Translating Timed I/O Automata Specifications for Theorem Proving in PVS*

Hongping Lim, Dilsun Kaynar, Nancy Lynch, and Sayan Mitra

Massachusetts Institute of Technology,
Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge MA 02139, USA
{hongping,dilsun,lynch,mitras}@csail.mit.edu

Abstract. Timed Input/Output Automaton (TIOA) is a mathematical framework for specification and analysis of systems that involve discrete and continuous evolution. In order to employ an interactive theorem prover in deducing properties of a TIOA, its state-transition based description has to be translated to the language of the theorem prover. In this paper, we describe a tool for translating TIOA to the language of the Prototype Verification System (PVS)—a specification system with an integrated interactive theorem prover. We describe the translation scheme, discuss the design decisions, and briefly present three case studies to illustrate the application of the translator in the verification process.

1 Introduction

Timed Input/Output Automata [1, 2] is a mathematical framework for compositional modeling and analysis of systems that involve discrete and continuous evolution. The state of a timed I/O automaton changes discretely through *actions*, and continuously over time intervals through *trajectories*. A formal language called TIOA [3, 4] has been designed for specifying timed I/O automata. Like in its predecessor IOA [5], in the TIOA language, discrete transitions are specified in the precondition-effect style. In addition, TIOA introduces new constructs for specifying trajectories. Based on the TIOA language, a set of software tools is being developed [3]; these tools include a front-end type checker, a simulator, and an interface to the Prototype Verification System (PVS) theorem prover [6] (see Figure 1). This paper describes the new features of the TIOA language and a tool for translating specifications written in TIOA to the language of PVS; this tool is a part of the third component of the TIOA toolkit.

Motivation. Verification of timed I/O automata properties typically involves proving invariants or simulation relations between pairs of automata. The timed I/O automata framework provides a means for constructing very stylized proofs, which take the form of induction over the length of the executions of an automaton or a pair of automata, and a systematic case analysis of the actions and the

* This research has been supported by Air Force Contract FA9550-04-C-0084.

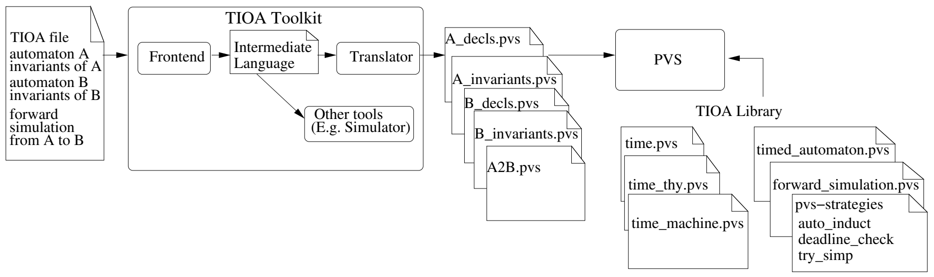

Fig. 1. Theorem proving on TIOA specifications

trajectories. Therefore, it is possible to partially automate such proofs by using an interactive theorem prover, as shown in [7]. Apart of partial automation, theorem prover support is useful for (a) managing large proofs, (b) re-checking proofs after minor changes in the specification, and (c) generating human readable proofs from proof scripts. We have chosen to use the PVS theorem prover because it provides an expressive specification language and an interactive theorem prover with powerful decision procedures. PVS also provides a way of developing special strategies or tactics for partially automating proofs, and it has been used in many real life verification projects [8].

To use a theorem prover like PVS for verification, one has to write the description of the timed I/O automaton model of the system in the language of PVS, which is based on classical, typed higher-order logic. One could write this automaton specification directly in PVS, but using the TIOA language has the following advantages. (a) TIOA preserves the state-transition structure of a timed I/O automaton, (b) allows the user to write programs to describe the transitions using operational semantics, whereas in PVS, transition definitions have to be functions or relations, (c) provides a natural way for describing trajectories using differential equations, and also (d) allows one to use other tools in the TIOA toolkit. Therefore, it is desirable to be able write the description of a timed I/O automaton in the TIOA language, and then use an automated tool to translate this description to the language of PVS.

Related Work and Contributions. Various tools have been developed to translate IOA specifications to different theorem provers, for example, Larch [9, 10], PVS [11], and Isabelle [12, 13]. Our implementation of the TIOA to PVS translator builds upon [9]. However, unlike IOA, TIOA allows the state of a timed I/O automaton to evolve continuously over time through *trajectories*. The main contribution of this paper is the design of a translation scheme from TIOA to PVS that can handle trajectories, and the implementation of the translator.

The Timed Automata Modeling Environment (TAME) [7] provides a PVS theory template for describing MMT automata [14]— a special type of I/O automaton that adds time bounds for enabled actions. This theory template has to be manually instantiated with the states, actions, and transitions of an automaton. A similar template is instantiated automatically by our translator to specify timed I/O automata in PVS. This entails translating the operational descriptions of transitions in TIOA to their corresponding functional descriptions in PVS. Moreover, unlike a timed I/O automaton which uses trajectories, an MMT automaton uses a *time passage action* to model continuous behavior. In TAME, this time passage action is written as another action of the automaton, with the properties of the pre- and post-state expressed in the enabling condition of the action. This approach, however, if applied directly to translate a trajectory, does not allow assertion of properties that must hold throughout the duration of the trajectory. Our translation scheme solves this problem by embedding the trajectory as a functional parameter of the time passage action.

We illustrate the application of the translator in three case studies: Fischer's mutual exclusion algorithm, a two-task race system, and a simple failure detector [15, 2]. The TIOA specifications of the system and its properties are given as input to the translator and the output is a set of PVS theories. The PVS theorem prover is then used to verify the properties using inductive invariant proofs. In two of these case studies, we describe time bounds on the actions of interest using an abstract automaton, and then prove the timing properties by a simulation relation from the system to this abstraction. The simulation relations typically involve inequalities between variables of the system and its abstraction. Our experience with the tool suggests that the process of writing system descriptions in TIOA and then proving system properties using PVS on the translator output can be helpful in verifying more complicated systems.

In the next section we give a brief overview of the timed I/O automata framework and the TIOA language. In Section 3, we describe the translation scheme; in Section 4, we illustrate the application of the translator with brief overviews of three case studies. Finally, we conclude in Section 5.

2 TIOA Mathematical Model and Language

Here we briefly describe the timed I/O automaton model and refer the reader to [1] for a complete description of the mathematical framework.

2.1 TIOA Mathematical Model

Let V be the set of variables of a timed I/O automaton. Each variable $v \in V$ is associated with a *static type*, $type(v)$, which is the set of values v can assume. A *valuation* for V is a function that associates each variable $v \in V$ to a value in $type(v)$. $val(V)$ denotes the set of all valuations of V. Each variable $v \in V$ is also associated with a *dynamic type*, which is the set of trajectories v may follow.

The time domain T is a subgroup of $(R, +)$. A time interval J is a nonempty, left-closed sub-interval of R. J is said to be *closed* if it is also right-closed. A trajectory τ of V is a mapping $\tau : J \rightarrow val(V)$, where J is a time interval starting with 0. The domain of τ, $\tau.dom$, is the interval J. A *point trajectory* is one with the trivial domain $\{0\}$. The first time of τ, $\tau.ftime$, is the infimum of $\tau.dom$. If $\tau.dom$ is closed then τ is *closed* and its limit time, $\tau.ltime$, is the supremum of $\tau.dom$. For any variable $v \in V$, $\tau \downarrow v(t)$ denotes the restriction of τ to the set $val(v)$. Let τ and τ' be trajectories for V, with τ closed. The *concatenation* of τ and τ' is the union of τ and the function obtained by shifting $\tau'.dom$ until $\tau.ltime = \tau'.ftime$. The *suffix* of a trajectory τ is obtained by restricting $\tau.dom$ to $[t, \infty)$, and then shifting the resulting domain by $-t$.

A *timed automaton* $\mathcal{A}$ is a tuple of $(X, Q, \Theta, E, H, D, \mathcal{T})$ where:

1. X is a set of *variables.*
2. $Q \subseteq val(X)$ is a set of *states.*
3. $\Theta \subseteq Q$ is a nonempty set of *start states.*
4. A is a set of actions, partitioned into *external* E and *internal actions* H.
5. $\mathcal{D} \subseteq Q \times A \times Q$ is a set of *discrete transitions.* We write a transition $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$ in short as $\mathbf{x} \overset{a}{\rightarrow} \mathbf{x}'$. We say that a is *enabled* in $\mathbf{x}$ if $\mathbf{x} \overset{a}{\rightarrow} \mathbf{x}'$ for some $\mathbf{x}'$.
6. $\mathcal{T}$ is a set of trajectories for X such that $\tau(t) \in Q$ for every $\tau \in \mathcal{T}$ and every $t \in \tau.dom$, and $\mathcal{T}$ is closed under prefix, suffix and concatenation.

A *timed I/O automaton* is a timed automaton with the set of external actions E partitioned into input and output actions. This distinction is necessary for composing timed I/O automata. In this paper, we consider only individual timed I/O automata and so we do not differentiate input and output actions. We use the terms timed I/O automaton and timed automaton synonymously.

An *execution fragment* of a timed I/O automaton $\mathcal{A}$ is an alternating sequence of actions and trajectories $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$, where $\tau_i \in \mathcal{T}, a_i \in A$, and if τ_i is not the last trajectory in α then τ_i is finite and $\tau_i.lstate \overset{a_{i+1}}{\rightarrow} \tau_{i+1}.fstate$. An execution fragment is *closed* if it is a finite sequence and the domain of the final trajectory is a finite closed interval. An *execution* is an execution fragment whose first state is a start state of $\mathcal{A}$. A state of $\mathcal{A}$ is *reachable* if it is the last state of some execution. An *invariant* property is one which is true in all reachable states of $\mathcal{A}$. A *trace* of an execution fragment α is obtained from α by removing internal actions and modifying the trajectories to contain only information about the amount of elapsed time. $traces_{\mathcal{A}}$ denotes the set of all traces of $\mathcal{A}$. We say that automaton $\mathcal{A}$ *implements* automaton $\mathcal{B}$ if $traces_{\mathcal{A}} \subseteq traces_{\mathcal{B}}$. A *forward simulation relation* [1] from $\mathcal{A}$ to $\mathcal{B}$ is a sufficient condition for showing that $\mathcal{A}$ implements $\mathcal{B}$. A *forward simulation* from automaton $\mathcal{A}$ to $\mathcal{B}$ is a relation $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ satisfying the following conditions for all states $\mathbf{x}_{\mathcal{A}} \in Q_{\mathcal{A}}$ and $\mathbf{x}_{\mathcal{B}} \in Q_{\mathcal{B}}$:

1. If $\mathbf{x}_{\mathcal{A}} \in \Theta_{\mathcal{A}}$ then there exists a state $\mathbf{x}_{\mathcal{B}} \in \Theta_{\mathcal{B}}$ such that $\mathbf{x}_{\mathcal{A}}\ R\ \mathbf{x}_{\mathcal{B}}$.
2. If $\mathbf{x}_{\mathcal{A}}\ R\ \mathbf{x}_{\mathcal{B}}$ and α is a transition $\mathbf{x} \overset{a}{\rightarrow}_{\mathcal{A}} \mathbf{x}'$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{x}_{\mathcal{B}}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate\ R\ \beta.lstate$.
3. If $\mathbf{x}_{\mathcal{A}}\ R\ \mathbf{x}_{\mathcal{B}}$ and α is an execution fragment of $\mathcal{A}$ consisting of a single closed trajectory, with $\alpha.fstate = \mathbf{x}_{\mathcal{A}}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{x}_{\mathcal{B}}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate\ R\ \beta.lstate$.

2.2 TIOA Language

The TIOA language [3] is a formal language for specifying the components and properties of timed I/O automata. The states, actions and transitions of a timed

```
    automaton TwoTaskRace(a1,a2,b1,b2 : Real)
 2  where (a1 > 0) ∧ (a2 > 0) ∧ (b1 ≥ 0) ∧ (b2 > 0) ∧ (a2 ≥ a1) ∧ (b2 ≥ b1)

 4  signature
     internal increment, decrement, set
 6   output report

 8  states
     count: Int := 0, flag: Bool := false,
10   reported: Bool := false, now: Real := 0,
     first_main: Real := a1, last_main: AugmentedReal := a2,
12   first_set:  Real := b1, last_set:  AugmentedReal := b2

14  transitions
     internal increment                        internal decrement
16    pre                                       pre
       ¬flag ∧ now ≥ first_main                 flag ∧ count > 0 ∧ now ≥ first_main
18    eff                                       eff
       count := count + 1;                       count := count - 1;
20     first_main:= (now + a1);                  first_main := (now + a1);
       if (now+a2) ≥ 0 then                     if (now+a2) ≥ 0 then
22       last_main := now+a2                       last_main := (now+a2)
       fi                                        fi
24
     internal set                              output report
26    pre                                       pre
       ¬flag ∧ now ≥ first_set                   flag ∧ count = 0 ∧ ¬reported
28    eff                                         ∧ now ≥ first_main
       flag := true;                            eff
30     first_set := 0;                           reported := true;
       last_set := infty                         first_main := 0;
32                                               last_main := infty
     trajectories
34     trajdef traj1
         invariant now ≥ 0
36       stop when now = last_main ∨ now = last_set
         evolve d(now) = 1
```

Fig. 2. TIOA description of *TwoTaskRace*

I/O automaton are specified in TIOA in the same way as in the IOA language [5]. New features of the TIOA language include trajectories and a new `AugmentedReal` data type. The trajectories are defined using differential and algebraic equations, invariants and stopping conditions. This approach is derived from [16], in which the authors had used differential equations and English informally to describe trajectories. Figure 2 shows an example of a TIOA specification. The `AugmentedReal` type extends reals with a constructor for infinity. Each variable has an explicitly defined static type, and an implicitly defined dynamic type. The dynamic type of a `Real` variable is the set of piecewise-continuous functions; the dynamic type of a variable of any other simple type or of the type **discrete** `Real` is the set of piecewise constant functions.

The set of trajectories of a timed I/O automaton is defined systematically by a set of trajectory definitions. A *trajectory definition* w is defined by an invariant $inv(w)$, a stopping condition $stop(w)$, and a set of differential and algebraic equations $daes(w)$ (see definition of `traj1` in Figure 2, lines 34–37). $W_{\mathcal{A}}$ denotes the set of trajectory definitions of $\mathcal{A}$. Each $w \in W_{\mathcal{A}}$ defines a set of trajectories, denoted by $traj(w)$. A trajectory τ belongs to $traj(w)$ if the following conditions hold: for each $t \in \tau.dom$: (a) $\tau(t) \in inv(w)$. (b) If $\tau(t) \in stop(w)$, then $t = \tau.ltime$. (c) τ satisfies the set of differential and algebraic equations in $daes(w)$. (d) For each non-real variable v, $(\tau \downarrow v)(t) = (\tau \downarrow v)(0)$; that is, the value of v

```
TwoTaskRace_decls: THEORY
BEGIN
  IMPORTING common_decls
  states: TYPE =
         [#count: int, flag: bool,
           reported: bool, now: real,
           first_main: real, last_main: time,
           first_set: real, last_set: time#]

  start(s: states): bool =
    (count(s) = 0 ∧ flag(s) = FALSE ∧
     reported(s) = FALSE ∧ now(s) = 0 ∧
     first_main(s) = a₁ ∧ last_main(s) = fintime(a₂) ∧
     first_set(s) = b₁ ∧ last_set(s) = fintime(b₂))

  interval(i, j: (fintime?)): TYPE =
       {s: (fintime?) | i ≤ s ∧ s ≤ j ∧ i ≤ j}
  f_type(i, j: (fintime?)): TYPE =
       [interval(i, j) → states]

  actions: DATATYPE
   BEGIN
    nu_traj1(delta_t: {t: (fintime?) | dur(t) ≥ 0},
                F: f_type(zero, delta_t)) nu_traj1?
    increment: increment?
    decrement: decrement?
    set: set?
    report: report?
   END actions

  visible(a: actions): bool =
   CASES a OF
     nu_traj1(delta_t, F): TRUE, increment: FALSE,
     decrement: FALSE, set: FALSE, report: TRUE
     ENDCASES

  timepassageactions(a: actions): bool =
   CASES a OF
     nu_traj1(delta_t, F): TRUE, increment: FALSE,
     decrement: FALSE, set: FALSE, report: FALSE
   ENDCASES

 enabled(a: actions, s: states): bool =
  CASES a OF
    nu_traj1(delta_t, F):
      ∀ (t: interval(zero, delta_t)):
        ((now(F(t)) ≥ 0) ∧
          (fintime(now(F(t))) = last_main(F(t)) ∨
           fintime(now(F(t))) = last_set(F(t)))
             ⇒ t = delta_t) ∧
          F(t) = s WITH [now := now(s) + 1 × dur(t)],
    increment: ((¬ flag(s)) ∧ (now(s) ≥ first_main(s))),
    decrement:
      ((flag(s) ∧ (count(s) > 0)) ∧
      (now(s) ≥ first_main(s))),
    set: ((¬ flag(s)) ∧ (now(s) ≥ first_set(s))),
    report:
      (((flag(s) ∧ (count(s) = 0)) ∧
      (¬ reported(s))) ∧ (now(s) ≥ first_main(s)))
    ENDCASES

 trans(a: actions, s: states): states =
  CASES a OF
   nu_traj1(delta_t, F): F(delta_t),
   increment: s WITH
      [first_main := (now(s) + a₁),
       last_main := (IF ((now(s) + a₂) ≥ 0)
                        THEN fintime(now(s) + a₂)
                        ELSE last_main(s) ENDIF),
       count := (count(s) + 1)],
   decrement: s WITH
      [first_main := (now(s) + a₁),
       last_main := (IF ((now(s) + a₂) ≥ 0)
                        THEN fintime(now(s) + a₂)
                        ELSE last_main(s) ENDIF),
      count := (count(s) − 1)],
   set: s WITH
      [last_set := infinity, first_set := 0, flag := TRUE],
   report: s WITH
      [first_main:=0, reported:=TRUE, last_main:=infinity]
  ENDCASES

END TwoTaskRace_decls
```

Fig. 3. PVS description of *TwoTaskRace*

is constant throughout the trajectory. The set of trajectories $\mathcal{T}_\mathcal{A}$ of automaton $\mathcal{A}$ is the concatenation closure of the functions in $\bigcup_{w \in W_\mathcal{A}} traj(w)$.

3 Translation Scheme

For generating PVS theories that specify input TIOA descriptions, our translator implements the approach prescribed in TAME [7]. The translator instantiates a predefined PVS theory *template* that defines the components of a generic automaton. The translator automatically instantiates the template with the states, actions, and transitions of the input TIOA specification. This instantiated theory, together with several supporting library theories, completely specifies the automaton, its transitions, and its reachable states in the language of PVS (see Figure 1). Figure 3 shows the translator output in PVS for the TIOA description in Figure 2. In the following sections, we describe in more detail the translation of the various components of a TIOA description.

3.1 Data Types, Automaton Parameters, and States

Simple static types of the TIOA language `Bool`, `Char`, `Int`, `Nat`, `Real` and `String` have their equivalents in PVS. PVS also supports declaration of TIOA types `enumeration`, `tuple`, `union`, and `array` in its own syntax. The type `AugmentedReal` is translated to the type time introduced in the time theory of TAME. time is defined as a DATATYPE consisting of two subtypes: fintime and infinity. The subtype fintime consists of only non-negative reals; infinity is a constant constructor.

The TIOA language allows the user to introduce new types and operators by declaring the types and the signature of the operators within the TIOA description. The semantics of these types and operators are written in PVS library theories, which are imported by the translator output.

The TIOA language provides the construct **states** for declaring the variables of an automaton (see Figure 2, lines 8–12). Each variable can be assigned an initial value at the start state. An optional **initially** predicate can be used to specify the start states. An automaton can have parameters which can be used in expressions within the description of the automaton (see Figure 2, lines 1–2).

In PVS, the state of an automaton is defined as a record with fields corresponding to the variables of the automaton. A boolean predicate start returns true when a given state satisfies the conditions of a start state (see Figure 3, lines 3–15). Assignments of initial values to variables in the TIOA description are translated as equalities in the start predicate in PVS, while the **initially** predicate is inserted as an additional conjunction into the start predicate. Automaton parameters are declared as constants in a separate PVS theory common_decls (see Figure 3, line 2) with axioms stating the relationship between them.

3.2 Actions and Transitions

In TIOA, actions are declared as **internal** or external (**input** or **output**). In PVS, these are declared as subtypes of an action DATATYPE. A visible predicate returns true for the external and time passage actions.

In TIOA, discrete transitions are specified in precondition-effect style using the keyword **pre** followed by a predicate (precondition), and the keyword **eff** followed by a program (effect). We define a predicate enabled in PVS parameterized on an action a and a state s to represent the preconditions. enabled returns true when the corresponding TIOA precondition for a is satisfied at s.

The program of the effect clause specifies the relation between the post-state and the pre-state of the transition. The program consists of sequential statements, which may be assignments, **if-then-else** conditionals or **for** loops (see Figure 2, lines 14–32). A non-deterministic assignment is handled by adding extra parameters to the action declaration and constraining the values of these parameters in the enabled predicate of the action.

In TIOA, the effect of a transition is typically written in an imperative style using a sequence of statements. We translate each type of statement to its corresponding functional relation between states, as shown in Table 1. The term P is a program, while $trans_P(s)$ is a function that returns the state obtained by

Table 1. Translation of program statements. v is a state variable; t is an expression; *pred* is a predicate; A is a finite set; choose picks an element from the given set A. WITH makes a copy of the record s, assigning the field v with a new value t

program P	$trans_P(s)$
`v := t`	s WITH $[v{:=}\ t]$
if pred **then** P_1 **fi**	IF pred THEN $trans_{P_1}(s)$ ELSE s ENDIF
if pred **then** P_1 **else** P_2 **fi**	IF pred THEN $trans_{P_1}(s)$ ELSE $transP_2(s)$ ENDIF
for `v` **in** `A` **do** P_1 **od**	forloop(A, s): RECURSIVE states = IF empty?(A) THEN s
	ELSE LET v=choose(A), s'=forloop(remove(v, A), s) IN
	$trans_{P1}(s')$ ENDIF MEASURE card(A)

```
signature
  internal foo(i: int), bar
transitions
  internal foo(i: Int)      internal bar
    eff x := x + i;           eff t := x;
        y := x * x;               if x ≠ y then
        x := x - 1;                 x := y;
        y := y + 1                  y := t
                                fi
```

Fig. 4. Actions and transitions in TIOA

performing program P on state s. In PVS, we define a function trans parameterized on an action a and a state s, which returns the post-state of performing the corresponding TIOA effect of a on s. Sequential statements like $P_1; P_2$ are translated to a composition of the corresponding functions $trans_{P_2}(trans_{P_1}(s))$. Our translator can perform this composition in following two ways:

Substitution method: We first compute $trans_{P_1}$, then substitute each variable in $trans_{P_2}$ with its intermediate value obtained from $trans_{P_1}$. This approach explicitly specifies the resulting value of each variable in the post-state in terms of the variables in the pre-state [9]. Figure 4 shows a simple example to illustrate this approach. `foo` performs some arithmetic, while `bar` swaps `x` and `y` if they are not equal. The translation is shown in the left column of Figure 5. In the transition of bar, x and y are assigned new values only when their values are not equal in the pre-state. Otherwise, they are assigned their previous values.

LET method: Instead of performing the substitution explicitly, we make use of the PVS LET keyword to obtain intermediate states on which to apply subsequent programs. The program P_1; P_2 can be written as LET $s = trans_{P_1}(s)$ IN $trans_{P_2}(s)$. The right column in Figure 5 shows the translation of the effects of `foo` and `bar` using LET statements.

In the substitution method, the translator does the work of expressing the final value of a variable in terms of the values of the variables in the pre-state. In the LET method, the prover has to perform these substitutions to obtain an expression for the post-state in an interactive proof. Therefore, the substitution method is more efficient for theorem proving, whereas the LET method preserves the sequential structure of the program, which is lost with the substitution method. Since the style of translation in some cases may be a matter of preference, we currently support both approaches as an option for the user.

```
trans(a: actions, s: states): states =
  CASES a OF
   foo(i): s WITH
        [x := (x(s) + i) − 1,
         y := (x(s) + i) × (x(s) + i) + 1],
   bar: s WITH
        [y := (IF (x(s) ≠ y(s)) THEN x(s)
               ELSE y(s) ENDIF),
         x := (IF (x(s) ≠ y(s)) THEN y(s)
               ELSE x(s) ENDIF),
         t := x(s)]
ENDCASES
```

```
trans(a: actions, s: states): states = CASES a OF
  foo(i):
    (LET s: states = s WITH [x := x(s) + i] IN
    (LET s: states = s WITH [y := x(s) × x(s)] IN
    (LET s: states = s WITH [x := x(s) − 1] IN
    (LET s: states = s WITH [y := y(s) + 1] IN s)))),
  bar:
    (LET s: states = s WITH [t := x(s)] IN
    (LET s: states =
      IF (x(s) ≠ y(s)) THEN
       (LET s: states = s WITH [x := y(s)] IN
       (LET s: states = s WITH [y := t(s)] IN s))
      ELSE s ENDIF
     IN s))
ENDCASES
```

Fig. 5. Translation of transitions using substitution (left) and LET (right)

3.3 Trajectories

The set of trajectories of an automaton is the concatenation closure of the set of trajectories defined by the trajectory definitions of the automaton. A trajectory definition w is specified by the **trajdef** keyword in a TIOA description followed by an **invariant** predicate for $inv(w)$, a **stop when** predicate for $stop(w)$, and an **evolve** clause for specifying $daes(w)$ (see `traj1` in Figure 2, lines 34–37).

Each trajectory definition in TIOA is translated as a time passage action in PVS containing the trajectory map as one of its parameters. The precondition of this time passage action contains the conjunction of the predicates corresponding to the invariant, the stopping condition, and the evolve clause of the trajectory definition. In general, translating an arbitrary set of differential and algebraic equations (DAES) in the evolve clause to the corresponding precondition may be hard, but our translation scheme is designed to handle a large class of DAES, including the most common classes like constant and linear differential equations, and ordinary differential equations. The translator currently handles algebraic equations, constant differential equations and inclusions; it is being extended to handle linear differential equations.

Like other actions, a time passage action is declared as a subtype of the action DATATYPE, and specified using enabled-trans predicates. A time passage action has two required parameters: the length of the time interval of the trajectory, delta_t, and a trajectory map F mapping a time interval to a state of the automaton.

The transition function of the time passage action returns the last state of the trajectory, obtained by applying the trajectory map F on delta_t (see Figure 3, line 63). The precondition of a time passage action has conjunctions stating (1) the trajectory invariant, (2) the stopping condition, and (3) the evolution of variables (see nu_traj1 in Figure 3, lines 44-50, corresponding to `traj1` in Figure 2).

If the **evolve** clause contains a constant differential inclusion of the form $d(x) \leq k$, we introduce an additional parameter x_r in the time passage action for specifying the rate of evolution. We then add a fourth conjunction into the precondition to assert the restriction x_r $\leq k$. The following example uses a constant differential inclusion that allows the rate of change of `x` to be between

0 and 2. The PVS output in Figure 6 contains an additional parameter x_r as the rate of change of x. The value of x_r is constrained in the precondition.

```
trajdef progress invariant x ≥ 0 stop when x = 10
  evolve d(x) ≥ 0; d(x) ≤ 2
```

```
actions: DATATYPE                                           enabled(a: actions, s: states): bool =
 BEGIN                                                        CASES a OF
  nu_progress(delta_t: {t: (fintime?) | dur(t) ≥ 0},           nu_progress(delta_t, F, x_r):
   F: f_type(zero, delta_t), x_r: real) : nu_progress?           x_r ≥ 0 ∧ x_r ≤ 2 ∧
 END actions                                                        (∀ (t: interval(zero, delta_t)):
                                                                        (x(F(t)) ≥ 0) ∧
trans(a: actions, s: states): states =                                 (((x(F(t)) ≤ 10)) ⇒ t = delta_t) ∧
 CASES a OF nu_progress(delta_t, F, x_r):                             F(t) = s WITH
   F(delta_t)                                                              [x := x(s) + x_r × dur(t)])
 ENDCASES                                                     ENDCASES
```

Fig. 6. Using an additional parameter to specify rate of evolution

3.4 Correctness of Translation

Consider a timed I/O automaton $\mathcal{A}$, and its PVS translation $\mathcal{B}$. A closed execution of $\mathcal{B}$ is an alternating finite sequence of states and actions (including time passage actions): $\beta = s_0, b_1, s_1, b_2, \ldots, b_r, s_r$, where s_0 is a start state, and for all i, $0 \leq i \leq r$, s_i is a state of $\mathcal{B}$, and b_i is an action of $\mathcal{B}$. We define the following two mappings:

Let $\beta = s_0, b_1, s_1, b_2, \ldots, b_r, s_r$ be a closed execution of $\mathcal{B}$. We define the mapping $\mathcal{F}(\beta)$ as a sequence $\tau_0, a_1, \tau_1, \ldots$ obtained from β by performing the following: (1) Each state s_i is replaced with a point trajectory τ_j such that $\tau_j.fstate = \tau_j.lstate = s_i$. (2) Each time passage action b_i is replaced by $T(b_i)$, where $T(b_i)$ is the parameter F of b_i, which is the same as the corresponding trajectory in $\mathcal{A}$. (3) Consecutive sequences of trajectories are concatenated into single trajectories.

Let $\alpha = \tau_0, a_1, \tau_1, \ldots$ be a closed execution of $\mathcal{A}$. We define the mapping $\mathcal{G}(\alpha)$ as a sequence $s_0, b_1, s_1, b_2, \ldots, b_r, s_r$ obtained from α by performing the following. Let τ_i be a concatenation of $\tau_{(i,1)}, \tau_{(i,2)}, \ldots$, such that $\tau_{(i,j)} \in traj(w_j)$ for some trajectory definition w_j of $\mathcal{A}$. Replace $\tau_{(i,1)}, \tau_{(i,2)}, \ldots$ with $\tau_{(i,1)}.fstate$, $\nu(\tau_{(i,1)})$, $\tau_{(i,1)}.lstate$, $\nu(\tau_{(i,2)})$, $\tau_{(i,2)}.lstate$, $\ldots$, where $\nu(\tau)$ denotes the corresponding time passage action in $\mathcal{B}$ for τ.

Using these mappings, we state the correctness of our translation scheme as a theorem, in the sense that any closed execution (or trace) of a given timed I/O automaton $\mathcal{A}$ has a corresponding closed execution (resp. trace) of the automaton $\mathcal{B}$, and vice versa, where $\mathcal{B}$ is described by the PVS theories generated by the translator. Owing to limited space, we state the theorem and omit the proof, which will be available in a complete version of the paper.

Theorem 1 *(a) For any closed execution β of $\mathcal{B}$, $\mathcal{F}(\beta)$ is a closed execution of $\mathcal{A}$. (b) For any closed execution α of $\mathcal{A}$, $\mathcal{G}(\alpha)$ is a closed execution of $\mathcal{B}$.*

```
invariant of fischer_me:
  ∀ i: Int ∀ j: Int
    ((i > 0 ∧ j > 0 ∧ i ≠ j) ⇒
      ((pc[i] ≠ pc_crit) ∨
        (pc[j] ≠ pc_crit)))
```

```
Inv(s: states): bool =
  ∀ (i: int): (∀ (j: int):
    (((((i > 0) ∧ (j > 0)) ∧ (i ≠ j)) ⇒
      ((pc(s)(i) ≠ pc_crit) ∨ (pc(s)(j) ≠ pc_crit)))))
lemma: LEMMA (∀ (s: states): reachable(s) ⇒ Inv(s));
```

Fig. 7. TIOA and PVS descriptions of the mutual exclusion property

3.5 Implementation

Written in Java, the translator is a part of the TIOA toolkit (see Figure 1). The implementation of the tool builds upon the existing IOA to Larch translator [9, 17]. Given an input TIOA description, the translator first uses the front-end type checker to parse the input, reporting any errors if necessary. The front-end produces an intermediate language which is also used by other tools in the TIOA toolkit. The translator parses the intermediate language to obtain Java objects representing the TIOA description. Finally, the translator performs the translation as described in this paper, and generates a set of files containing PVS theories specifying the automata and their properties. The translator accepts command line arguments for selecting the translation style for transitions, as well as for specifying additional theories that the output should import for any user defined data types. The current version of the translator can be found at: http://web.mit.edu/hongping/www/tioa/tioa2pvs-translator

4 Proving Properties in PVS

In this section, we briefly discuss our experiences in verifying systems using the PVS theorem prover on the theories generated by our translator. To evaluate the translator we have so far studied the following three systems. We have specifically selected distributed systems with timing requirements so as to test the scalability and generality of our proof techniques. Although these distributed systems are typically specified component-wise, we use a single automaton, obtained by composing the components, as input to the translator for each system.

(1) *Fischer's mutual exclusion algorithm* [15]: In this algorithm, each process proceeds through different phases like `try`, `test`, etc. in order to get to the `critical` phase where it gains access to the shared resource. The *safety property* we want to prove is that no two processes are simultaneously in the `critical` phase, as shown in Figure 7. Each process is indexed by a positive integer; `pc` is an array recording the region each process is in. Notice that we are able to state the invariant using universal quantifiers without having to bound the number of processes.

(2) The *two-task race system* [15, 4] (see Figure 2) increments a variable `count` repeatedly, within `a1` and `a2` time, `a1` < `a2`, until it is interrupted by a `set` action. This `set` action can occur within `b1` and `b2` time from the start, where `b1` ≤ `b2`. After `set`, the value of `count` is decremented (every [`a1`, `a2`]

time) and a `report` action is triggered when `count` reaches 0. We want to show that the time bounds on the occurrence of the `report` action are: *lower bound*: **if** `a2` $<$ `b1` **then** `min(b1,a1)` + $\frac{\texttt{(b1-a2)*a1}}{\texttt{a2}}$ **else** `a1`, and *upper bound*: `b2 + a2 +` $\frac{\texttt{b2*a2}}{\texttt{a1}}$. To prove this, we create an abstract automaton `TwoTaskRaceSpec` which performs a `report` action within these bounds, and show a forward simulation from `TwoTaskRace` to `TwoTaskRaceSpec`.

(3) A simple *failure detector system* [2] consisting of a sender, a delay prone channel, and a receiver. The sender sends messages to the receiver, within `u1` time after the previous message. A `timed_queue` delays the delivery of each message by at most `b`. A failure can occur at any time, after which the sender stops sending. The receiver timeouts after not receiving a message for at least `u2` time. We are interested in proving two properties for this system: (a) *safety*: a timeout occurs only after a failure has occurred, and (b) *timeliness*: a timeout occurs within `u2 + b` time after a failure. As in the two-task race example, to show the time bound, we first create an abstract automaton that timeouts within `u2 + b` time of occurrence of a failure, and then we prove a forward simulation.

We specify the systems and state their properties in the TIOA language. The translator generates separate PVS theory files for the automaton specifications, invariants, and simulation relations (see Figure 1). We invoke the PVS-prover on these theories to interactively prove the translated lemmas and theorems.

One advantage of using a theorem prover like PVS is the ability to develop and use special strategies to partially automate proofs. PVS strategies are written to apply specific proof techniques to recurring patterns found in proofs. In proving the system properties, we use special PVS strategies developed for TAME and TIOA [7, 18]. As many of the properties involve inequalities over real numbers, we also use the strategies in the Manip [19] and the Field [20] packages.

PVS generates Type Correctness Conditions (TCCs), which are proof obligations to show that certain expressions have the right type. As we have defined the `enabled` predicate and `trans` function separately, it is sometimes necessary to add conditional statements into the **eff** program of the TIOA description, so as to ensure type correctness in PVS.

Prior to proving the properties using the translator output, we had proved the same properties using hand-translated versions of the system specifications [4]. These hand-translations were done assuming that all the differential equations are constant, and that the all invariants and stopping conditions are convex. In the proof of invariants, we are able to use a strategy to handle the induction step involving the parameterized trajectory, thus the length of the proofs in the hand translated version were comparable to those with the translators output. However, such a strategy is still not available for use in simulation proofs, and therefore additional proof steps were necessary when proving simulation relations with the translator output, making the proofs longer by 105% in the worst case[1]. Nonetheless, the advantage of our translation scheme is that it is general enough to work for a large class of systems and that it can be implemented in software.

[1] We did not attempt to make the proofs compact.

4.1 Invariant Proofs for Translated Specifications

To prove that a property holds in all reachable states, we use induction to prove that (a) the property holds in the start states, and (b) given that the property holds in any reachable pre-state, the property also holds in the post-state obtained by performing any action that is enabled in the pre-state.

We use the `auto_induct` strategy to inductively prove invariants. This strategy breaks down the proofs into a base case, and one subgoal for each action type. Trivial subgoals are discharged automatically, while other simple branches are proved by using TIOA strategies like `apply_specific_precond` and `try_simp` with decision procedures of PVS. Harder subgoals require more careful user interaction in the form of using simpler invariants and instantiating formulas.

In branches involving time passage actions, to obtain the post-state, we instantiate the universal quantifier over the domain of the trajectory in the time passage action with the limit time of the trajectory. A commonly occurring type of invariant asserts that a continuously evolving variable, say v, does not cross a deadline, say d. Within the trajectory branch of the proof of such an invariant, we instantiate the universal quantifier over the domain of the trajectory with the time required for v to reach the value of d. In particular, if v grows at a constant rate k, we instantiate with $(d-v)/k$. We have also written a PVS strategy `deadline_check` which performs this instantiation.

The strategies provided by Field and Manip deals only with real values, while our inequalities may involve time values. For example, in the two-task race system, we want to show that $\mathsf{last\_set} \geq \mathsf{fintime(now)}$. Here, $\mathsf{last\_set}$ is a time value, that is, a positive real or infinity, while now is a real value. If $\mathsf{last\_set}$ is infinite, the inequality follows from the definitions of $\geq$ and $\mathsf{infinity}$ in the time theory of TAME. For the finite case, we extract the real value from $\mathsf{last\_set}$, and then prove the version of the same inequality involving only reals.

4.2 Simulation Proofs for Translated Specifications

In our examples, we prove a forward simulation relation from the system to the abstract automaton to show that the system satisfies the timing properties. The proof of the simulation relation involves using induction, performing splits on the actions, and verifying the inequalities in the relation. The induction hypothesis assumes that a pre-state $\mathbf{x}_{\mathcal{A}}$ of the system automaton $\mathcal{A}$ is related to a pre-state $\mathbf{x}_{\mathcal{B}}$ of the abstract automaton $\mathcal{B}$. If the action $a_{\mathcal{A}}$ is an external action or a time passage action, we show the existence of a corresponding action $a_{\mathcal{B}}$ in $\mathcal{B}$ such that the $a_{\mathcal{B}}$ is enabled in $\mathbf{x}_{\mathcal{B}}$ and that the post-states obtained by performing $a_{\mathcal{A}}$ on $\mathbf{x}_{\mathcal{A}}$ and $a_{\mathcal{B}}$ on $\mathbf{x}_{\mathcal{B}}$ are related. If the action $a_{\mathcal{A}}$ is internal, we show that the post-state of $a_{\mathcal{A}}$ is related to $\mathbf{x}_{\mathcal{B}}$. To show that two states are related, we prove that the relation holds between the two states using invariants of each automaton, as well as techniques for manipulating inequalities and the time type. We have not used automaton-specific strategies in our current proofs for simulation relations. Such strategies have been developed in [21]. Once tailored to our translation scheme, they will make the proofs shorter.

A time passage action contains the trajectory map as a parameter. When we show the existence of a corresponding action in the abstract automaton, we need to instantiate the time passage action with an appropriate trajectory map. For example, in the proof of the simulation relation in the two-task race system, the time passage action `nu_traj1` of `TwoTaskRace` is simulated by the following time passage action of `TwoTaskRaceSpec`:

```
nu_post_report(delta_t(a_A), LAMBDA(t:TTRSpec_decls.interval(zero,delta_t(a_A))):
    s_B WITH [now := now(s_B)+dur(t)])
```

The time passage action nu_post_report of TwoTaskRaceSpec (abbreviated as TTR-Spec) has two parameters. The first parameter has value equal to the length of a_A, the corresponding time passage action in the automaton TwoTaskRace. The second parameter is a function that maps a given time interval of length t to a state of the abstract automaton. This state is same as the pre-state s_B of TwoTaskRaceSpec, except that the variable now is incremented by t.

5 Conclusion and Future Work

In this paper we have introduced the TIOA language and presented a tool for translating TIOA descriptions to the language of the PVS theorem prover. Although the TIOA language provides convenient and natural constructs for describing a timed I/O automaton, it cannot be used directly in a theorem prover such as PVS. Our tool performs the translation from TIOA to PVS, translating programs in the transition effects of TIOA descriptions into functional relations in PVS, and trajectories into parameterized time passage actions. We have described briefly three case studies in which we have successfully written the systems in TIOA, and proved properties of the systems in PVS using the output of the translator. Our experience suggests that the process of writing system descriptions in TIOA and then proving system properties using PVS on the translator output is useful for analyzing more complicated systems.

Some features remain to be implemented in the translator tool, like **for** loops, and composition of automata. In future, we want to develop PVS strategies to exploit the structure of the translator output for shorter and more readable proofs. We will continue to work on other case studies to evaluate the translator as a theorem proving interface for the TIOA language. These examples include clock synchronization algorithms and implementation of atomic registers.

References

1. Kaynar, D., Lynch, N., Segala, R., Vaandrager, F.: Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In: RTSS 2003: The 24th IEEE International Real-Time Systems Symposium, Cancun, Mexico (2003)
2. Kaynar, D., Lynch, N., Segala, R., Vaandrager, F.: The theory of timed I/O automata. Technical Report MIT/LCS/TR-917, MIT Laboratory for Computer Science (2003) Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

3. Kaynar, D., Lynch, N., Mitra, S., Garland, S.: TIOA Language. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA. (2005)
4. Kaynar, D., Lynch, N., Mitra, S.: Specifying and proving timing properties with tioa tools. In: Work in progress session of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004), Lisbon, Portugal (2004)
5. Garland, S., Lynch, N., Tauber, J., Vaziri, M.: IOA User Guide and Reference Manual. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA. (2003) Available at `http://theory.lcs.mit.edu/tds/ioa.html`.
6. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T.A., eds.: Computer-Aided Verification, CAV '96. Number 1102 in Lecture Notes in Computer Science, New Brunswick, NJ, Springer-Verlag (1996) 411–414
7. Archer, M.: TAME: PVS Strategies for special purpose theorem proving. Annals of Mathematics and Artificial Intelligence **29** (2001)
8. Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: PVS: an experience report. In Hutter, D., Stephan, W., Traverso, P., Ullman, M., eds.: Applied Formal Methods—FM-Trends 98. Volume 1641 of Lecture Notes in Computer Science., Boppard, Germany, Springer-Verlag (1998) 338–345
9. Bogdanov, A., Garland, S., Lynch, N.: Mechanical translation of I/O automaton specifications into first-order logic. In: Formal Techniques for Networked and Distributed Sytems - FORTE 2002 : 22nd IFIP WG 6.1 International Conference, Texas, Houston, USA (2002) 364–368
10. Garland, S.J., Guttag, J.V.: A guide to LP, the Larch prover. Technical report, DEC Systems Research Center (1991) Available at `http://nms.lcs.mit.edu/Larch/LP`.
11. Devillers, M.: Translating IOA automata to PVS. Technical Report CSI-R9903, Computing Science Institute, University of Nijmegen (1999) Available at `http://www.cs.ru.nl/research/reports/info/CSI-R9903.html`.
12. Ne Win, T.: Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA (2003)
13. Paulson, L.C.: The Isabelle reference manual. Technical Report 283, University of Cambridge (1993)
14. Merritt, Modugno, Tuttle: Time-constrained automata. In: CONCUR: 2nd International Conference on Concurrency Theory, LNCS, Springer-Verlag (1991)
15. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
16. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Information and Computation **185** (2003) 105–157
17. Bogdanov, A.: Formal verification of simulations between I/O automata. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA (2000) Available at `http://theory.lcs.mit.edu/tds/ioa.html`.
18. Mitra, S., Archer, M.: Reusable PVS proof strategies for proving abstraction properties of I/O automata. In: STRATEGIES 2004, IJCAR Workshop on strategies in automated deduction, Cork, Ireland (2004)
19. Muñoz, C., Mayero, M.: Real automation in the field. Technical Report NASA/CR-2001-211271 Interim ICASE Report No. 39, ICASE-NASA Langley, ICASE Mail Stop 132C, NASA Langley Research Center, Hampton VA 23681-2199, USA (2001)
20. Vito, B.: A PVS prover strategy package for common manipulations (2003) Available at `http://shemesh.larc.nasa.gov/people/bld/manip.html`.
21. Mitra, S., Archer, M.: PVS strategies for proving abstraction properties of automata. Electronic Notes in Theoretical Computer Science **125** (2005) 45–65