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

*Funding for this research has been provided by ONR

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.

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 a new TAME strategy for proving weak refinement, and shows its usage with examples. Finally, Section 6 discusses some related work, and Section 7 presents our conclusions and plans for future work.

2 Background

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].

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

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.

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

```
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`

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.

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.

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

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.

```
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`

A particular instance of the TAME template for stating the abstraction properties as theorems is shown in Figure 5. This theory 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.

4 Examples

In this section, we illustrate with three 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).

```
tip_abstraction: THEORY
  BEGIN

    IMPORTING automaton
    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*

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.

Failure Prone Memory Component. Our second case study concerns the specification and implementation of the memory component of a remote procedure call (RPC)

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

Figure 6: Initial base case sequent for tip_abstraction.

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.

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

So far, we have used PVS to prove two `weak_refinement` examples and one `forward_simulation` example. For `weak_refinement`, we have developed an initial strategy called **PROVE_REFINEMENT**. **PROVE_REFINEMENT** is designed to be invoked on a theorem which, like `tip_refinement_thm` in Figure 5, asserts `weak_refinement`. **PROVE_REFINEMENT** undertakes to prove the weak-refinement theorem inductively by exploiting its known structure. First, it splits a theorem into a base case and an induction step. Then, the induction step is further subdivided into cases for each individual action type of the concrete automaton.

For the *TIP* example the base case yields the sequent in Figure 6, in which `tip_decls.start` and `tip_spec_decls.start` are the start predicates of *TIP* and *SPEC*, respectively. The base case is handled by skolemizing, applying PVS's EXPAND to the definitions of `start` and `ref`, and then performing some minor simplifications. In both our refinement case studies, this sufficed to make the base case "trivial" (see Figure 9).

The induction step of the refinement proof is handled by the substrategy **REFINEMENT_INDUCTION**. This substrategy splits up the induction step into individual subgoals for each of the action types in the `actions` datatype. Then each subgoal is skolemized, and the definition of `visible` is expanded. After simplification, this gives different sets of subgoals for visible and invisible actions. For each invisible action, a single 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 are generated from lines 5 and 6 in `weak_refinement_step`. For example, Figures 7 and 8 show the two subgoals generated for the (visible) `nu` branch in *TIP*.

The (enablement) subgoal in Figure 7 is further split into subgoals for the general (timeliness) precondition and the specific precondition of the action, respectively handled by **APPLY_GENERAL_PRECOND** and **APPLY_SPECIFIC_PRECOND**, followed by simplification. The second subgoal (congruence) is handled by expanding the transition definition and repeatedly simplifying. This sequence of operations resolves many simple action cases of refinement proofs. For the remaining action cases, the user must interact with PVS, using steps such as TAME's **APPLY_INV_LEMMA**, **INST_IN** and **SKOLEM_IN** (see Figure 1), as in Figure 9 below.

Using **PROVE_REFINEMENT**, we have established the weak-refinement relations in our leader election and failure-prone memory case studies. Figure 9 shows the saved proofs, which will be better structured once **PROVE_REFINEMENT** is polished. In the leader election (*TIP/SPEC*) example, all but the base case and the inductive goals for the `root` action were resolved by the strategy automatically. The base case can be discharged with **TRY_SIMP**. Proving the `root` enablement goal required using two invariant properties (invariants 13 and 15 from [3]) *TIP*, proved earlier with TAME. The `root` congruence goal required **INST_IN**. In the failure-prone memory (*REL_MEM/MEM*) example, because of the simple relationship between *REL_MEM* and *MEM*, all but the base case and the `return` action subgoals were resolved automatically by **PROVE_REFINEMENT**, and these remaining goals were easily discharged with **TRY_SIMP**.

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

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

```
tip_refinement_thm.2.2 :
;;; Induction cases

[-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 7: Initial enablement sequent for the action nu in *TIP*.

```
;;; Induction cases
[-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 8: Initial congruence sequent for the action nu in *TIP*.

```
(""  (prove_refinement)
 (("1" (try_simp))
  ("2"
   (skolem_in "C.action enabled"" "v_1")
   (apply_inv_lemma "15" "sC_1")
   (inst_in "lemma_15" "rootVC_action")
   (inst_in "lemma_15" "v_1" "rootVC_action")
   (skolem_in "lemma_15" "e_1")
   (apply_inv_lemma "13" "sC_1" "e_1"))
  ("3" (inst_in "C.action enabled" "rootVC_action")))
```

```
("" (prove_refinement)
 (("1" (try_simp))
  ("2" (try_simp))
  ("3" (try_simp)))))
```

Figure 9: TAME refinement proofs for *TIP*/*SPEC* (left) and *REL_MEM*/*MEM* (right).

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 a reusable PVS weak refinement strategy and supporting PVS theories, and added them to TAME. In the example weak refinement proofs we have done so far, existing TAME strategies provide sufficient proof steps for interactively completing the refinement proofs. We have begun the development of a reusable forward simulation strategy.

We plan to complete the development of the forward simulation strategy and add similar support for proving other abstraction relations using 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.