skolemizing, applying PVS's EXPAND to the definitions of `start` and `ref`, and making some minor simplifications. **PROVE_REFINE_MENT** then probes to see if the base case can be discharged trivially. Next, **PROVE_REFINE_MENT** turns over the induction branch to the substrategy **SETUP_REF_INDUCT_CASES**.

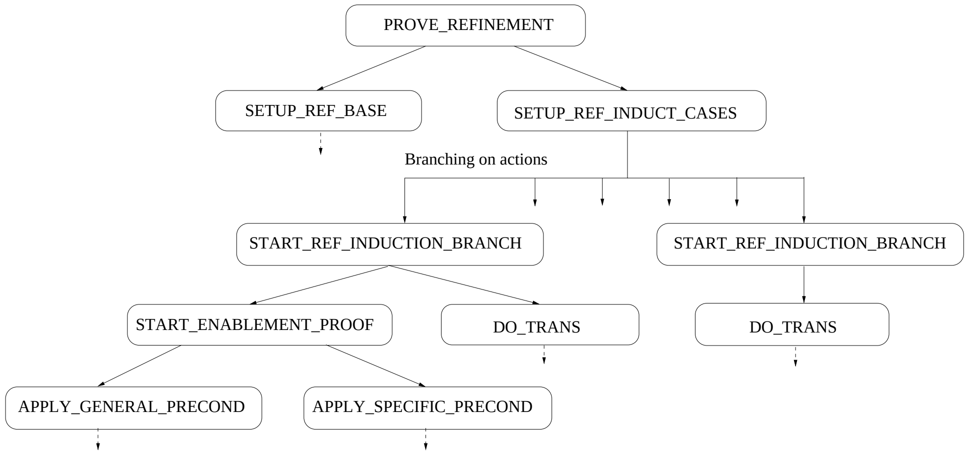

Fig. 6. The **PROVE_REFINEMENT** strategy and related substrategies.

The substrategy **SETUP_REF_INDUCT_CASES** splits up the induction step into individual subgoals for each of the action types in the `actions` datatype, and hands off these individual subgoals to the substrategy **START_REF_INDUCTION_BRANCH**, which performs skolemization and expands the definition of `visible`. As reflected in Figure 6, 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 10-11 in Figure 3. For each visible action, two new subgoals: an enablement subgoal and a congruence subgoal, are generated from lines 7 and 8 in Figure 3, respectively.

Congruence subgoals concern the correspondence of poststates, and **START_REF_INDUCTION_BRANCH** applies 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 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 next section we introduce the Tree Identify Protocol (TIP), which will serve us as a case study for illustrating the operation of the above strategies.

3.2 Refinement for correctness of the TIP algorithm

The TIP algorithm is a part of IEEE 1394 Firewire standard [4] and has been used as a case study for many different formal verification approaches. The TIP leader election protocol is invoked after a bus reset in the network (i.e. when a node is added to, or removed from, the network). Immediately after a bus reset all nodes in the network have equal status, and know only to which nodes they are directly connected. A leader needs to be chosen to act as the manager of the bus for subsequent operations. The TIP algorithm “grows” a directed spanning tree by means of `parent-request` messages sent from nodes to connected nodes until a root (the leader) of the tree is identified. Contention may arise when two nodes simultaneously send `parent-request`s to each other, and it is broken by nondeterministic back-off and retry. Following the authors of [4], we model the TIP algorithm as an untimed I/O automaton *TIP* which performs all the operations of the algorithm (sending `parent-request` messages, breaking contention, etc.) through invisible actions, and triggers its only visible action `root` only when a leader is identified.

For correctness, the *TIP* automaton must satisfy two properties: (a) at any given point in time there is at most one leader, and (b) in any execution at most one leader is ever elected, i.e., the `root` action occurs only once. Property (a) is an invariant of *TIP* and has been proved both directly in PVS by the authors of [4] and in TAME [3]. Property (b) is not an invariant, but it is captured by the executions of the simple automaton *SPEC* from [4]. The *SPEC* automaton has only one action: a visible action called `root` that is disabled after its first occurrence. By proving a that there exists a refinement from *TIP* to *SPEC*, we establish that all traces of *TIP* are included in the set of traces of *SPEC*, which in turn proves property (b).

Figure 5 shows our refinement template instantiated with the automata *TIP* and *SPEC*. 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 *SPEC* and the *TIP* automata; `amap` is a map from the actions of *TIP* to the actions of *SPEC*, and `ref` is the refinement map 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.

3.3 Applying the refinement strategy in the TIP case study

For the *TIP* example, **PROVE_REFINEMENT** divides the proof of `tip_refinement_thm` into the base case and the induction step. The base case sequent, which is handled by **SETUP_REF_BASE**, is shown in Figure 7, in

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

Fig. 7. Initial base case sequent for `tip_abstraction`.

which `TIP_decls.start` and `SPEC_decls.start` are the start predicates of *TIP* and *SPEC*, respectively. The induction step is split up into six branches by **START_ENABLEMENT_PROOF**, one for each of the six action subtypes in the `TIP_decls.actions` datatype. Corresponding to each visible action, two subgoals, for enablement and congruence, are generated. Figures 8 and 9 show the two subgoals generated for the (visible) `nu` action in *TIP*.

As seen in the saved proof for the *TIP* case study (Figure 10) all but two parts of the inductive goal for the `root` action—the specific enablement subgoal and the congruence subgoal—were resolved by this strategy automatically. Proving the `root` specific enablement subgoal required using two invariant properties of *TIP* (invariants 13 and 15 from [4]), proved earlier with TAME. (Informally, invariant 13 says that a root node has the property that all its edges connect it to its children, and invariant 15 says that at most one node has this property.) The `root` congruence subgoal required **INST_IN**. The

```
[-1,(reachable C.prestate)]
      reachable(sC_theorem)
[-2,(enabled C.action)]
      enabled(nu(timeofC_action), sC_theorem)
  |-------
{1,(enabled A.action)}
      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) #))
```

Fig. 8. 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) #) =
      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) #))
```

Fig. 9. Initial congruence sequent for the action `nu` in *TIP*.

`root` specific enablement and congruence subgoals both required the TAME "it is now trivial" step **TRY_SIMP** (see Figure 1) to complete.

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 10 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 10 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

```
(""
 (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))))
```

Fig. 10. TAME refinement proof for *TIP*/*SPEC*.

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 [2]).

3.4 Applying the refinement strategy in the RPC case study

Our second case study for refinement proofs concerns the specification and implementation of the memory component of a remote procedure call (RPC) module taken from [16]. 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 map `ref` is a bijection and the action map `amap` is an injection. As noted in [16], a weak refinement from *REL_MEM* to *MEM*, suffices to establish trace inclusion. We state this weak refinement property by instantiating the weak refinement property theory template in a manner analogous to that in the *TIP*/*SPEC* example. In the proof in this case, all but the base case and one of the induction branches were resolved automatically by **PROVE_REFINEMENT**, and these remaining goals were easily discharged with **TRY_SIMP**.

4 Strategies for forward simulation proofs

In this section, we present the strategies we have developed for proving forward simulations. We illustrate the application of these strategies by proving time bounds for a failure detector and for a two process race system.

4.1 Design of the forward simulation strategies

We have developed two generic strategies for aiding forward simulation proofs: **PROVE_FWD_SIM** and **FWD_SIM_ACTION_REC**. These strategies use new substrategies together with some of the substrategies discussed in the previous section. **PROVE_FWD_SIM** is similar to **PROVE_REFINEMENT** in that it first breaks down a forward simulation theorem (Figure 4) into its base case and induction step, and then splits the induction step into cases for the individual action subtypes of the concrete automaton. The **FWD_SIM_ACTION_REC** strategy is meant to be applied to the individual action branches produced by **PROVE_FWD_SIM**; it is used to prove the `A.invisible_seq_trans` or the `A.time_seq_trans` predicates (see Section 2.4) in the individual action branches. This strategy takes an action sequence $\sigma = a_1, a_2, \ldots, a_n$, a starting state s_1, and a known target state s_2, and produces the following set of subgoals:

- $s_2 = \mathtt{trans}(a_n, \mathtt{trans}(a_{n-1}, \mathtt{trans}(a_{n-2}, \ldots, \mathtt{trans}(a_1, s_1) \ldots)))$,
- For each action a_i in σ, a_i is not `visible`, and
- For each $i \in 1, \ldots, n$, a_i is `enabled` in $\mathtt{trans}(a_{i-1}, \mathtt{trans}(a_{i-2}, \ldots, s) \ldots)$.

The strategy then discharges some of these subgoals by expanding definitions (e.g., `visible` and `enabled`) and then simplifying. The remaining non-trivial subgoals are then presented to the user with properly labeled sequents. To illustrate the details of operation of these strategies, we next present another case study: a failure detector algorithm.

4.2 Proving a time bound in a failure detector

This case study uses a forward simulation relation to prove the time bound of a failure detector taken from [8]. The failure detector implementation consists of three components: (1) a sending process P which `sends` a heartbeat message every u_1 time units as long as it has not failed, (2) a timed channel C which delivers to T each of the messages sent by P within b time units after it is sent, and (3) a timeout process T which performs a `timeout` action if it does not receive any message over a time interval longer than u_2 units. The sending process P fails when an externally controlled `fail` action occurs and stops sending the messages. As a result of the `timeout` action, the process T *suspects* P to have failed. The implementation is modeled as an automaton called *TIMEOUT* with the two visible actions `timeout` and `fail`.

Assuming $u_2 > u_1 + b$, we are interested in proving two properties: (a) safety: T suspects P implies that P has really failed, and (b) timeliness: if P fails then it is suspected by T within $b+u_2$ time units. The safety property (a) is an invariant of *TIMEOUT*, and can be proved using the invariant strategies of TAME. The timeliness property (b) is modeled as a simple specification automaton *TO_SPEC* which has just two actions, `fail` and `timeout`, in addition to the time passage action. The automaton *TO_SPEC* simply triggers a `timeout` action within $u_2 + b$ time units of the occurrence of a `fail` action. To

```
MC: THEORY = automaton :-> TIMEOUT_decls
MA: THEORY = automaton :-> TO_SPEC_decls

rel(s_C: MC.states, s_A: MA.states):bool =
  failed(s_C) = failed(s_A) AND
  suspected(s_C) = suspected(s_A) AND
  now(s_C) = now(s_A)  AND
  IF (not failed(s_A))
     THEN inftime?(last_timeout(s_A))
     ELSE IF nonemptyqueue?(queue(s_C))
       THEN last_timeout(s_A) >= last_deadline(s_C) + u_2
       ELSE last_timeout(s_A) >= t_clock(s_C)
     ENDIF
  ENDIF
```

Fig. 11. The simulation relation for the failure detector.

show that *TIMEOUT* implements *TO_SPEC*, we proved that the relation `rel` between *TIMEOUT* and *TO_SPEC* defined in Figure 11 is a forward simulation relation. In this definition, *TIMEOUT* is represented by `MC`, *TO_SPEC* is represented by `MA`, `last_timeout` is the deadline for the `timeout` action of *TO_SPEC*, `last_deadline` is the deadline for the delivery of the last message in the channel, and `t_clock` is the deadline for timing out when no interim message is received.

4.3 Applying forward simulation strategies to TIMEOUT

For proving the *TIMEOUT/TO_SPEC* simulation property, we applied the **PROVE_FWD_SIM** strategy to the forward simulation theorem. This application produces a base case subgoal and one subgoal for each of the five actions of `MC`: `nu`, `send`, `receive`, `fail`, and `timeout`. The Base case is handled by **SETUP_REF_BASE**. For each action a of `MC`, the subgoals produced by **PROVE_FWD_SIM** correspond to proving that the relation `rel` is *preserved*, that is (in the notation used in the `forward_simulation` property theory in Figure 4 on page 8), (1) there exists an action sequence of `MA` starting from `s_A` that leads to the state `s3_A`, and (2) given that `rel(s_C, s_A)` holds, `rel(s1_C, s3_A)` also holds. The second subgoal is common to all actions, but the subgoals produced for proving (1) depend on the type of the action a. For example, the time passage action `nu` produces a `time_seq_trans` subgoal (see Figure 12), the invisible action *send* produces an `invisible_seq_trans` subgoal, and the visible action `timeout` produces two `invisible_seq_trans` subgoals and two additional subgoals to show that the `timeout` action of `MA` takes `s1_A` to `s2_A`.

For each action, showing that the post-states are related means that we have to show that the four conjuncts in `rel` are satisfied. This leads to four subgoals in each action branch. Some of these subgoals are trivial; others require the application of some previously proved invariants. The last subgoal requires us to prove inequalities involving real expressions; for this, we have found the Field [13] and the Manip [5] strategy packages to be useful.

```
timeout_fw_simulation_thm.2 :
;;; Case nu(t)
{-1,(finduct)}
     MC.reachable(s_C)
{-2,(finduct)}
     MA.reachable(s_A)
{-3,(finduct)}
     ref(s_C, s_A)
{-4,(finduct)}
     MC.enabled(nu(t), s_C)
{-5,(finduct)}
     s1_C = MC.trans(nu(t), s_C)
  |-------
{1,(finduct)}
     EXISTS (s3_A: MA.states): MA.time_seq_trans(s_A, s3_A, dur(t))
        & ref(s1_C, s3_A)

timeout_fw_simulation_thm.3 :
;;; Case send(m)
...
{-4,(finduct)}
     MC.enabled(send(m), s_C)
{-5,(finduct)}
     s1_C = MC.trans(send(m), s_C)
  |-------
{1,(finduct)}
     EXISTS (s3_A: MA.states):
       MA.invisible_seq_trans(s_A, s3_A) & ref(s1_C, s3_A)

timeout_fw_simulation_thm.5 :
;;; Case timeout
...
{-4,(finduct)}
     MC.enabled(timeout, s_C)
{-5,(finduct)}
     s1_C = MC.trans(timeout, s1_C)
  |-------
{1,(finduct)}
     EXISTS (s1_A, s2_A, s3_A: MA.states):
       MA.invisible_seq_trans(s_A, s1_A)
       & MA.invisible_seq_trans(s2_A, s3_A) & ref(s1_C, s3_A)
       & MA.enabled(timeout, s1_A) & MA.trans(timeout, s1_A) = s2_A
```

Fig. 12. Sequents produced by **PROVE_FWD_SIM** for the time passage action `nu`, invisible action `send`, and visible action `timeout`.

4.4 Proving time bounds for two process race

The second case study in which we applied our strategies to prove time bounds through a forward simulation relation is the two process race system described in [10], which we will call *RACE*. The automaton *RACE* models two processes running in parallel. The process `main` updates the counter or produces a `report` action within every time interval $[a_1, a_2]$. The second process produces a `set` action within the time interval $[b_1, b_2]$. The counter is initially set to zero, and it is incremented by `main` until the occurrence of the `set` action, from which point onward, the counter is decremented. Once the `main` process counts down to zero a `report` action is triggered.

The property of interest here is the upper and lower time bounds on the occurrence of the `report` action. The upper bound is given by $b_2+a_2+b_2a_2/a_1$. The intuition behind this bound as follows: in order to maximize the value of the counter until the `set` action occurs, the `main` process should increment `counter` every a_1 time. Thus, the value of `counter` when the `set` action occurs is b_2/a_1. Thereafter, the maximum time taken to decrement `counter` down to zero is a_2b_2/a_1. The latest time when the `set` action occurs is b_2, and therefore the upper time bound for `report` is $a_2 + b_2 + a_2b_2/a_1$. Reasoning similarly, one can show that the lower bound for the `report` action is $b_1 + (b_1 - a_2)a_1/a_2$ if $a_2 < b_1$, and a_1 otherwise. These time bounds on `report` are specified as a simple abstract automaton *RACE_SPEC* which triggers a `report` action within the above time bounds.

The forward simulation relation `rel` used to prove that *RACE* implements *RACE_SPEC* is shown in Figure 13. In the definition of `rel`, `first_`*action*

```
MC: THEORY = automaton :-> RACEdecls
MA: THEORY = automaton :-> RACE_SPEC_decls
rel(s_C: MC.states, s_A: MA.states):bool =
  now(s_A) = now(s_C)  AND
  reported(s_A) = reported(s_C) AND
  NOT flag(s_C) AND last_main(s_C) < first_set(s_C)
    IMPLIES first_report(s_A) <=  first_set(s_C) +
      (count(s_C) + (first_set(s_C) - last_main(s_C))/a2)*a1  AND
  flag(s_C) or last_main(s_C) >= first_set(s_C)
    IMPLIES first_report(s_A) <= first_main(s_C) + count(s_C)*a1  AND
  NOT flag(s_C) AND first_main(s_C) <= last_set(s_C)
    IMPLIES last_report(s_A) >= last_set(s_C) +
      (count(s_C) + 2 + (last_set(s_C) - first_main(s_C))/a1)*a2  AND
  NOT reported(s_C) AND (flag(s_C) OR first_main(s_C) > last_set(s_C))
    IMPLIES last_report(s_A) >= last_main(s_C) + count(s_C)*a2
```

Fig. 13. Simulation relation for *RACE* and *RACE_SPEC*.

and `last_`*action* denote the lower and the upper bounds on the time of occurrence of *action*, respectively. This simulation relation is more complex than that for the *TIMEOUT/TO_SPEC* example because (a) it captures both the upper and the lower time bounds, and (b) the relation between the states differs depending on whether or not `flag` has been set by the `set` action. The structure of the proof is similar to that in the *TIMEOUT/TO_SPEC* example, and so our high level **PROVE_FWD_SIM** strategy successfully breaks up the simulation proof into subgoals for the individual actions. Applying **FWD_SIM_ACTION_REC** with the proper arguments instantiates and simplifies the action branches. Generally, each action branch leads to six subgoals corresponding to the six high level conjuncts in the simulation relation, but the trivial subgoals (e.g., first two subgoals) are discharged automatically by the strategy. The remaining subgoals required the application of previously proved invariants and reasoning about inequalities involving real expressions, in which we found the Field and the Manip strategies to be useful.

5 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 [14], who also developed an associated verification framework. Example proofs of forward simulation have been done for at least simple example automata 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. [4]. However, to our knowledge, no one has developed "generic" PVS strategies to support proving abstraction properties with PVS.

Our work is related to the tools being developed for the TIOA project [7,6]. The TIOA to PVS translator, which is currently under development, produces PVS specifications of timed (or untimed) I/O automata in the style described in this paper. The translator and our strategies are designed to mask the details of the PVS theorem prover, so that the user can specify a TIOA and prove its properties in PVS without learning the details of the PVS language and prover.

6 Conclusions and future work

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 added both a reusable PVS weak refinement strategy and a reusable forward simulation strategy to TAME, and have applied these strategies to examples. 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. While previously existing TAME proof steps were useful in completing the forward simulation proofs, the TAME steps had to be supplemented, for example with proof steps from the Field [13] and Manip [5] strategy packages.

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.

As with any other product of a development project, our strategies will require more testing, tuning, and optimization after the initial conceptual phase of development whose results we have described in this paper. We have begun work on developing new strategies useful for (interactively) completing proofs of action cases. Much of this is currently being undertaken within the TIOA project. We also plan to add proof support for other abstraction properties and to continue adding new strategies for interactive proof completion.

Acknowledgements

We thank Sam Owre and Natarajan Shankar of SRI International for adding the features to PVS that have made our work possible. We also thank Nancy Lynch and Dilsun Kaynar for useful discussions on the case studies. Finally, we thank the anonymous reviewers for many helpful suggestions on improving this paper.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

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

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

[4] 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.

[5] B. Di Vito. A PVS prover strategy package for common manipulations. Technical Memorandum NASA/TM-2002-211647, NASA Langley Research Center, Hampton, VA, April 2002.

[6] Dilsun Kaynar, Nancy Lynch, and Sayan Mitra. Specifying and proving timing properties with TIOA tools. In *Work in progress session of the 25th IEEE International Real-Time Systems Symposium (RTSS-WIP)*, Lisbon, Portugal, December 2004.

[7] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917a, MIT Laboratory for Computer Science, 2004. Available at http://theory.lcs.mit.edu/tds/reflist.html.

[8] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS 2003: The 24th IEEE International Real-Time Systems Symposium*, Cancun,Mexico, December 2003.

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

[10] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[11] 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.

[12] Sayan Mitra and Myla Archer. Developing strategies for specialized theorem proving about untimed, timed, and hybrid I/O automata. In *Proceedings of the First International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, Rome, Italy, Sept. 8 2003. NASA Proceedings NASA/CP-2003-212448; also, NRL Memorandum Report NRL/MR/5540–0308722.

[13] C. Muñoz and M. Mayero. Real automation in the field. Technical Report Interim ICASE Report No. 39, NASA/CR-2001-211271, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, December 2001.

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

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

[16] 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.

[17] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, November 2001.