Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 59–79
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Translation Templates to Support Strategy Development in PVS[1]

Hongping Lim[2]

*Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA*

Myla Archer[3]

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

Abstract

In presenting specifications and specification properties to a theorem prover, there is a tension between convenience for the user and convenience for the theorem prover. A choice of specification formulation that is most natural to a user may not be the ideal formulation for reasoning about that specification in a theorem prover. However, when the theorem prover is being integrated into a system development framework, a desirable goal of the integration is to make use of the theorem prover as easy as possible for the user. In such a context, it is possible to have the best of both worlds: specifications that are natural for a system developer to write in the language of the development framework, and representations of these specifications that are well matched to the reasoning techniques provided in the prover. In a tactic-based prover, these reasoning techniques include the use of tactics (or strategies) that can rely on certain structural elements in the theorem prover's representation of specifications. This paper illustrates how translation techniques used in integrating PVS into the TIOA (Timed Input/Output Automata) system development framework produce PVS specifications structured to support development of PVS strategies that implement reasoning steps appropriate for proving TIOA specification properties.

Keywords: Mechanical Theorem Proving, Templates, Specification Translation, Strategies, I/O Automata, Timed Automata, Hybrid Automata.

1 Introduction

The task of developing strategies for proving classes of properties in a theorem prover divides naturally into at least two phases. The first phase is the formulation for the prover of problem specifications, i.e., of settings and assertions to be proved

[1] This research is supported by AFOSR and ONR.

[2] hongping@csail.mit.edu

[3] archer@itd.nrl.navy.mil

in the settings. The second phase is the provision of techniques for guiding the prover in proving the assertions as automatically as possible.

In the formulation phase, a tension arises between convenience for the formulator and the ultimate convenience for the theorem prover. In particular, the specification formulation most natural to a user may not be the ideal formulation for reasoning about properties of the specification in a theorem prover. One way to alleviate the tension is to provide an intermediate layer between the specifier and the prover that translates specifications expressed in a form natural to the user into a form more convenient for the prover.

A natural setting for providing such an intermediate layer is in the integration of a theorem prover into a system development framework. In such a context, it is possible to have the best of both worlds: specifications that are natural for a system developer to write in the language of the development framework, and representations of these specifications that are well matched to the reasoning techniques provided in the prover. In a tactic-based prover, these reasoning techniques include the creation and use of tactics (or strategies) that can rely on certain structural elements in the theorem prover's representation of specifications.

In this paper, we focus on the integration of the theorem prover PVS [27] into the TIOA (Timed Input/Output Automata) [9] system development framework. A combination of PVS features make PVS a good choice for theorem proving support in TIOA. First, the higher order nature of PVS allows the use of function-valued state variables in representing the state of an automaton. This is useful, for example, when there are state variables parameterized by a parameter whose type is uninterpreted (e.g., in a concurrent or distributed system, a parameter of type `process`). As will be seen below, the higher order constructs in PVS also provide a convenient method of treating periods of continuous state evolution in an automaton analogously to atomic state transitions. Second, as described in [2,1], the fact that PVS saves rerunnable proof scripts and supports automated assertion labeling and proof comments facilitates the implementation, as PVS strategies, of proof steps using which users can create PVS proof scripts of properties visibly, if roughly, isomorphic to high level hand proofs. This paper describes how the translation scheme central to our integration of PVS into TIOA produces PVS specifications structured by templates to support the creation of PVS strategies of this nature implementing reasoning steps suited to proving invariant and simulation properties of TIOA specifications.

The paper is organized as follows: Section 2 discusses how the work described in this paper relates to other work. Section 3 provides some background on the TIOA toolkit and on the PVS interface TAME used to integrate PVS into the toolkit. Section 4 describes the TIOA framework and its specification language. Section 5 describes a set of templates we designed for use in the TAME representations of TIOA specifications, and explains how they facilitate reusing old and developing new PVS strategies for TAME for reasoning about specification properties. Section 6 discusses how the TIOA-to-PVS translator in the toolkit has evolved from producing nearly literal translations of TIOA specifications to producing translations that follow the templates. Section 7 discusses several example TAME strategies that rely on the templates. Finally, Section 8 summarizes our work and presents some conclusions.

2 Relation to related work

Problem formulation.

The notion that the formulation of a problem is important in automated reasoning is hardly new. It is discussed by Arvo [5] in the context of problem solving. In the context of theorem proving, it has generally been discussed in terms of best formulation for automatic theorem proving. For example, Kerber [15] considers how to formulate higher order theorems in first order logic, Kerber and Präcklein [16] consider how to best formulate first order logic problems for resolution theorem proving, and Ramachandran and Amir [25] study how to compactly represent certain first order theories in propositional logic. The work in [16] is, like our work, concerned with transforming a human-friendly representation of a problem into a form better for a theorem prover. However, rather than focusing on formulating problems for better automatic theorem proving, our work is concerned with better supporting development of strategies to simplify interactive theorem proving in a higher order logic.

Translation to a theorem prover.

Various tools have been previously developed for translating specifications in the IOA (Input/Output Automata) language [8,10], the predecessor of the TIOA language, into the language of different theorem provers, including the Larch Prover [6,11], Isabelle [28,24,22,23], and PVS [7]. A previous translator from TIOA (and hence IOA) to PVS is described in [18]. The translator described in this paper, which is derived from the translator in [18], is the first TIOA-to-PVS translator designed especially to support strategy development.

3 Background

The TIOA toolkit.

The TIOA toolkit [9] is designed to support analysis of systems based on the TIOA model [13]. The toolkit provides a front end checker for type-checking specifications written in the TIOA formal language. Back end tools of the toolkit currently being developed include a simulator [20], an interface to the UPPAAL model-checker [17], and a translator to the PVS theorem prover that produces PVS specifications of systems and their properties suitable for use with the PVS interface TAME [2,1]. The initial version of the translator to PVS was described in [18]. Recent improvements to the translator are the central subject of this paper. The TIOA toolkit is part of the TIOA system development framework (see Section 4).

The PVS interface TAME.

TAME (Timed Automata Modeling Environment) is a PVS interface designed to simplify specifying and reasoning about automata (state machines). TAME provides templates for specifications of automata and their properties, and a set of

```
vocabulary fischer_types
  types process,
  PcValue enumeration [ pc_rem, pc_test, pc_set, pc_check,
    pc_leavetry, pc_crit, pc_leaveexit, pc_reset]

automaton fischer(l_check, u_set: Real) where
  u_set < l_check ∧ u_set ≥ 0 ∧ l_check ≥ 0
  imports fischer_types
  signature
    output try(i: process)    internal test(i: process)
    output crit(i: process)   internal set(i: process)
    output exit(i: process)   internal check(i: process)
    output rem(i: process)    internal reset(i: process)
  states
    turn: Null[process] := nil,
    now: Real := 0,
    pc: Array[process, PcValue] := constant(pc_rem),
    last_set: Array[process, AugmentedReal] := constant(u_set),
    first_check: Array[process, Real] := constant(0)
  transitions
    internal test(i)                   internal reset(i)
      pre pc[i] = pc_test                pre pc[i] = pc_reset
      eff if turn = nil then             eff pc[i] := pc_leaveexit;
             pc[i] := pc_set;                turn := nil;
             last_set[i] :=
               now + u_set                output try(i)
          fi                               pre pc[i] = pc_rem
                                           eff pc[i] := pc_test
    internal set(i)
      pre pc[i] = pc_set                 output crit(i)
      eff turn := embed(i);                pre pc[i] = pc_leavetry
          pc[i] := pc_check;               eff pc[i] := pc_crit
          last_set[i] := \infty;
          first_check[i] :=              output exit(i)
            now + l_check;                 pre pc[i] = pc_crit
                                           eff pc[i] := pc_reset
    internal check(i)
      pre pc[i] = pc_check ∧             output rem(i)
            first_check[i] ≤ now          pre pc[i] = pc_leaveexit
      eff if turn = embed(i) then          eff pc[i] := pc_rem;
             pc[i] := pc_leavetry
          else
             pc[i] := pc_test
          fi;
          first_check[i] := 0;

  trajectories
    trajdef traj
      invariant now ≥ 0
      stop when
        ∃ i: process (now = last_set[i])
      evolve
        d(now) = 1
```

Fig. 1. TIOA specification for `fischer`

mechanized proof steps that correspond to reasoning steps typical in high level hand proofs of automaton properties including invariant and simulation properties. The proof steps are implemented as PVS strategies. Through building automatic translators of specifications to PVS specifications that instantiate TAME templates and implementing additional, setting-specific TAME proof steps as PVS strategies, TAME has been adapted to provide theorem proving support in several settings [3].

4 The TIOA framework and its specification language

This section provides an overview of the TIOA toolkit and its specification language, using the TIOA description of the Fischer mutual exclusion algorithm in Figure 1 to illustrate the language. A more complete description of the language can be found in the TIOA User Manual and Reference Guide [9].

The TIOA system development framework [4] provides an environment and toolkit

[4] Under the name Tempo, a beta release of this framework was first made available in August, 2006 at

```
invariant of fischer:
  ∀ k: process (pc[k] = pc_set ⇒ (last_set[k] ≤ (now + u_set)))
invariant of fischer:
  ∀ k: process (now ≤ last_set[k])
invariant of fischer:
  ∀ k: process (pc[k] = pc_set ⇒ last_set[k] ≠ \infty)
invariant of fischer:
  ∀ i: process ∀ j: process
    (pc[i] = pc_check ∧ turn = embed(i) ∧ pc[j] = pc_set
        ⇒ first_check[i] > last_set[j])
invariant of fischer:
  ∀ i: process ∀ j: process
    (pc[i] = pc_leavetry ∨ pc[i] = pc_crit ∨ pc[i] = pc_reset
        ⇒ turn = embed(i) ∧ pc[j] ≠ pc_set)
invariant of fischer:
  ∀ i: process ∀ j: process (i ≠ j ⇒ pc[i] ≠ pc_crit ∨ pc[j] ≠ pc_crit)
```

Fig. 2. Invariants of `fischer` in TIOA form

for the specification, analysis, and refinement of distributed and concurrent systems. TIOA specifications model a system as an automaton with a set of states, one or more initial states, actions that cause state transitions, and *trajectories*. The TIOA specification language extends the IOA (Input/Output Automata or I/O Automata) language [8,10], which has been in use (initially informally) for nearly two decades (see, e.g., [19,12,22,26]), by adding constructs for defining trajectories that describe how a system state can evolve as the result of time passage. Complex systems can be modeled as a composition of automata; like I/O Automata, Timed I/O Automata can be composed by joining output actions to input actions.

A TIOA specification consists of the definition of one or more automaton models, together with the definition of properties of interest of these automata and, if needed, a *vocabulary* in which types, constants, and operators referred to in the automaton definitions are declared. With some exceptions (such as enumerated types), the semantics of the declarations in a specification vocabulary used in analysis of the specification is provided by way of the analysis tool being used. Thus, when PVS is applied to proving that certain properties of interest hold for automata specified in TIOA, the vocabulary takes its semantics from some appropriate PVS theory.

The TIOA specification language is designed to be is clear and concise, and to allow a user to define an automaton model by providing the necessary information in a natural way. In a TIOA specification (see Figure 1), the vocabulary required is declared using the `vocabulary` keyword, each automaton description is declared using the `automaton` keyword, and automaton properties are declared using the keywords `invariant` and/or `forward simulation` (see Figure 2).

The main components of an automaton description are the signature and the definitions of the states, transitions, and trajectories. To permit use of a vocabulary, an automaton imports it. Lines 1–4 of Figure 1 shows how the types `process` and `PcValue` are introduced by the vocabulary `fischer_types` imported by the automaton `fischer` in line 8. The automaton can be parameterized, with a `where` clause constraining the parameter values, as illustrated in lines 6–7. The `signature` of an automaton defines the set of internal (`internal`) and external (`input` and `output`) actions, together with the parameters the actions may take (see lines 9–13). State variables are declared using the `states` keyword. As shown in lines 14–19, the

http://www.veromodo.com.

declaration of each variable of `fischer` specifies its name, type, and initial value. Specifying the initial value of a variable is optional; the TIOA language also allows `initially` clause to constrain, or further constrain, the variable values in a start state. No `initially` clause is needed in the specification of `fischer`.

After the keyword `transitions`, the transitions triggered by the actions declared in the signature are specified in a precondition-effect style. The precondition `pre` of a transition asserts the conditions when the transition can take place, while the effect `eff` contains a small program specifying how the state variables are modified by the transition (see lines 20–46).

The trajectory definitions for the automaton follow the keyword `trajectories`. Each trajectory definition (see lines 47–53) consists of: 1) an optional `invariant` state predicate which must hold throughout the trajectory, 2) a stopping condition—a state predicate which ends the trajectory as soon as it is true—specified by the `stop when` clause, and 3) an `evolve` clause stating how the values of the state variables evolve over time. The `evolve` clause for the trajectory `traj` in Figure 1 indicates that the only variable that changes as `traj` evolves is `now`, at the (obvious) rate of 1 unit per time unit.

A state invariant property of an automaton can be specified as an `invariant`. An implementation relationship between a pair of automata [13] can be defined as a `forward simulation` from one to the other. Figure 2 shows the main state invariants of the automaton `fischer` in TIOA.

The TIOA language includes a type name `AugmentedReal` that refers to the real numbers augmented by (positive) infinity (denoted `\infty`). The language also allows for the "lifting" of any type to add a new "bottom" (undefined) element `nil`. Thus `turn`, which indicates the process whose turn it is to enter the critical region, has type `Null[process]`, the lifting of type `process`, and initial value `nil`, and `embed(i)` coerces `i` of type `process` to type `Null[process]`.

5 PVS templates for strategy support

As described in detail in [4], the PVS representations of TIOA specifications produced by the TIOA-to-PVS translator follow a variant of the automaton template used by TAME [2,1] and the TAME property templates, including the forward simulation template described in [21]. As a result, the PVS proof support provided in the TIOA toolkit includes all of the standard TAME strategies for proofs of properties of I/O automata described in [2,1,21].

Two major TAME strategies for proofs of properties of I/O automata are the strategies **auto_induct** and **prove_fwd_sim**. The strategy **auto_induct** is used to perform the initial stages of the proof of a state invariant by induction, while **prove_fwd_sim** does the same for a proof of forward simulation. Both strategies rely heavily on both the naming conventions and the structure conventions followed in the automaton and lemma templates. In particular, both **auto_induct** and **prove_fwd_sim** rely on the names `start`, `trans`, and `enabled`, respectively used for the start state predicate, transition function, and precondition predicate in the automaton template; **auto_induct** relies on TAME's standard invariant lemma template (see Figure 4 for examples of Inv_*invname*):

```
FORALL(s:states):  reachable(s) => Inv_invname
```

and the strategy **prove_fwd_sim** relies on both the (much more complex) definition structure and standard name of the `forward_simulation` property.

One important use of structure conventions is the assignment of labels to assertions in a proof goal. This is illustrated by TAME's PVS template used for the predicate `start` (see Figure 3 and Section 6.2):

```
start(s:states):bool =
     s = s WITH [ <initial values of some or all state variables> ]
   & <optional additional constraints> ;
```

This template allows **auto_induct** to separate the assertion `start(s)`, which is the hypothesis of the base case in an induction proof, into two separate hypotheses, labeled `start-state` and `start-constraints`. A strategy designed to automate the proof of the base case can then refer to either or both of these labels.

Because trajectories describe evolution of the state as time passes, they can be thought of as "extended transitions" over time (usually, continuous paths through the state space). In fact, as is discussed in more detail in Section 6.3, trajectories in a TIOA specification are represented in TAME as automaton actions with information about their invariant, stopping condition, and evolution captured in their precondition. As with the template for `start`, TAME's PVS template for the precondition of a trajectory action provides a structure that supports useful labeling:

```
enabled(a:actions, s:states):bool = CASES a OF
    traj-name(delta_t, F):
      (FORALL (t:(interval(zero,delta_t))):  traj_invariant(a)(F(t)))
        AND (FORALL (t:(interval(zero,delta_t))):  traj_stop(a)(F(t)) => t = delta_t)
        AND (FORALL (t:(interval(zero,delta_t))):  F(t) = traj_evolve(a)(t, s)),
    .  .  .
ENDCASES
```

The TAME strategy **apply_specific_precond**—which, in an induction proof, introduces into the hypothesis of an induction step subgoal the details of the precondition of the current action—can take advantage of this organization of the precondition into a three-part conjunction to separate it into three hypotheses and give each a separate label. Afterwards, these labels can be used to focus each of the three TAME strategies **(apply_traj_invariant** *timeval***)**, **(apply_traj_stop** *timeval***)**, and **(apply_traj_evolve** *timeval***)** on just its relevant conjunct of the precondition, to define respectively for the given time value *timeval*: 1) the value of the trajectory invariant, 2) the value of the trajectory stopping condition, and 3) the state to which the trajectory has evolved. The ability to separate concerns in this way also makes it possible to use **(apply_traj_stop** *timeval***)** and **(apply_traj_evolve** *timeval***)** to define a relatively simple TAME strategy for reasoning about deadlines.

Besides supporting a helpful labeling scheme, the structure of the trajectory action precondition template facilitates the separation of concerns at an early point in reasoning by avoiding the use of a shared universal quantifier for the three parts of

```
fischer_decls : THEORY BEGIN
[. . .]
l_check: real
u_set: real
const_facts: AXIOM
  u_set < l_check AND u_set >= 0 AND l_check >= 0

states: TYPE = [#
  turn: lift[process],
  now: real,
  pc: array[process -> PcValue],
  last_set: array[process -> time],
  first_check: array[process -> real] #]

start(s: states): bool = s=s WITH [
  turn := bottom,
  now := 0,
  pc := (LAMBDA(i_0: process): pc_rem),
  last_set :=
    (LAMBDA(i_0: process): fintime(u_set)),
  first_check := (LAMBDA(i_0: process): 0)]

f_type(i, j: (fintime?)):
  TYPE = [(interval(i, j))->states]

actions: DATATYPE BEGIN
  nu_traj(delta_t: {t: (fintime?) | dur(t)>=0},
          F: f_type(zero, delta_t)): nu_traj?
  test(i:process):test?      set(i:process):set?
  check(i:process):check?    reset(i:process):reset?
  try(i:process):try?        crit(i:process):crit?
  exit(i:process):exit?      rem(i:process):rem?
END actions

visible?(a:actions):bool =
  try?(a) OR crit?(a) OR exit?(a) OR rem?(a)
timepassageaction?(a:actions):bool = nu_traj?(a)
length(a:(timepassageaction?)):real =
   dur(delta_t(a))

traj_invariant(a:(timepassageaction?))(s:states):
  bool = CASES a OF nu_traj(delta_t, F):
           now(s) >= 0 ENDCASES
traj_stop(a:(timepassageaction?))(s:states):bool =
  CASES a OF nu_traj(delta_t, F):
    EXISTS (i: process):
      fintime(now(s)) = last_set(s)(i)
  ENDCASES
traj_evolve(a:(timepassageaction?))(t:(fintime?),
             s:states):states =
  CASES a OF nu_traj(delta_t, F):
      s WITH [now := now(s) + 1 * dur(t)]
  ENDCASES

enabled(a:actions, s:states):bool =
 CASES a OF
   nu_traj(delta_t, F):
       (FORALL (t:(interval(zero,delta_t))):
          traj_invariant(a)(F(t)))
        AND (FORALL (t:(interval(zero,delta_t))):
               traj_stop(a)(F(t)) => t = delta_t)
        AND (FORALL (t:(interval(zero,delta_t))):
               F(t) = traj_evolve(a)(t, s)),
   test(i):pc(s)(i)=pc_test, set(i):pc(s)(i)=pc_set,
   check(i):pc(s)(i)= pc_check AND
            first_check(s)(i)<=now(s),
   reset(i): pc(s)(i) = pc_reset
   try(i):pc(s)(i)=pc_rem, exit(i):pc(s)(i)=pc_crit,
   crit(i):pc(s)(i)=pc_leavetry,
   rem(i):pc(s)(i)=pc_leaveexit,
  ENDCASES

trans(a:actions, s:states):states = CASES a OF
 nu_traj(delta_t, F): F(delta_t),
 test(i): s WITH
  [last_set := IF turn(s) = bottom THEN
     last_set(s) WITH [(i):=fintime(now(s)+u_set)]
    ELSE last_set(s) ENDIF,
   pc := IF turn(s) = bottom THEN
    pc(s) WITH [(i) := pc_set] ELSE pc(s) ENDIF],
 set(i): s WITH [turn := up(i),
    last_set := last_set(s) WITH [(i):=infinity],
    first_check := first_check(s) WITH
         [(i) := now(s) + l_check],
    pc := pc(s) WITH [(i) := pc_check]],
 check(i): s WITH
  [first_check := first_check(s) WITH [(i) := 0],
    pc := IF turn(s) = up(i) THEN
            pc(s) WITH [(i) := pc_leavetry]
          ELSE pc(s) WITH [(i) := pc_test] ENDIF],
 reset(i): s WITH [turn := bottom,
            pc := pc(s) WITH [(i):=pc_leaveexit]],
 try(i): s WITH [pc:=pc(s) WITH [(i):=pc_test]],
 crit(i): s WITH [pc:=pc(s) WITH [(i):=pc_crit]],
 exit(i): s WITH [pc:=pc(s) WITH [(i):=pc_reset]],
 rem(i): s WITH [pc:=pc(s) WITH [(i):=pc_rem]]
ENDCASES

IMPORTING timed_auto_lib@time_machine
  [states, actions, enabled, trans, start, visible?,
   timepassageaction?, length]
END fischer_decls
```

Fig. 3. TAME representation of `fischer`

the precondition. A shared universal quantifier would require a shared instantiation of the variable `t`, even in cases where one might desire a different instantiation for different parts of the precondition.

The structure of the template used for the transition function `trans` (see Section 6.5) also provides a separation of concerns:

```
trans(a:actions, s:states):states = CASES a OF
    traj-name_1(delta_t,F): F(delta_t),
    . . .
    action_1:  s WITH [ <updates to individual variables> ],
    . . .
    action_n:  s WITH [ <updates to individual variables> ],
ENDCASES
```

Representing `trans` using this template allows the updated values of individual variables in the poststate of a discrete transition to be accessed separately using a standard sequence of PVS commands. These values can then be reasoned about in isolation, without having to reason about the values of other variables as well.

Section 6 discusses further details of the example templates in this section and

```
Inv_0(s:states):bool =
  FORALL (k: process):
    pc(s)(k) = pc_set => last_set(s)(k) <= fintime(now(s) + u_set)

Inv_1(s:states):bool =
  FORALL (k: process): fintime(now(s)) <= last_set(s)(k)

Inv_2(s:states):bool =
  FORALL (k: process): pc(s)(k) = pc_set => last_set(s)(k) /= infinity

Inv_3(s:states):bool =
  FORALL (i: process, j: process):
    pc(s)(i) = pc_check AND turn(s) = up(i) AND pc(s)(j) = pc_set
      => fintime(first_check(s)(i)) > last_set(s)(j)

Inv_4(s:states):bool =
  FORALL (i: process, j: process):
    pc(s)(i) = pc_leavetry OR pc(s)(i) = pc_crit OR pc(s)(i) = pc_reset
      => turn(s) = up(i) AND pc(s)(j) /= pc_set

Inv_5(s:states):bool =
  FORALL (i: process, j: process):
    i /= j => pc(s)(i) /= pc_crit OR pc(s)(j) /= pc_crit
```

Fig. 4. TAME representation of `fischer` invariants

some additional templates, along with the evolution of the TIOA-to-PVS translator towards providing template support for strategy development.

6 Translating TIOA specifications into PVS templates

This section begins with an overview of the current translation scheme employed by the TIOA-to-PVS translator. It then discusses the issues involved with previously used (or considered) translation schemes and, for each issue, discusses how it was solved by updating the translation scheme to follow templates updated to improve strategy support (including those discussed in Section 5). An important goal of the TIOA-to-PVS translator is to avoid forcing the user to change the form of a TIOA specification to support adherence of its PVS translation to a particular template. As will be seen below, with some minor exceptions, we have achieved this goal. For a more complete description of the translator and the translation scheme, we refer the reader to [18].

6.1 Overview of the translation scheme

As indicated above and in in Sections 3 and 5, the TIOA-to-PVS translation scheme is targeted to TAME specification templates; hence, we will also speak of translation into TAME. The TAME templates structure the specification of an automaton and its parts and properties, and, in conjunction with the TAME libraries of datatype and other definitions, provide definitions of TIOA concepts in PVS. The translator instantiates the TAME automaton template with the states, actions, transitions and trajectories of an input TIOA specification automatically. Both the (discrete) actions and transitions and the trajectory definitions in a TIOA specification are translated as actions and transitions (with associated preconditions) in TAME, with trajectories becoming time passage actions parameterized by the time they consume and by their “state evolution” function mapping a time interval to a path through the state space. The state transition associated with a trajectory action in TAME is computed from the evolution function and the time passage, and the precondition of the trajectory action captures the constraints on the trajectory represented in its TIOA definition.

Figures 3 and 4 show, respectively, the TAME representation of the TIOA description of `fischer` in Figure 1 and the `fischer` invariants in Figure 2 generated by the translator, illustrating the translation scheme. Automaton parameters are declared as constants, while the `where` clause is translated as an axiom named `const_facts` (lines 3–6). State variables are declared within a record type `states` (lines 8–13). A `start` predicate is defined to be true for start states. Action signatures are declared in the data type `actions` (lines 26–33). A `visible?` predicate is defined to be true for external actions, while the predicate `timepassageaction?` is defined to be true for time passage actions. The predicate `enabled` asserts the preconditions of the actions, while the function `trans` represents the transition function which returns the post-state obtained by applying an action to a given pre-state (lines 55–97). As noted above, a trajectory definition in TIOA is translated into TAME as a time passage action parameterized by its evolution function `F` and a time increment `delta_t` (lines 27–28). The function `F` is of type `f_type` which maps a given time interval into the state space (lines 23–24). For describing time passage transitions, three functions are defined to represent the invariant, stopping condition and the evolve clause of the corresponding trajectory definition (see `traj_invariant`, `traj_stop`, and `traj_evolve` in lines 41–53). Within the `enabled` clause of the time passage action, the invariant, stopping condition and evolve clause are asserted for all elapsed times within `delta_t` (lines 57–63). The `trans` function for the time passage action simply returns the state obtained by applying the function `F` to the elapsed time `delta_t` (line 74).

An invariant is translated as an assertion in PVS (with a name of the form `Inv_`*invname*) together with a lemma in PVS (conforming to TAME's standard invariant lemma template—see Section 5) stating that the assertion of the invariant holds throughout all reachable states of the automaton. Figure 4 shows only the assertions, omitting the (boilerplate) invariant lemmas.

6.2 Start states

The issue. In a previous version of the TIOA description of `fischer`, the start state is written in the following form, in which the initial values of the arrays `pc`, `last_set`, `first_check` are asserted with an `initially` clause:

```
states
  turn:  Null[process] := nil,
  now:  Real := 0,
  pc:  Array[process, PcValue],
  last_set:  Array[process, AugmentedReal],
  first_check:  Array[process, Real]
  initially ∀ i:  process (pc[i] = pc_rem) ∧
            ∀ i:  process (last_set[i] = u_set) ∧
            ∀ i:  process (first_check[i] = 0)
```

Correspondingly, the start state was previously translated as a conjunction of the equalities of each variable to its initial value, together with the `initially` clause:

```
start(s: states): bool =
  turn(s) = bottom AND
  now(s) = 0 AND
  FORALL(i: process): pc(s)(i) = pc_rem AND
  FORALL(i: process): last_set(s)(i) = u_set AND
  FORALL(i: process): first_check(s)(i) = 0
```

This scheme asserts the start state condition using a conjunction of clauses, and asserts the initial values of function (i.e., array) valued state variables in terms of assertions universally quantified over their arguments (indices). Thus, when (as is often the case) there are state variables of function type, reasoning about the start state at the state variable level is not supported, and automated support for the reasoning about the start state is complicated by the presence of quantifiers.

Solution. To solve this issue in our current translation scheme, we made a change in the TIOA language to allow an array to be assigned an initial value. In addition, we use the (new) TIOA operator `constant` in the TIOA description to define an array in which all elements have the same value as the given operand (see lines 15–19 of Figure 1). The use of the `constant` operator avoids the use of the universal quantifiers, and allows translation of array values into `LAMBDA` expressions in PVS (see lines 18-21 of Figure 3). Although in this instance the form of the TIOA specification was modified to facilitate the desired translation, this modification can eventually be performed automatically by a preprocessor and hidden from the user. Casting the predicate `start` as a record equality by way of the `LAMBDA` expressions instead of as a conjunction containing universally quantified clauses allows simple substitution for the start state `s` in the base case of an invariant proof.

6.3 *Trajectory definitions*

The issue. In the earlier version of the translation scheme described in [14], we translated a trajectory definition into a time passage action containing only the time interval as a parameter. The `enabled` predicate for the time passage action asserts that the invariant of the trajectory holds, and that the values of the variables stay within the limits of any stopping condition inequality. The `trans` function returns the post-state of the time passage action by incrementing the variables according to the evolve clause. The invariant, stopping condition and evolve clause are also inserted directly into `enabled` and `trans`. For example, the translation of the trajectory definition in lines 47–53 of Figure 1 using this translation scheme produces the following PVS output:

```
enabled(a: actions, s: states): bool = CASES a OF
    traj(delta_t):
      now(s)>=0 AND EXISTS(i:process): now(s)+delta_t <= last_set(s)(i),
    . . .
  ENDCASES
trans(a: actions, s: states): states = CASES a OF
    traj(delta_t): s WITH [now := now(s) + delta_t],
    . . .
  ENDCASES
```

This translation scheme, however, does not conveniently capture, for the purpose of reasoning in the theorem prover about the time passage action `traj(delta_t)`, the fact that the trajectory invariant must hold throughout the duration of the trajectory. The invariant can only be asserted either at the beginning or the end of the trajectory, but not in between; asserting the invariant at an intermediate value requires reasoning in addition about time passage actions `traj(t)` where `0<=t<=delta_t`.

Initial solution; new issue. To solve this problem, we embed the trajectory as a functional parameter of the time passage action. This approach allows us to use the functional parameter `F` to assert properties throughout the duration of the trajectory using a `FORALL` quantifier.

An initial version of this solution makes use of only a single `FORALL` quantifier, inserting the expressions of the invariant, stopping condition and evolve clause directly under the quantifier:

```
enabled(a: actions, s: states): bool = CASES a OF
    traj(delta_t, F):
      FORALL(t:(interval(zero, delta_t))):
        now(F(t)) >= 0 AND
        EXISTS(i:process): now(F(t)) = last_set(s)(i) => t = delta_t AND
        F(t) := s WITH [now := now(s) + t],
    . . .
  ENDCASES
trans(a: actions, s: states): states = CASES a OF
    traj(delta_t, F): F(delta_t),
    . . .
  ENDCASES
```

This translation scheme, however, poses problems in proofs and strategies when we only want to reason about a specific component of the trajectory definition. For example, when we only want to reason about how the evolve clause of the trajectory affects the state variables, we still have to deal with the entire universally quantified expression covering all three clauses. In addition, we have to identify the evolve clause component of the expression under the quantifier, which may not be straightforward to do, as this expression is not guaranteed to be a conjunction of three subexpressions.

Improved solution. To address these remaining problems, we further refine our translation scheme by adding another layer of abstraction based on the definitions of three functions, `traj_invariant`, `traj_stop` and `traj_evolve`, and the use of three separate `FORALL` clauses (see lines 41–53, and 57–63 of Figure 3). As mentioned in Section 5, the use of these definitions with standard names within three separate quantifiers aids the development of strategies which can pick out the respective components easily. These definitions also allow specifications containing multiple trajectory definitions to be handled without any modifications or added complications to the strategies. For example, if we have two trajectory definitions named `traj1` and `traj2`, then the PVS translation will take the form shown in Figure 5, in which additional trajectory definitions simply add more cases to each function definition.

```
traj_invariant(a:(timepassageaction?))(s:states):bool = CASES a OF
    nu_traj1(delta_t, F): . . .,
    nu_traj2(delta_t, F): . . .
  ENDCASES

traj_stop(a:(timepassageaction?))(s:states):bool = CASES a OF
    nu_traj1(delta_t, F): . . .,
    nu_traj2(delta_t, F): . . .
  ENDCASES

traj_evolve(a:(timepassageaction?))(t:(fintime?), s:states):states = CASES a OF
    nu_traj1(delta_t, F): s WITH [ . . . ],
    nu_traj2(delta_t, F): s WITH [ . . . ]
  ENDCASES

enabled(a:actions, s:states):bool = CASES a OF
    nu_traj1(delta_t, F):
      (FORALL (t:(interval(zero,delta_t))): traj_invariant(a)(F(t)))
        AND (FORALL (t:(interval(zero,delta_t))):
              traj_stop(a)(F(t)) => t = delta_t)
        AND (FORALL (t:(interval(zero,delta_t))):
              F(t) = traj_evolve(a)(t, s)),
    nu_traj2(delta_t, F):
      (FORALL (t:(interval(zero,delta_t))): traj_invariant(a)(F(t)))
        AND (FORALL (t:(interval(zero,delta_t))):
              traj_stop(a)(F(t)) => t = delta_t)
        AND (FORALL (t:(interval(zero,delta_t))):
              F(t) = traj_evolve(a)(t, s)),
    . . .
  ENDCASES
```

Fig. 5. TAME translation of multiple trajectory definitions

6.4 *Automaton parameters and the* `where` *clause*

The issue. In an earlier version of the TIOA to TAME translation scheme, the `where` clause stating the relationship among the automaton parameters was translated as an additional clause conjoined to the `start` predicate. Then, an invariant duplicating the `where` clause is specified, proved, and used in other invariants requiring the use of the assertion about the automaton parameters. This invariant is trivially proved[5], because it is by definition true in the start state, and because the values of the automaton parameters are never modified by any transitions. In particular, applied to the automaton `fischer`, the earlier translation scheme produces the following form of the `start` predicate, which has an additional clause conjoined:

```
start(s: states): bool = s=s WITH [
  turn := bottom,
  now := 0,
  pc := (lambda(i_0: process): pc_rem),
  last_set := (lambda(i_0: process): fintime(u_set)),
  first_check := (lambda(i_0: process): 0)]
  AND (u_set < l_check AND u_set >= 0 AND l_check >= 0)
```

The additional invariant and invariant lemma included in the translation of `fischer` take the form:

```
Inv_0(s:states):bool =
    u_set < l_check AND u_set >= 0 AND l_check >= 0
lemma_0: LEMMA FORALL (s:states): reachable(s) => Inv_0(s);
```

The user must then prove this invariant, and when the constraints on the automaton

[5] In particular, it can be proved by the single TAME proof command (**auto_induct**).

parameters are needed in other proofs, the user must use the TAME command (**apply_inv_lemma** "0").

Solution. To relieve the user from having to prove the additional invariant lemma for every parameterized automaton and to apply the invariant to introduce the constraints in other proofs, the translation scheme has been modified to translate the `where` clause as a separate axiom named `const_facts`. This decision allows the user to invoke the axiom directly with a standard TAME proof step (also called **const_facts**[6]) rather than introducing the information by applying an invariant lemma. It also allows separation of concerns between constraints on the start state and the parameters.

6.5 Program statements in definitions of transitions

The issue. In the TIOA language, the effects of transitions are specified in the form of programs which allow tests followed by a branch, and assignments affecting values used in later assignments. There are tradeoffs regarding where to place the burden of the computation (translator or prover) and regarding clarity of equivalence of the TAME representation of a transition to the original TIOA representation.

A partial solution. The translator currently supports two styles of translation for program statements in specifications of transition effects in a TIOA specification. The first style uses explicit substitution, as illustrated by the `trans` function in the translation in Figure 3, using symbolic computation to express the final value of every state variable in the post-state in terms of the original values of the variables in the pre-state. This substitution is performed by the translator during the process of translation.

The second style of translation preserves the structure of the statements in the original program in the effect by using a series of `LET` statements. Each `LET` statement corresponds to a statement in the original program, and modifies the state `s` accordingly. The modified state is then used as the state parameter in the subsequent `LET` statement in a similar fashion. As an example, the following code[7] shows how the effect of the transition `test(i)` in Figure 1 would be translated using `LET` statements within the `trans` function:

[6] See the command (**const_facts**) in both the proof of the base case and the proof of the induction step for `test(i_action)` in the proof of `lemma_1` of `fischer` in Appendix A.

[7] In PVS, modifications to functions, records, and arrays are indicated by `WITH` followed by a list of one or more "update assignments" in square brackets. Modification to a record is indicated by assignment to the field in the record; modification of the value of a function (or array) at some argument (or index) is indicated by an assignment to the parenthesized argument. Thus, "`s WITH [pc := pc(s) WITH [(i) := pc_set]]`" denotes the state `s` with its program counter array `pc` updated to `pc` modified to have the value `pc_set` at argument `i`. The alternate representation in Figure 3 of the effect of `test(i)` and other actions can be interpreted analogously.

```
test(i):
  LET s = IF turn(s) = bottom
          THEN
            LET s=s WITH [pc := pc(s) WITH [(i) := pc_set]] IN
            LET s=s WITH [last_set :=
                            last_set(s) WITH [(i) := fintime(now(s) + u_set)]] IN s
          ELSE s
          ENDIF
  IN s,
```

The use of explicit substitution tends to be more efficient in terms of theorem proving, because the translator has done the work of computing the final value of each variable, allowing reasoning about individual variables to be performed easily. For short programs, the explicit substitution method also produces more compact code. On the other hand, for longer programs which might have deep levels of dependencies among variables, the explicit substitution method may yield more complicated expressions. In such cases, translation using the `LET` keyword may produce a simpler translation which corresponds more clearly to the original program.

When there is a need to reason about the updated values of individual state variables after a transition, however, the use of a nested sequence of `LET` expressions may significantly complicate the reasoning. This is because for a given variable, additional proof steps will usually be required to simplify the nested `LET` expressions to the point where the update "assignments" for the variables can be accessed and computed. Since these additional proof steps for simplification are destined to form part of an application-independent strategy, they are likely to require much more computation than is needed to reason about the updated values of particular variables.

Our current solution. Currently, we have chosen to move the burden of computation outside of the theorem prover and into the translator, i.e., to use explicit substitution as the default translation method for `trans`.

6.6 *Type Correctness Conditions*

The issue. In our current translation scheme, the preconditions and transitions are defined separately, in the `enabled` predicate and the `trans` function respectively. This is done first, because it is a natural separation of concerns, and second, because it allows proofs of properties to reflect which preconditions, if any, are actually used.

But a side effect of this separation is that some unprovable Type Correctness Conditions (TCCs) may be generated by PVS as a result of the translation. As an illustration, consider the following TIOA transition, where `z` is some state variable:

```
output divide(x, y:Int)
pre y ≠ 0
eff z := x / y
```

The transition asserts in the precondition that parameter `y` is non-zero, and then proceeds to divide the parameter `x` by `y`. The translation of the above transition into the `enabled` predicate and `trans` function in PVS is as follows:

```
enabled(a: actions, s: states): bool = CASES a OF
    divide(x, y): y /= 0
  ENDCASES
trans(a: actions, s: states): states = CASES a OF
    divide(x, y): s WITH [z := x / y]
  ENDCASES
```

When we perform a type check on on `trans` in PVS, we will have to prove the TCC that `y` is non-zero for all states. However, since the precondition is now separate from the effect, we are unable to prove this TCC.

Solution. One way to resolve this issue is simply to have the translator condition the effect of the transition on it precondition in the representation of `trans`:

```
trans(a: actions, s: states): states =
  IF enabled(a, s)
  THEN CASES a OF divide(x, y): s WITH [z := x / y] ENDCASES
  ELSE s ENDIF
```

Doing so will allow the use of the precondition clause within the `enabled` predicate to resolve the TCC.

An alternative approach to handle the TCC is to have the user manually condition the effect of the transition on the precondition (or the part of it needed for type correctness) in the TIOA specification:

```
output divide(x, y:Int)
pre y ≠ 0
eff if y ≠ 0 then z := x / y fi;
```

The translation would yield the following, allowing the TCC for `x/y` to be resolved:

```
trans(a: actions, s: states): states =
  CASES a OF
    divide(x, y): s WITH [z := IF y /= 0 THEN x / y ELSE z ENDIF]
  ENDCASES
```

Either approach supports proving properties by induction over the reachable states, since the `THEN` case corresponds to the intended change of state in the TIOA specification when the precondition is satified, and the `ELSE` case corresponds to "no change of state", which is consistent with the action not being triggered when the precondition is *not* satisfied.

Since there may be no TCC to resolve in the effect of a transition, and when there is, the precondition of the transition may be stronger than the actual expression needed to resolve the TCC (e.g., the precondition of other transitions involving division by `y` could assert `y /= 0` together with several other constraints), automatically replicating the `enabled` clause in the transition function `trans` could frequently complicate the sequents in a proof with unnecessary formulas. Hence, we currently require the user to adopt the second approach of manually asserting any condition that may be needed to resolve the TCC, so long as it is implied by the precondition of the transition. This approach has worked well in the examples with which we have tested the translator: the occurrence of an unprovable TCC for `trans` has been rare; there are none in `fischer`, for example. We might adopt the first approach in future if we want to completely shield the specifier from having to modify the specifications just to avoid the generated TCCs.

6.7 Combining universal quantifiers in invariants

The issue. When an invariant of a TIOA specification contains two or more consecutive universal quantifiers, a direct translation of these quantifiers into PVS can complicate automatic reasoning in PVS. For example, it makes it difficult for the TAME strategy **auto_induct** to coordinate the skolemization of the inductive conclusion with the instantiation of the inductive hypothesis in the induction step.

Solution. The translator automatically combines the quantified variables under a single `FORALL` quantifier in the PVS output. For example, the last three invariants of the TIOA specification of `fischer` in Figure 2 contain the universal quantifiers over `i` and `j` ($\forall$`i:process` $\forall$`j:process`). The corresponding translation in PVS combines each pair of universal quantifiers into a single `FORALL (i:process, j:process)` expression, as shown in Figure 4.

7 Discussion

In this section, we provide a little more detail on how several TAME strategies take advantage of the translation templates. An example proof that illustrates the use of several of the strategies mentioned can be found in Appendix A.

The strategy base_case.

The TAME strategy **base_case** is not normally invoked directly by the user; rather, it is invoked by the strategy **auto_induct** (see Section 5) that does the initial steps of the induction proof of an automaton state invariant. In operation, **base_case** first computes the assertion representing the base case of the induction. The hypothesis of this assertion is that the start state predicate `start` holds for some state `s`. The template form of `start` is a conjunction whose first component associates explicit values with some state variables and whose second component provides additional constraints on the values of the variables (see Section 6.2). Using standard PVS steps for decomposing conjunctions and labeling the new formulas that are produced, **base_case** breaks the hypothesis into two parts labeled `start-state` and `start-constraints`. This allows **auto_induct** to continue by substituting for `s` based on the formula `start-state` and then attempting to complete the proof by applying simplifications. This discharges the base case automatically in many cases.

The strategy apply_traj_evolve.

The strategy **apply_traj_evolve**, given a time value parameter `T`, computes the new state of a trajectory after it has evolved for time `T`, provided `T` is between `0` and the duration `delta_t` of the trajectory. This strategy is used in reasoning about the transition resulting from a trajectory "action" `traj`. It relies on the TAME strategy **apply_specific_precond** having been applied first, with the result that there are hypotheses labeled `specific-precondition_part_`i, for $i = 1$ to 3. Hypothesis `specific-precondition_part_3` will be of the form:

```
(FORALL (t:(interval(zero,delta_t))): F(t) = traj_evolve(a)(t, s))
```

with traj substituted for a and prestate (the pre-state of the trajectory action) substituted for s. The strategy **apply_traj_evolve** is then able to compute the value of F(T) (representing the state after time T) by instantiating the above formula with T and then using the PVS definition expansion command to expand traj_evolve. Finally, **apply_traj_evolve** uses a PVS command to replace F(T) by its value wherever it occurs.

The strategy apply_traj_stop.

The strategy **apply_traj_stop** performs similarly to **apply_traj_evolve**. It also relies on the TAME strategy **apply_specific_precond** having been applied first, but rather than using hypothesis specific-precondition_part_3, it uses the hypothesis specific-precondition_part_2, which will be of the form:

```
(FORALL (t:(interval(zero,delta_t))): traj_stop(a)(F(t)) => t = delta_t)
```

It then expands the definition of traj_stop. The ultimate effect of **apply_traj_stop** with parameter T is to introduce the fact among the hypotheses of the proof goal that, for a given trajectory traj and time T, if the stopping condition of traj holds after time T, then the trajectory has reached its end.

The strategy deadline_reason.

The strategy **deadline_reason**, when given an absolute time deadline D as a parameter, tries to prove that, on the current trajectory traj, absolute time cannot pass beyond time D. It does this by first applying **apply_traj_stop** to time T = D - now, and then using **apply_traj_evolve** to compute F(T) so that traj_stop(traj)(F(T)) can be evaluated. (If it evaluates to true, the trajectory traj must stop after time T, i.e., at absolute time D.)

The organization of the precondition of traj through the trajectory precondition template makes it possible to define this strategy as a focused computation without any superfluous manipulation.

8 Conclusions

In this paper we have considered a particular case of a general problem: How to provide efficient theorem proving support in an interactive, higher order logic prover for establishing properties of a model of some given class, without forcing the user of the theorem prover to specify the model for the convenience of the prover rather than in a form natural to the user. In the case of automata models of systems, we have shown that this can be done by translating specifications written in a language designed for specifying automata (TIOA) into the language of a theorem prover (PVS) while adhering to a set of templates governing how various aspects of the automaton model are represented in the theorem prover. We have discussed how both the structural and naming conventions captured in these templates can be used to advantage in developing efficient domain specific proof steps aimed at interactive reasoning about the aspects of an automaton model for which there are templates.

The general principle we have followed of designing the translator to convert source specifications into problem formulations that match templates convenient for analysis can no doubt be applied to advantage in other domains. An interesting question is the extent to which the connection between templates and strategies that is possible in PVS, with its ability to attach labels to formulas, can be duplicated in other higher order logic provers.

Acknowledgements

We wish to thank Ramesh Bharadwaj, Elizabeth Leonard, and the anonymous reviewers of an earlier version of this paper for their helpful comments.

References

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

[2] Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb. 2001.

[3] Myla Archer. Basing a modeling environment on a general purpose theorem prover. Technical Report NRL/MR/5546–06-8952, NRL, Wash., DC, Dec. 2006.

[4] Myla Archer, HongPing Lim, Nancy Lynch, Sayan Mitra, and Shinya Umeno. Specifying and proving properties of Timed I/O Automata in the TIOA Toolkit. In *Formal Methods and Models for Codesign (MEMOCODE 2006)*, 2006.

[5] James Arvo. Computer aided serendipity: The role of autonomous assistants in problem solving. In *Proc. of Graphics Interface '99*, pages 183–192, 1999.

[6] Andrej Bogdanov, Stephen Garland, and Nancy Lynch. Mechanical translation of I/O automaton specifications into first-order logic. In *Formal Techniques for Networked and Distributed Systems - FORTE 2002 : 22nd IFIP WG 6.1 Intern. Conf.*, pages 364–368, Houston, TX, USA, Nov. 2002.

[7] Marco Devillers. Translating IOA automata to PVS. Technical Report CSI-R9903, Computing Science Institute, University of Nijmegen, February 1999.

[8] S. J. Garland and N. A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Technical Report MIT/LCS/TR-762, MIT Laboratory for Computer Science, August 1998.

[9] Stephen Garland. TIOA User Guide and Reference Manual. Technical report, MIT CSAIL, Cambridge, MA, 2006. URL http://tioa.csail.mit.edu.

[10] Stephen Garland, Nancy Lynch, Joshua Tauber, and Mandana Viziri. IOA User Guide and Reference Manual. Technical Report MIT-LCS-TR-961, MIT CSAIL, Cambridge, MA, 2004.

[11] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[12] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, December 1994.

[13] D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O automata*. Synthesis Lectures on Computer Science. Morgan Claypool Publishers, 2005.

[14] Dilsun Kaynar, Nancy Lynch, and Sayan Mitra. Specifying and proving timing properties with TIOA tools. In *Work-In-Progress Proc. 2004 IEEE Real-Time Systems Symp. (RTSS'04)*, Lisbon, Portugal, December 2004.

[15] Manfred Kerber. How to prove higher order theorems in first order logic. Seki Report SR-90-19, Fachbereich Informatik, Universität Kaiserslautern, Germany, 1990.

[16] Manfred Kerber and Axel Präcklein. Tactics for the improvement of problem formulation in resolution-based theorem proving. Seki Report SR-92-09, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[17] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[18] Hongping Lim. Translating timed I/O automata specifications for theorem proving in PVS. Master's thesis, Mass. Inst. of Tech., Cambridge, MA, 2006. URL http://tioa.csail.mit.edu/.

[19] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.

[20] Panayiotis P. Mavromattis. TIOA Simulator Manual. February 15, 2006. URL http://tioa.csail.mit.edu/public/Tools/simulator/.

[21] Sayan Mitra and Myla Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theor. Comp. Sci.*, 125(2):45–65, 2005.

[22] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 996 of *LNCS*, pages 101–119. Springer, 1995.

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

[24] Lawrence Paulson. The Isabelle reference manual. Technical Report 283, Univ. of Cambridge, 1993.

[25] Deepak Ramachandran and Eyal Amir. Compact propositional encodings of first-order theories. In *Proc. 20*th *Natl. Conf. on Artif. Intel. and 17*th *Innovative Appl. of Artif. Intel. Conf., July 9-13, 2005, Pittsburgh, PA*, pages 340–345.

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

[27] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Comp. Sci. Lab., SRI Intl., Menlo Park, CA, Nov. 2001.

[28] Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2003.

A Saved TAME proof script for `fischer lemma_1`

To illustrate the utility of the TAME strategies, Figure A.1 shows the verbose version of the saved TAME proof for the `fischer` invariant lemma with the most complex proof, namely, lemma `lemma_1`, which asserts:

```
lemma_1: LEMMA (FORALL (s: states): reachable(s) => Inv_1(s)
```

where the definition of `Inv_1` is (see Figure 4):

```
Inv_1(states):bool = FORALL(k:process): fintime(now(s)) <= last_set(s)(k)
```

The function `fintime` in this saved proof coerces a real number into a time value. The value `k_theorem` is the skolem constant for the universally quantified variable `k` generated by the TAME strategy **auto_induct**. The strategy **auto_induct**, plus the TAME strategies **apply_traj_evolve**, **apply_traj_stop**, and **const_facts**, have been described above. The names of most of the other TAME strategies (e.g., **apply_inv_lemma**) should indicate their purpose. The strategy **inst_in** instantiates a quantifier that is not at the top level of a formula, and the strategy **try_simp** does straightforward reasoning (translates as "it is obvious") in an attempt to complete a proof branch. Anything following a ";;" is a TAME generated comment. Using the (optional) verbose form of the strategies produces the extended comments shown in Figure A.1.

If the body of `Inv_1` did not have a universal quantifier, it would be of the ideal form for applying **deadline_reason** to the value `last_set(s)(k)`. However, as this saved proof shows, the proof can still be handled by using **apply_traj_evolve** and **apply_traj_stop** combined with an appropriate instantiation.

```
fischer_invariants.lemma_1: proved - complete [shostak](6.49 s)

(""
 (auto_induct)
 (("1" ;; Base case
   (const_facts)
   ;; Applying the facts about the constants:
   ;; u_set < l_check AND u_set >= 0 AND l_check >= 0
   (try_simp))
  ("2" ;; Case nu_traj(delta_t_action, F_action)
   (apply_inv_lemma "0" "poststate" "k_theorem")
   ;; Applying the lemma
   ;; FORALL (k: process):
   ;;   pc(poststate)(k) = pc_set =>
   ;;    last_set(poststate)(k) <= fintime(now(poststate) + u_set)
   (apply_specific_precond)
   ;; Applying the precondition
   ;; (FORALL (t: (interval(zero, delta_t_action))):
   ;;    traj_invariant(nu_traj(delta_t_action, F_action))(F_action(t)))
   ;;  AND
   ;;  (FORALL (t: (interval(zero, delta_t_action))):
   ;;     traj_stop(nu_traj(delta_t_action, F_action))(F_action(t)) =>
   ;;      t = delta_t_action)
   ;;   AND
   ;;   (FORALL (t: (interval(zero, delta_t_action))):
   ;;      F_action(t) =
   ;;       traj_evolve(nu_traj(delta_t_action, F_action))(t, prestate))
   (apply_traj_evolve "delta_t_action")
   ;; Using the fact that
   ;;  F_action(delta_t_action) =
   ;;  prestate WITH [now := 1 * dur(delta_t_action) + now(prestate)]
   (suppose "last_set(F_action(zero))(k_theorem) = infinity")
   (("1" ;; Suppose last_set(F_action(zero))(k_theorem) = infinity
     (apply_traj_evolve "zero")
     ;; Using the fact that
     ;;  F_action(zero) = prestate WITH [now := 1 * dur(zero) + now(prestate)]
     (try_simp))
    ("2" ;; Suppose not [last_set(F_action(zero))(k_theorem) = infinity]
     (apply_traj_evolve
      "last_set(prestate)(k_theorem) - fintime(now(prestate))")
     (("1"
       (apply_traj_stop
        "last_set(prestate)(k_theorem) - fintime(now(prestate))")
       ;; Recall the stopping condition
       ;;  (EXISTS (i: process):
       ;;   fintime(now
       ;;            (F_action(last_set(prestate)(k_theorem) -
       ;;                        fintime(now(prestate)))))
       ;;    =
       ;;    last_set
       ;;     (F_action(last_set(prestate)(k_theorem)-fintime(now(prestate))))
       ;;         (i))
       ;;  =>
       ;;  last_set(prestate)(k_theorem)-fintime(now(prestate))
       ;;   = delta_t_action
       (inst_in "stop?" "k_theorem")
       (("1" (try_simp))
        ("2" (try_simp))))
      ("2" ;; Type correctness
       (try_simp))))
    ("3" ;; Type correctness
     (try_simp))))
  ("3" ;; Case test(i_action)
   (const_facts)
   ;; Applying the facts about the constants:
   ;; u_set < l_check AND u_set >= 0 AND l_check >= 0
   (try_simp))
  ("4" ;; Case set(i_action)
   (try_simp))))
```

Fig. A.1. Proof of invariant lemma `lemma_1` of `fischer`