NRL Release Number 06-1226-1940

STRATEGIES 2006
The Sixth International Workshop on Strategies in Automated Deduction
(Seattle, WA, August 16, 2006).

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.

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

[1] This research is supported by AFOSR and ONR.
[2] hongping@csail.mit.edu
[3] archer@itd.nrl.navy.gov

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 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 more natural to the user into a form designed to be convenient for the development of partially or fully automated reasoning support in the prover.

A natural context 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 [20] into the TIOA (Timed Input/Output Automata) [8] 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 instances of continuous state evolution in an automaton analogously to atomic state transitions. Second, as described in [1,2], 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 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 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 specification language and its user-friendliness. 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. Finally, Section 7 discusses 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 [4] 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 [12] considers how to formulate higher order theorems in first order logic, Kerber and Präcklein [13] consider how to best formulate first order logic problems for resolution theorem proving, and Ramachandran and Amir [19] study how to compactly represent certain first order theories in propositional logic. The work in [13] 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 [7], the predecessor of the TIOA language, into the language of different theorem provers, including the Larch Prover [5,9], Isabelle [21,18], and PVS [6]. A previous translator from TIOA (and hence IOA) to PVS is described in [15]. The translator described in this paper, which is derived from the translator in [15], is the first TIOA-to-PVS translator designed especially to support strategy development.

3 Background

The TIOA toolkit.

The TIOA toolkit [8] is designed to support analysis of systems based on the TIOA model framework [10]. 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 [16], an interface to the UPPAAL model-checker [14], 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 [1,2]. The initial version of the translator to PVS was described in [15]. Recent improvements to the translator are the central subject of this paper.

```
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`

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 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.

4 The TIOA specification language

This section provides an overview of the TIOA specification language, using the TIOA description of the Fischer mutual exclusion algorithm in Figure 1 for illustration. A more complete description of the language can be found in The TIOA User Manual and Reference Guide [8].

The TIOA specification language is clear and concise. It allows a user to define an automaton model by providing the minimum necessary information in a natural way. A TIOA specification (see Figure 1) consists of a vocabulary of data types declared using the `vocabulary` keyword, automaton descriptions declared using the `automaton` keyword, and properties of automata declared

```
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)
```

```
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. 2. Invariants of `fischer` in TIOA form and TAME/PVS form

using the keywords `invariant` and `simulation` (see Figure 2).

The main components of an automaton description are the signature, states, transitions, and trajectories, where trajectories can be thought of as "extended transitions" over time. (Usually, trajectories are continuous paths through the state space.) To make use of user defined types, an automaton description can import a vocabulary. Lines 1–4 of Figure 1 shows how the types `process` and `PcValue` are introduced by the vocabulary named `fischer_type`, which is imported by the automaton `fischer` in line 8. The automaton can be parameterized, with a `where` clause constraining the values of the parameters, as illustrated in lines 6–7. The `signature` of an automaton defines the set of `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 type of each variable is specified, together with its initial value. The TIOA language also allows the use of an `initially` clause to further constrain the values of the variables in a start state. No `initially` clause is needed in the specification of `fischer`.

Transitions are specified in a precondition-effect style. The precondition asserts the conditions when the transition can take place, while the effect contains a small program specifying how the state variables are modified by the transition (see lines 20–46).

A trajectory definition (see lines 47–53) consists of an optional `invariant` predicate, a stopping condition specified by the `stop when` clause, and an `evolve` clause stating how the values of the state variables evolve over time.

A state invariant property of an automaton can be specified as an `invariant`. An implementation relationship between a pair of automata [10] can be defined as a `forward simulation` from one to the other. Figure 2 (left column) shows the main state invariants of the automaton `fischer` in TIOA.

5 PVS templates for strategy support

As described in detail in [3], the PVS representations of TIOA specifications produced by the TIOA-to-PVS translator follow a variant of the automaton template used by TAME [1,2] and the TAME property templates, including the forward simulation template described in [17]. 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 [1,2,17].

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` used for the start state predicate, transition function, and precondition predicate in the automaton template; `auto_induct` relies on the standard invariant lemma structure (see Figure 2):

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

and the strategy `prove_fwd_sim` relies on both the (much more complex) 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 the PVS template used for the predicate `start`:

```
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 predicate `start`, 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.

As is explained in more detail in Section 6, trajectories in a TIOA specification are represented as automaton actions with information about their invariant, stopping condition, and evolution captured in their precondition. As with the template for `start`, the 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 step `apply_specific_precond`—which, in an induction proof, introduces into the hypothesis of an induction 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 steps (`apply_traj_invariant` *timeval*), (`apply_traj_stop` *timeval*), and (`apply_traj_evolve` *timeval*) on just its relevant conjunct of the precondition, to define for a given time value, respectively, the value of the trajectory invariant, the value of the trajectory stopping condition, or 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 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 the precondition. A shared universal quantifier would require a shared instantiation of the variable `t`, even in cases where one desires a different instantiation for different parts of the precondition.

The template used for the transition function `trans` also provides a separation of concerns:

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

Representing `trans` using this template allows the values of individual variables in the poststate of a transition to be accessed and reasoned about individually, without having to reason about the values of other variables.

The next section discusses the details of several additional templates, along with the evolution of the TIOA-to-PVS translator towards template support.

6 Translating TIOA specifications into PVS templates

This section provides an overview of the current translation scheme employed by the TIOA to TAME translator, and discusses the alternative translation schemes which have been considered or used previously. We also describe the changes made to the translation scheme to follow the templates mentioned in Section 5, and highlight issues encountered and how they were solved. We refer the reader to [15] for a more complete description of the translator and the translation scheme.

6.1 Overview of translation scheme

As mentioned in Sections 3 and 5, the translation scheme makes use of TAME templates. These templates together with the TAME definition and datatype libraries specify the components of an automaton and provide definitions of TIOA concepts in PVS. The translator instantiates the template with the states, actions and transitions of an input TIOA specification automatically, translating trajectory definitions in TIOA to time passage actions in TAME.

Figure 3 shows the TAME representation of the TIOA description 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?
  try(i: process): try? crit(i: process): crit?
  set(i: process): set? check(i: process): check?
  rem(i: process): rem? reset(i: process): reset?
  exit(i: process): exit?  test(i: process): test?
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)),
  try(i):pc(s)(i)=pc_rem,exit(i):pc(s)(i)=pc_crit,
  test(i):pc(s)(i)=pc_test,set(i):pc(s)(i)=pc_set,
  crit(i):pc(s)(i)=pc_leavetry,
  rem(i):pc(s)(i)=pc_leaveexit,
  check(i): pc(s)(i) = pc_check AND
              first_check(s)(i) <= now(s),
    reset(i): pc(s)(i) = pc_reset
  ENDCASES

trans(a:actions, s:states):states = CASES a OF
 nu_traj(delta_t, F): F(delta_t),
 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]],
 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]]
ENDCASES

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

Fig. 3. TAME representation of `fischer`

`fischer` in Figure 1 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). A trajectory definition in TIOA is translated as a time passage action parameterized by a function `F`, representing the trajectory, and a time interval `delta_t` in PVS (lines 27–28). The time passage action imitates a trajectory by incrementing the values of affected variables as time passes. The function `F` is of type `f_type` which maps a given time interval to a state (lines 23–24). For defining time passage actions, 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 a lemma in PVS stating that the assertion of the invariant holds throughout all reachable states of the automaton. The right column of Figure 2 shows the PVS translation of the invariants of `fischer`.

6.2 Start states

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)
```

A previous translation scheme translates the start state as a conjunction of the equalities equating 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 previous scheme asserts the start state condition using a conjunction of clauses, and uses universal quantifiers to assert the values of the arrays.

In our current translation scheme, we use the 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 assignments into `LAMBDA` expressions in PVS (see lines 18-21 of Figure 3). This is one instance where the form of the TIOA specification was modified to facilitate the desired translation; eventually, this modification can be performed invisibly to the user by a preprocessor. The use of a record equality together with the `LAMBDA` expressions instead of a conjunction of clauses containing universal quantifiers allows simple substitution for the start state `s` in the base case of an invariant proof.

6.3 Trajectory definitions

In an earlier version of the translation scheme, as described in [11], 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 translations of the TIOA expressions for 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 47–52 of Figure 1 using this translation scheme would produce 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 = s WITH [now := now(s) + delta_t],
    . . .
  ENDCASES
```

This translation scheme, however, does not allow assertion of properties that 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.

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 into 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

```
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. 4. TAME translation of multiple trajectory definitions

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 universal quantifier consisting of all the three clauses.

As a refinement of this translation scheme, we add a layer of abstraction by using the definitions `traj_invariant`, `traj_stop` and `traj_evolve`, together with 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 4, in which additional trajectory definitions will simply add more cases to each definition.

6.4 Automaton parameters and `where` *clause*

In a previous version of the 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, because it is by definition true in the start state, and because the values of the automaton parameters are never modified by any transitions. The translation scheme produces the following form, with an additional clause

conjoined to the `start` predicate, and the specifications requires an additional invariant:

```
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)

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);
```

As an attempt to relieve the user from having to manually specify and prove the additional invariant for every automaton, the translation scheme is modified such that the `where` clause is translated as a separate axiom named `const_facts`. This decision also allows the user to invoke the axiom directly with a single TAME proof step (also called `const_facts`), and it also allows strategies to automatically invoke and simplify the axiom when necessary.

6.5 Program statements

The translator currently supports two styles of translation for program statements in the effects of transitions of 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 shows how the effect of the transition `test(i)` would be translated using `LET` statements within the `trans` function:

```
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 of 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 substitution method may yield more complicated expressions. In such cases, translation using the `LET` keyword may produce a simpler translation which corresponds directly to the statements in the original program. However, the use of a sequence of `LET` statements may complicate the proof as additional proof steps will usually be required to simplify the `LET` expression into a form that allows easy reasoning about the updated values of individual variables. Since these additional proof steps for simplification will form part of an application-independent strategy, they are likely to perform more computation than is needed to find the updated values of particular variables. Currently, to move the burden of computation outside of the theorem prover and into the translator, we have been using the substitution method in our examples.

6.6 Type Correctness Conditions

In our current translation scheme, the preconditions and transitions are defined separately in the `enabled` predicate and the `trans` function respectively. A side effect of this separation is that some unprovable Type Correctness Conditions (TCCs) may arise as a result of the translation. As an illustration, consider the following TIOA transition, assuming that `z` is a 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 the translation 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.

One way to resolve this issue is simply to have the translator assert the precondition in a conditional expression in the `trans` function:

```
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. When proving an invariant, the assertion of the precondition will be provided as part of the induction hypothesis, and thus the consequent `THEN` case of the conditional expression will be evaluated as desired with the alternative `ELSE` case ignored.

An alternative approach to handle the TCC is to have the user manually assert the required condition 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 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
```

Since the precondition of a transition may be more complex than the actual expression needed to resolve the TCC (e.g., the precondition in the above case could assert `y /= 0` together with several other constraints), automatically replicating the `enabled` clause in the transition function `trans` could potentially complicate the sequent of a proof with unnecessary formulas. Thus, we currently require the user to adopt the second approach of manually asserting the necessary condition to resolve the TCC. This approach has worked well in the examples with which we have tested the translator. 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

When an invariant of a TIOA specification contains two or more consecutive universal quantifiers, the translator automatically combines the quantified variables into a single `FORALL` expression in the PVS output. For example, the last three invariants of the TIOA specification of `fischer` in the left column of Figure 2 contain the universal quantifiers over `i` and `j` (∀`i:process` ∀`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 the right column of Figure 2. The rationale for this automatic simplification is to allow the user and proof strategies to skolemize such expressions more easily. In particular, combining the quantifiers makes it easier for the strategy `auto_induct` to coordinate the skolemization of the inductive conclusion with the instantiation of the inductive hypothesis in the induction step.

7 Discussion and Conclusions

In this paper we have considered a particular case of the general problem of 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 the anonymous reviewers for their helpful comments.

References

[1] 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.

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

[3] 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.

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

[5] 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, Texas, Houston, USA, November 2002.

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

[7] 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.

[8] Stephen Garland. TIOA User Guide and Reference Manual. Technical report, MIT CSAIL, Cambridge, MA, 2006.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] Hongping Lim. Translating timed I/O automata specifications for theorem proving in PVS. Master's thesis, MIT, Cambridge, MA, 2006.

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

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

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

[19] 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, 2005.

[20] 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.

[21] 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.