Specifying and Proving Properties of Timed I/O Automata in the TIOA Toolkit *

Myla Archer
Center for High Assurance Computer Systems, Code 5546
Naval Research Laboratory, Washington, DC 20375 USA
archer@itd.nrl.navy.mil

HongPing Lim Nancy Lynch Sayan Mitra Shinya Umeno
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139 USA hongping,lynch,mitras,umeno@csail.mit.edu

Abstract

Timed I/O Automata (TIOA) is a mathematical framework for modeling and verification of distributed systems that involve discrete and continuous dynamics. TIOA can be used for example, to model a real-time software component controlling a physical process. The TIOA model is sufficiently general to subsume other models in use for timed systems. The TIOA toolkit, currently under development, is aimed at supporting system development based on TIOA specifications. The TIOA toolkit is an extension of the IOA toolkit, which provides a specification simulator, a code generator, and both model checking and theorem proving support for analyzing specifications. This paper focuses on modeling of timed systems with TIOA and the TAME-based theorem proving support provided in the toolkit for proving system properties, including timing properties. Several examples are provided by way of illustration.

1 Introduction

To achieve high assurance in the development of complex systems, an appropriate development framework supporting system specification, implementation, and analysis is essential. The support provided by the framework should apply not only to those systems that can be modeled as finite state machines but to those that cannot, such as many real-time embedded or hybrid systems involving software and/or continuous behavior. Thus an ideal general development framework should provide:

1. A mathematical model capable of capturing the range of discrete and continuous phenomena that arise in typical systems,
2. A well defined notion in the model of external (visible) behavior, and a definition of implementation of one component by another, or equivalence of two components, in terms of their visible behavior,
3. Compositionality—i.e, the ability to build larger systems by composing smaller components in a manner that respects the notion of implementation,
4. User-friendly tool support for proving the commonly encountered types of properties for the models, such as invariant properties, implementation relations, and stability, and
5. A basis supporting the use of automatic analysis and other software tools to the extent possible.

The Timed Input/Output Automaton (TIOA) toolkit [15, 9], currently under development, provides just such a framework. The TIOA toolkit, based on the TIOA model [16], is especially suited to the specification and analysis of real-time, embedded systems.

The focus of this paper is on the theorem proving support provided in the TIOA toolkit for the analysis of TIOA specifications. With a set of small examples, we illustrate how one can use the toolkit to model timed systems and specify their properties in the TIOA language, and then verify the specified properties using the theorem prover PVS [28] through the interface TAME [3].

The paper is organized as follows. Section 2 gives an overview of the Timed I/O Automaton (TIOA) model and the TIOA toolkit that supports its use. Section 3 describes how one can specify and prove properties of TIOA models and how the TIOA toolkit supports verifying (or proof checking) the properties mechanically in PVS. Section 4 presents our example TIOA specifications of automata and their properties, and shows how the properties
TIOA file
automaton A
invariants of A
forward simulation
from A to B

TIOA Toolkit
Frontend
Abstract Syntax Tree
Translator
Other tools
(E.g. Simulator,
Model Checker)

A_decls.pvs
A_invariants.pvs
B_decls.pvs
B_invariants.pvs
A2B.pvs

PVS

time.pvs
time_machine.pvs
timed_automaton.pvs
time_thy.pvs

time_pvs
pvs−strategies
auto_induct
deadline_reason
try_simp

TAME Library for TIOA

Figure 1. TIOA framework for theorem-proving

can be proved in a “natural”, high-level fashion in PVS using
the toolkit’s TAME support. Finally, Section 5 discusses
some lessons learned from these and other examples, Sec-

tion 6 mentions some related work, and Section 7 describes
our future plans and presents some conclusions.

2 Background

2.1 The TIOA model

The TIOA model is a timed version of the I/O automaton
model described in [22]. In the I/O automaton model,
states are represented by an assignment of values to state
variables, and state transitions are the result of actions. Ac-
tions may have parameters, and their transitions are defined
in terms of preconditions and effects. Actions are classified
as external (i.e., input or output) or internal. I/O automata
can be composed through shared actions: an output action
of one automaton can be combined with compatible input
actions of one or more other I/O automata.

Timing can be added to I/O automata by various means:
see, for example [25, 23]. In the TIOA model, time pas-
sage is modeled using trajectories, which represent paths
through the state space that are followed during the pas-
sage of time. A trajectory is specified by 1) a description
of its evolution over time, which may be nondeterministic,
given, e.g., in terms of algebraic or differential equations or
inequalities; 2) an (optional) stopping condition that, when
it becomes true, ends the trajectory; and 3) an (optional)
state invariant that must hold throughout the trajectory. The
TIOA model is sufficiently general to subsume most other
commonly used models for timed automata (e.g., [2, 1]). A
detailed description of the theory of TIOA and its compari-
son with other models can be found in [16].

2.2 The TIOA toolkit

The TIOA toolkit [9], currently under development, is a
formal framework for system development based on spec-
ifications in the TIOA language. The TIOA language con-
structs related to timing are discussed in Section 4; example
TIOA specifications can be found in Figures 2, 7, and 8. As
an extension of the IOA toolkit [11], the TIOA toolkit pro-
vides a specification simulator, a code generator, and both
model checking and theorem proving support for analyzing
specifications. For model checking an appropriately re-
stricted class of timed systems in TIOA, an interface to UP-
PAAL [17] is being developed.

The TIOA framework for theorem proving in [15], pro-
vides an approach for writing a system specification in the
TIOA language, translating the TIOA description into the
language of PVS, and then using PVS to verify properties
of the system. The framework makes use of a PVS theory
template which is instantiated with the states, actions and
transitions of an automaton. To perform this translation and
instantiation automatically, a translator tool has been devel-
oped [20, 18] as part of the TIOA toolkit. The PVS theory
template used in the TIOA toolkit is a variant of the TAME
(Timed Automata Modeling Environment) [3, 5] automa-
ton template, whose original variants supported modeling
and proving properties of MMT automata [25] and SCR au-
tomata [14]. An important part of the design of TAME proof
support for any particular automaton model is the design
of the PVS theory template which representations of model
instances will follow. The design of the TAME TIOA tem-
plate is especially aimed at supporting TAME proof steps
(which are implemented as PVS strategies) for reasoning
about trajectories. Table 1 describes the new TAME strat-
egies for reasoning about trajectories. Example proofs using
apply_traj_evolve and deadline_reason can be seen in Section 4.1, Figure 6 and Section 4.2, and Figure 13
respectively.

<table>
<thead>
<tr>
<th>TAME proof step</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>(apply_traj_evolve t)</td>
<td>Compute state time t from now</td>
</tr>
<tr>
<td>(apply_traj_stop t)</td>
<td>Deduce that the stopping condition cannot hold after time t in a trajectory T unless T ends at t</td>
</tr>
<tr>
<td>(apply_traj_invariant t)</td>
<td>Deduce trajectory invariant holds time t from now</td>
</tr>
<tr>
<td>(deadline_reason t)</td>
<td>Deduce trajectory cannot evolve more than time t if a deadline is reached time t from now</td>
</tr>
</tbody>
</table>

Table 1. New TAME strategies for trajectories.
3 Overview of the TIOA proof methodology

The TIOA mathematical model is useful for specifying timed distributed systems and analyzing properties of the systems as invariants and simulation relations. The model also provides a means of organizing proofs of such properties by induction over the length of the execution of an automaton into a systematic case analysis with respect to the actions and trajectories. It is therefore possible to develop PVS strategies to partially automate such proofs.

The TIOA methodology for theorem proving involves (1) writing the specification of a system and its properties in the TIOA language, (2) using the translator tool to generate the PVS equivalent of the system, and then (3) proving the properties in PVS using TAME strategies (see Figure 1). The user describes the system in the TIOA language using the state-transition structure. The user writes simple program statements to describe transitions, and specifies trajectories using differential equations. Once the TIOA description is type checked by the front end of the toolkit, the translator generates a set of PVS files. Together with the TAME library containing PVS definitions for timed I/O automata and any additional data type theories, these generated files specify the automaton and its properties. The user then uses TAME strategies developed for TIOA to prove the properties of the system in PVS.

By using this approach, the user avoids having to write the automaton description directly in PVS. Moreover, the translator also performs the task of translating program statements in TIOA into functional relations in PVS, and trajectories with differential equations into time-passage actions. An additional benefit gained from using the approach is that the user can also use other tools in the toolkit including the simulator, code generator and model checker.

4 Examples

This section provides three simple examples that together illustrate how TIOA is used to represent systems and properties, how trajectories can be used to capture desired timing behavior, and how system properties can be mechanically verified using PVS. The first example, fischer, is a timed version of Fischer’s mutual exclusion algorithm. We use this example to illustrate in some detail how various features of a TIOA specification, in particular, its trajectories, are represented in PVS. We also illustrate how its main correctness property, an invariant, can be proved using TAME. The second example, TwoTaskRace (representing, as its name suggests, a two task race), is used as an example in which the main correctness property is an abstraction property (forward simulation). The last example, timeout, representing a simple timeout system, is used to illustrate the support provided for expressing and reasoning about complex data types in the TIOA toolkit.

4.1 Fischer’s mutual exclusion algorithm

Fischer’s mutual exclusion algorithm solves the mutual exclusion problem in which multiple processes compete for a shared resource. Figure 2 shows the TIOA specification of a timed version of the Fischer algorithm.

In the Fischer algorithm, each process proceeds through different phases in order to get to the critical phase where it gains access to the shared resource. In the automaton used to model the algorithm, each phase has a corresponding action; timing is modeled in the algorithm by time bounds on the actions. The interesting action cases are test, set, and check. The action set has an upper time bound, u_set, while the action check has a lower time bound l_check, and u_set < l_check. When a process enters the test phase, it tests whether the value of a shared variable x has been set by any process; if not, the process can proceed to the next phase, set, within the upper time bound, u_set. In the set phase, the process sets a shared variable x to its index. Thereafter, the process can proceed to the next phase check only after l_set amount of time has elapsed. In the check phase, the process checks to see if x contains the index of the process. If so, it proceeds to the critical phase.

The safety property we want to prove is that no two processes are simultaneously in the critical phase. We also prove simpler invariants to help us prove this main invariant. Figure 3 shows all the invariants that we have proved, the last invariant being the safety property.

To illustrate how the various elements of an automaton specification in TIOA translate into TAME, Figure 4 shows the TAME specification output by the TIOA-to-TAME translator applied to the TIOA specification in Figure 2. The TAME specification has been edited slightly to save space. In the TAME specification, automaton parameters are translated as constants, and the where clause constraining the parameters is expressed as an axiom named const_facts. The state variables are represented as a record type named states. A start predicate is defined to be true for states with the specified initial values. The actions of the automaton are declared as a subset of the actions data type in the TAME specification. A predicate enabled captures the precondition for each action, while a transition function trans captures the post-state obtained by applying the transition of an action on a given pre-state. In translating the effect of an action into the transition function, the translator performs explicit substitutions in accordance with the program statements in the specification of the effect of the action in TIOA, in order to express each state variable in the post-state explicitly in terms of the variables in the pre-state.

The trajectory definition traj in the TIOA specification is translated as a time passage action nu_traj in the TAME specification which has two parameters:
Figure 2. TIOA specification for fischer.

\[
\text{fischer_decl} : \text{THEORY BEGIN} \\
\text{  l_check: real; u_set: real} \\
\text{  const_facts: AXIOM U_set < l_check AND u_set >= 0 AND l_check >= 0} \\
\text{  states: TYPE = \{#} \\
\text{  turn: lift[process], now: real, } \\
\text{  pc: Array[process, PcValue] := constant(pc_rem), } \\
\text{  last_set: Array[process, AugmentedReal] := constant(u_set), } \\
\text{  first_check: Array[process, Real] := constant(0)} \\
\text{END fischer_decls} \\
\text{IMPORTING timed_auto_lib@time_machine} \\
\text{ Figure 4. TAME representation of fischer.}
\]

\[
\text{fischer_invariant : THEORY BEGIN} \\
\text{  \forall k: process \{pc[k] = pc_set \Rightarrow (last_set[k] \leq (now + u_set))\}} \\
\text{  \forall k: process \{pc[k] = pc_set \Rightarrow (last_set[k] \neq \infty)\}} \\
\text{  \forall k: process \{(pc[k] = pc_set \Rightarrow \text{last_set[k] \leq (now + u_set))\}} \\
\text{  \forall i: process \text{\{\text{pc[i] = pc_set \Rightarrow last_set[i] \leq (now + u_set)\}}} \\
\text{  \forall i: process \{(pc[i] = pc_set \Rightarrow first_check[i] \leq last_set[i]\}}} \\
\text{  \forall i: process \{(pc[i] = pc_set \Rightarrow first_check[i] > last_set[i]\}}} \\
\text{  \forall i: process \text{\{(pc[i] = pc_set \Rightarrow first_check[i] > last_set[i]\}}} \\
\text{END fischer_invariants} \\
\text{Figure 3. TIOA invariants for fischer.}
\]
Figure 5. TAME 

**lemma_5** for fischer

```plaintext

lemma_5: LEMMA FORALL (s:states): reachable(s) => Inv_5(s);

;; Proof lemma_5-1 for formula fischer_invariants.lemma_5
;; developed with shostak decision procedures
;;
;; (auto_induct)
;;
;; (FORALL (t: (interval(zero, delta_t_action))):
;; traj_invariant(nu_traj(delta_t_action, F_action))
;; (successor(t))
;; (F_action(t))
;;
;; AND
;; (FORALL (t: (interval(zero, delta_t_action))):
;; traj_stop(nu_traj(delta_t_action, F_action))
;; (F_action(t))
;;
;; AND
;; (FORALL (t: (interval(zero, delta_t_action))):
;; traj_evaluate(nu_traj(delta_t_action, F_action))
;;
;; (t, prestate)
;; (apply_traj_evolve "delta_t_action")
;;
;; Using the fact that
;;
;; traj_evaluate(delta_t_action) =
;; prestate with
;;
;; [now := 1 * dur(delta_t_action) + now(prestate)]
;;
;; (try_simp)
;;
;; ;; Case crit(i_action)
;; (apply_specific_precond)
;;
;; Applying the precondition
;;
;; (FORALL (i: process, j: process):
;; pc(prestate)(i) = pc_leavety
;; (apply_inv_lemma "i_theorem")
;;
;; Applying the lemma
;;
;; (FORALL (i: process, j: process):
;; pc(prestate)(i) = pc_leavety OR
;; pc(prestate)(j) = pc_leavety OR
;;
;; (try_simp)
;;
;; ;; Case crit(i_action)
;; (apply_specific_precond)
;;
;; Applying the precondition
;;
;; (FORALL (i: process, j: process):
;; pc(prestate)(i) = pc_leavety OR
;; pc(prestate)(j) = pc_leavety OR
;;
;; (try_simp)
```

Figure 6. TAME proof of lemma_5 in fischer

delta_t\_action, the duration of the trajectory, and F, a function representing the trajectory, which maps time values to states. The definitions traj\_invariant, traj\_stop, and traj\_evaluate capture the invariant, stopping condition and evolve clause of the trajectory definition respectively. The effect of the “trajectory action” nu\_tra j is constrained—and thus, effectively, captured—by the precondition of nu\_tra j, which asserts that (1) the invariant holds throughout the duration of the trajectory, (2) the stopping condition holds only in the last state of the trajectory, and (3) the evolution of the state variables satisfies the evolve clause. The transition function for nu\_tra j simply returns the post-state obtained by applying the trajectory function F after an elapsed time of delta_t\_action. This method of representation, adapted from a technique of Luchangco [21], allows trans to be represented as a function from states and actions to states while allowing the result of a nu\_tra j “action” to be nondeterministic.

The new TAME strategies in Table 1, combined with the existing TAME strategies, provide a set of proof steps that allow the fischer invariants shown in Figure 3 to be proved interactively in PVS in a clear, high-level fashion. The TIOA-to-TAME translator transforms the six invariants in Figure 3 into TAME invariants and lemmas numbered starting from 0. Thus, the goal safety property, the last invariant in Figure 3, becomes the TAME invariant/lemma pair shown in Figure 5.

Figure 6 shows a verbose TAME proof of lemma_5 in Figure 5. To create this proof, which can be rerun in PVS, the user simply types in the eight TAME proof steps in the proof script—(auto_induct), (apply_specific_precond), and so on. The comments in this proof (which appear as text after semicolons) are generated by the TAME strategies, and serve to label the proof branches and document the facts introduced by the proof steps in these branches. Because TAME automatically handles “trivial” cases, only the proof steps requiring human guidance need to be recorded. This proof can be understood as follows: The proof step auto_induct automates as far as possible the standard initial steps of a proof by induction on the reachable states, including skolemization. The values with names ending in “_theorem” or “_action” are skolem constants standing for variables in the lemma and parameters in the current action, respectively. The name prestate refers to the prestate of the current action, and the values of state variables in any state s are represented as functions of s. The base case and all the action cases except nu\_tra j(delta_t_action, F_action) and crit(i_action) are trivial. The nu\_tra j(delta_t_action, F_action) case is proved by recalling the full precondition with apply_specific_precond, and then using the new TAME step traj\_evaluate in Table 1 to compute what the current state will be after time delta_t\_action. Once this is done, only “obvious” reasoning is needed, which is performed by try_simp. The proof in the crit(i_action) case first recalls the precondition and then uses apply_inv_lemma to apply two earlier invariant lemmas to appropriate instances of their quantified variables. Then, only “obvious” reasoning with try_simp is needed to complete the proof.

### 4.2 A two task race

The two-task race system (see Figure 7 for its TIOA description) 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 between 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 \(a_2 < b_1\) then \(\min(b_1, a_1) + \frac{(b_1 - a_2) * a_1}{a_2}\) else \(a_1\), and upper bound: \(b_2 + a_2 + \frac{b_2 * a_2}{a_1}\). This property is proved by specifying an abstract automaton TwoTaskRaceSpec which performs a report action within these bounds (see Figure 8) and defining a forward simulation relation from TwoTaskRace to TwoTaskRaceSpec (see Figure 10).

The abstract automaton TwoTaskRaceSpec has two trajectories: pre_report and post_report. The TAME representation of TwoTaskRaceSpec (see Figure 9) illustrates how the translator represents multiple trajectories in TAME: the preconditions in enabled and postconditions in trans are expressed identically, while the details of the trajectories are captured in separate cases in traj_invariant, traj_stop, and traj_evolve.

The TIOA-to-TAME translator transforms the TIOA specification in Figure 10 of the forward simulation relation into the PVS theory in Figure 11 that asserts (as a theorem to be proved) the property forward_simulation. The theory in Figure 11 follows the TAME template
forward simulation from TwoTaskRace to TwoTaskRaceSpec:

∀ a1: Real ∀ a2: Real ∀ b1: Real ∀ b2: Real
∀ last_set: Real ∀ last_main: Real ∀ last_report: Real
(a1 > 0 ∧ a2 > 0 ∧ b1 ≥ 0 ∧ b2 > 0 ∧ a1 > a2 ∧ b1 > b2 ∧ b1 > b2) ⇒
∧ last_set > 0 ∧ last_main - TwoTaskRace.last_set
∧ last_main ≠ TwoTaskRace.last_main
∧ last_report > TwoTaskRace.last_report
⇒ TwoTaskRace.reported = TwoTaskRaceSpec.reported
∧ TwoTaskRace.now = TwoTaskRaceSpec.now
∧ (TwoTaskRace.flag ∧ last_main > TwoTaskRace.first_main ∧ last_report ≥ 0 ∧ last_main < TwoTaskRace.first_set) ⇒
TwoTaskRaceSpec.first_report ≤ (min(TwoTaskRace.first_set, TwoTaskRace.first_main) + (TwoTaskRace.first_count + (TwoTaskRace.first_report - last_report) ≤ (TwoTaskRace.count + a2[1] * a1[1])) ∧
(TwoTaskRace.flag ∧ last_main ≥ TwoTaskRace.first_set ∧ last_report > [last_set + (TwoTaskRace.count + 2 + ((last_set - TwoTaskRace.first_main) / a1[1] * a2[1])]) ∧
(TwoTaskRace.flag ∧ last_main ≤ TwoTaskRace.first_set ⇒ last_report > [last_set + (TwoTaskRace.count + a2[1])])

Figure 10. Forward simulation from TwoTaskRace to TwoTaskRaceSpec

TwoTaskRace2TwoTaskRaceSpec: THEORY BEGIN
IMPORTING TwoTaskRaceSpec.invariants
IMPORTING TwoTaskRaceSpec_invariants
timed_auto_lib: LIBRARY = ../timed_auto_lib
MA: THEORY = timed_auto_lib@timed_automaton
IMPORTING TwoTaskRace.decis
MB: THEORY = timed_auto_lib@timed_automaton
import TwoTaskRaceSpec.decls

Figure 11. Simulation relation in TAME

invariant of TwoTaskRace:
(a1 > 0 ∧ a2 > 0 ∧ b1 > 0 ∧ b2 > 0 ∧ a1 > a2 ∧ b1 > b2) ⇒
invariant of TwoTaskRace: now > 0

Figure 12. TwoTaskRace invariants 0–4.

for formulating abstraction relations between automata described in [26]. The theory forward simulation imported in Figure 11 just before the statement of the theorem provides the generic definition in PVS of the property forward_simulation stating what it means for a relation between two automata to be a forward simulation. The PVS formulation of the forward simulation property is

Figure 13. Proof of TwoTaskRace invariant 4.

based on the definition in [24]. The proof of this property for TwoTaskRace and TwoTaskRaceSpec uses invariants of both automata.

The invariants of TwoTaskRace and TwoTaskRaceSpec needed for the forward simulation proof have all been proved in TAME. The proofs of these invariants are all quite simple; in fact, all of the invariants needed for TwoTaskRaceSpec are proved automatically by the TAME induction strategy auto_induct. The proofs of a few of the invariants for TwoTaskRace are interesting because they illustrate the use of the new TAME strategy deadline_reason, which was not used in the invariant proofs for fischer. One such invariant is invariant 4 in Figure 12, whose TAME proof is shown in Figure 13. Invariant 4 essentially says that in the TIOA model of TwoTaskRace, the current time now cannot pass beyond the deadline last_main. In this proof, auto_induct has determined that the base case and four of the five possible action cases are nontrivial. The crux of this proof
is the reasoning in the single time passage case, namely, the action case nu\_traj(delta\_action). After using apply\_specific\_precond and apply\_traj\_e\_\_volve to compute the state after time delta\_action and using apply\_INV\_lemma to use invariant 1 to establish that now \(\geq 0\) at the beginning of the trajectory, the new TAME step deadline\_reason argues that now \(\leq \text{last}_\text{main}\) at the end of the trajectory. The step try\_simp then completes the proof with “obvious reasoning”. The remaining cases are easily proved using “obvious reasoning” following, in some cases, the use of \texttt{const\_facts} to introduce facts about the constants in the specification.

TAME also provides strategies for establishing abstraction relations between automata, including forward simulation. Forward simulation proofs have a high-level structure similar to the structure of induction proofs of invariants; however, rather than beginning with the proof step auto\_induct, they begin with the proof step prove\_fwd\_sim. For more details, see [26].

4.3 A simple timeout system

A simple timeout system consists of a sender, a delay prone channel, and a receiver (see Figure 14 for its TIOA description). The sender sends messages to the receiver, within \(u1\) time after the previous message has been sent. A timed\_message\_Queue delays the delivery of each message by at most \(b\) time. A failure can occur at any time, after which the sender stops sending. The receiver times out after not receiving a message for at least \(u2\) time.

We are interested in proving the following properties for this system: (1) Safety: A timeout occurs only after a failure has occurred; (2) Timeliness: A timeout occurs within \(u2 + b\) time after a failure. The safety property can be captured by an invariant of the system. As in the two-task race example, to show the timeliness, we first create an abstract automaton that times out within \(u2 + b\) time of occurrence of a failure, and then we prove a forward simulation from the system to its abstraction. Both the safety and timeliness properties have been proved using the TAME strategies in a manner analogous to the invariant and forward simulation proofs.

```
automaton timeout(u1, u2, b: Real)
  where u1 \geq 0 \& u2 \geq 0 \& b \geq 0 \& u2 > (u1 + b)
  imports timed_queue
  signature
    internal send(m: M)
    internal receive(m: M)
    output fail
    output timeout
  states
    p_clock: AugmentedReal := 0,
    t_clock: AugmentedReal := u2,
    suspected: Bool := false,
    failed: Bool := false,
    now: Real := 0,
    queue: timed_message_Queue := mtQ
  transitions
    internal send(m)
    pre now \geq 0 \& \neg failed \& p_clock = now
    eff if (now + u1) \geq 0 then p_clock := now + u1 fi;
    if (now + b) \geq latest\_deadline(queue) then
      queue := enQ(MK\_timed\_message\_Queue(n, now + b), queue)
      fi;
    internal receive(m)
    pre now \geq 0 \& enQ\_qn(queue) \& m = earliest\_msg\_Queue(queue)
    eff if (now + u2) \geq 0 then t_clock := now + u2 fi;
    if enQ\_qn(queue) then queue := deQ(queue) fi
    output fail
    pre failed
    eff failed := true;
    p_clock := \text{linfty}
    output timeout
    pre now \geq 0 \& \neg suspected \& t_clock = now
    eff suspected := true;
    t_clock := \text{linfty}
  trajectories
    trajdef tral
    stop when now \geq 0 \& (now = p_clock \lor now = t_clock
    \lor now = earliest\_deadline(queue))
    evolve d(now) = 1
```

Figure 14. TIOA description of timeout.

After the timeout occurs, the clock eventually reaches the \(\text{latest}_\text{deadline}\) and fails. The safety property can be argued with the obvious reasoning following, in some cases, the use of \texttt{const\_facts} to introduce facts about the constants in the specification.
ward simulation proofs in the previous examples, with one extra complication: the need to introduce knowledge about special data types referred to in the TIOA specifications.

The timeout system makes use of a custom data type timed\_message\_queue. TIOA provides a vocabulary syntax to allow the user to declare custom data types and operators. Figure 15 shows how the data type for timed\_message\_queue and the associated operators are declared in TIOA. The actual PVS definitions of these types and operators are provided as part of a TIOA library of data type theories; Figure 16 shows a sample of these definitions. Aside from the PVS operator enQ? (which implements the TIOA operator enQ\_qt for querying whether a timed\_message\_queue is a nonempty queue), the PVS vocabulary is identical to the TIOA vocabulary. Properties of these data types have been proved in PVS, and have been used in proofs of the specification properties.

5 Discussion

Developing theorem proving support. Our approach to developing appropriate theorem proving support for TIOA is to study many examples of TIOA specifications and their properties and identify what is needed for implementing a standard, straightforward set of proof steps sufficient to mechanize proofs of the properties. One lesson we have learned is that the details of the specification template that a translator to PVS targets, if chosen carefully, can greatly facilitate the implementation of PVS strategies. Details of the TAME template for TIOA that have proved helpful for strategy development include the overall scheme for representing trajectories illustrated in Figure 9 and the scheme for representing the start state predicate start(s) as an equality of the form s = ... , possibly in conjunction with additional restrictions (see, for example, Figure 4). Another detail of our translation scheme is the use of symbolic computation, if necessary, to permit the effects of transitions, which are defined in TIOA as the effect of a sequence of computations, to be represented in trans by explicit updates to state variables. This allows the theorem prover to reason directly about new state values of individual variables with less effort.

One goal in developing support for interactive theorem proving is to find a minimal set of proof steps that are natural to use in high level reasoning and that are sufficient (or nearly so) for mechanizing proofs of properties. Studying many examples has helped us in this regard. For example, we observed that many proofs included the observation that time cannot pass beyond a given deadline unless some discrete action occurs. This observation led us to include deadline\_reason among our set of proof steps.

Mechanizing proofs. The theorem proving support we are developing for TIOA does not make mechanizing proofs of properties automatic, but it does make it simpler. A user who wishes to prove properties of a TIOA specification using TAME must in general be a domain expert for the system modeled in TIOA. To prove the desired safety or simulation properties, the user often must first find an appropriate set of supporting lemmas. Doing this may require some creativity; some guidance on how to go about it can be found in [24]. The user must also be able to sketch out at a high level why, based on the set of supporting lemmas, a given property is expected to hold. To produce a mechanical proof of the property, the user then can apply TAME reasoning steps that match this high level reasoning. Typically, this can be done using steps such as const\_facts, apply\_inv\_lemma, apply\_specific\_precond, deadline\_reason, and so on, to introduce the facts appealed to in each nontrivial case in the proof sketch, and then invoking try\_simp to do the “obvious” reasoning based on these facts.

While it is good to have a mechanical check of a proof’s validity, it is equally important to have some feedback on what went wrong if the mechanical check fails. For failed proofs, TAME provides some useful feedback: the saved TAME proof script can be used to detect the place in the proof where the proof breaks down. The user can then review the high level reasoning to see whether there is an error or if introducing additional facts can complete the proof.

Scalability. We have begun experimentation with using the TAME support for TIOA on larger examples. Our first larger example is the Small Aircraft Traffic System protocol SATS developed at NASA Langley. An abstract model of this system has been defined in [8]. An IOA version of this model has been represented and verified in PVS [29]. We have used the TIOA-to-TAME translator to represent the IOA model in TAME, and have begun redoing the proofs using the TAME strategies.

The SATS example has raised an issue that is likely to arise in many large examples: the use by specifiers of multi-layered definitions of application-specific functions and predicates. One way to manage the many definition expansions for proof efficiency would be to expand them in layers to allow reasoning to proceed at the highest possible layer. A goal for the translator is to generate “local strategies” for a specific application that group definitions by layer. A scheme of this sort is used in the SCR-to-TAME translator to increase the efficiency of the TAME strategies that support reasoning about SCR automata [3].

6 Related work

Previous work has been performed to develop tools to translate specifications written in the IOA language to the language of various theorem provers, for example, Larch [6, 10], PVS [7], and Isabelle [30, 27]. Our implementation of the TIOA to PVS translator described in [20] builds upon [6]. The target PVS specifications of this trans-
lator strongly resemble TAME specifications. In addition, an early version of TAME's deadline_reason strategy was implemented as the PVS strategy deadline_check described in [20]. The TIOA-to-TAME translator is essentially a version of the TIOA-to-PVS translator of [20] with modifications that allow the straightforward implementation of new TAME strategies for TIOA and the most effective use of existing TAME strategies. A more complete description of the recent improvements made to the translator scheme and strategies described in [20] can be found in [19]. In [12], a slightly different approach using urgency predicates instead of stopping conditions or invariants to limit trajectories is used to describe timed I/O automata. An approach to proving invariant properties of timed I/O automata using urgency predicates is described, but no tool support. A proposed design for supporting urgency predicates in the TIOA toolkit is given in [4].

7 Conclusion

The TIOA framework is ultimately intended to support all phases of system development from specification, through verification and validation, to implementation. In this paper, we have focused on the usability of the TIOA framework for modeling and mechanical verification of properties of timed systems with both discrete and continuous transitions. We have described the theorem proving support provided, and illustrated how it is used in examples where the properties of interest are invariant properties or simulation properties, and where the models involve non-trivial data types.

Our plan for the future is experiment with more complex examples, such as SATS or the Dynamic Host Configuration Protocol DHCP (using models based on the work described in [13]), to explore extensions and improvements to our proof support.

Acknowledgements

We wish to thank the anonymous reviewers for helpful suggestions for improvements to this paper.

References