Des Autom Embed Syst
DOI 10.1007/s10617-008-9022-2

Specifying and proving properties of timed I/O automata using Tempo

Myla Archer · Hongping Lim · Nancy Lynch · Sayan Mitra · Shinya Umeno

Received: 13 November 2006 / Accepted: 29 April 2008
© Springer-Verlag 2008

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 *Tempo Toolset*, currently under development, is aimed at supporting system development based on TIOA specifications. The Tempo Toolset 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 the modeling of timed systems and their properties with TIOA and on the use of TAME4TIOA, the TAME[1] (Timed Automata Modeling Environment) based theorem proving support provided in Tempo, for proving system properties, including timing properties. Several examples are provided by way of illustration.

[1] TAME is a trademark of the U.S. Naval Research Laboratory.

This research is funded by the Air Force Office of Scientific Research and the Office of Naval Research.

M. Archer (✉)
Naval Research Laboratory, Code 5546, Washington, DC 20375, USA
e-mail: archer@itd.nrl.navy.mil

H. Lim · N. Lynch · S. Mitra · S. Umeno
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

H. Lim
e-mail: hongping@mit.edu

N. Lynch
e-mail: lynch@mit.edu

S. Mitra
e-mail: mitras@mit.edu

S. Umeno
e-mail: umeno@mit.edu

Keywords System development frameworks · Modeling environments · Tool suites · Automata models · Timed automata · Hybrid systems · Formal methods · Specification · Verification · Theorem proving

1 Introduction

For the development of high assurance complex systems, an appropriate development framework supporting system specification, implementation, and analysis is essential. To be generally usable, the support provided by the framework should apply not only to those systems that can be modeled as finite state machines but also 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 development and analysis tools to the extent possible.

The Tempo Toolset, based on the Timed Input/Output Automaton (TIOA) specification language and tools [10, 16, 27], provides just such a framework. The TIOA model [17] is especially suited to the specification and analysis of real-time, embedded systems.

The Tempo Toolset provides a spectrum of complementary analysis tools, including support for simulation of specifications and analysis of specification properties using model checking and theorem proving. The focus of this paper is on Tempo's support for theorem proving. With a set of small examples, we illustrate how one can use Tempo to model timed systems and specify their properties in the TIOA language, and then verify the specified properties using the theorem prover PVS [34] through the interface TAME4TIOA, an extension of TAME [3].

This paper is an expanded version of [6]. It makes at least three contributions: First, it demonstrates the use of the Tempo Toolset by showing how it can be used to specify three example systems and their properties of interest and to verify these properties using PVS through TAME4TIOA. Second, it describes how TAME has been extended to TAME4TIOA by adding new strategies for reasoning about the trajectories in a TIOA specification, and illustrates how these new strategies can be used in mechanized proofs of invariant properties. Third, it illustrates by example the improved translation scheme that supports the new strategies.

The paper is organized as follows. Section 2 gives an overview of the Timed I/O Automaton (TIOA) model and the Tempo Toolset that supports its use. Section 3 describes how one can specify and prove properties of TIOA models and how Tempo 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 can be proved in a "natural", high-level fashion in PVS using the toolkit's TAME4TIOA support. Finally,

Sect. 5 discusses some lessons learned from these and other examples, Sect. 6 mentions some related work, and Sect. 7 describes our future plans and presents some conclusions.

Except where the distinction is significant, in the remainder of this paper, we will refer to TAME4TIOA simply as TAME.

2 Background

2.1 The TIOA model

The TIOA model is a timed version of the I/O (Input/Output) automaton model described in [24]. In the I/O automaton model, the states `states`($\mathbf{A}$) of an automaton $\mathbf{A}$ are represented as assignments of values to state variables, the possible initial states of $\mathbf{A}$ are a subset `init`($\mathbf{A}$) of `states`($\mathbf{A}$), and state transitions are the result of actions. The state transitions resulting from actions 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. Actions may have parameters. Typical uses of parameters are, e.g., to index the action by the system process performing it, or, for a shared action, to represent information that is being passed. To be compatible with an input action, an output action must have parameters whose types are compatible with (i.e., at least subtypes of) the types of the input action's parameters.

Timing can be added to I/O automata by various means: see, for example [25, 29]. In the TIOA model, time passage is modeled using *trajectories*, which represent paths through the state space that are followed during the passage of time. A trajectory is specified by (1) an evolve clause describing 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., [1, 2]). A detailed description of the theory of the TIOA model and its comparison with other models can be found in [17].

2.2 The Tempo toolset

The Tempo Toolset [10, 27] is a tool supported formal framework for system development based on specifications in the TIOA language. The formal theory underlying the Tempo (or TIOA) framework is described in detail in [16] and [17]. The theory distinguishes TIOA from the very similar HIOA (Hybrid Input/Output Automata) by restricting trajectories in TIOA to being internal rather than potentially external (and hence potentially sharable) actions. The TIOA language extends the IOA language [12], a programming and modeling language based on the I/O Automaton model, with additional constructs for describing timing characteristics of systems. The TIOA language constructs related to timing are discussed in Sect. 4, which also contains several example TIOA specifications; see, for instance, Figs. 2, 7, and 8.

Tempo extends and updates the IOA toolkit described in [12]. Like the IOA toolkit, Tempo provides a specification simulator, as well as both model checking and theorem proving support, for analyzing specifications. Using Tempo, a user can write a system specification in the TIOA language and apply one of the analysis tools to verify properties of the system, using an appropriate automatic translator, if necessary, to recast the specification in

Table 1 New TAME strategies for trajectories

TAME proof step	Effect
`(apply_traj_evolve t)`	Compute state time t from now
`(apply_traj_stop t)`	Deduce that the stopping condition cannot hold after time t in a trajectory T unless T ends at t
`(apply_traj_invariant t)`	Deduce trajectory invariant holds time t from now
`(deadline_reason t)`	Deduce trajectory cannot evolve more than time t if a deadline is reached time t from now

the language of a particular tool. Tempo's beta version 0.2.2 was released on February 6, 2008, and is available at www.veromodo.com.

The TIOA simulator [28] provides a way to test an automaton before developing correctness proofs. By simulating the execution of an automaton for a fixed number of steps, it is possible to either discover the presence of errors in the automaton's specification (by finding cases when a desired invariant fails), or increase the confidence that an automaton works as expected (by demonstrating that the desired invariants hold at all steps of the simulation).

For model checking an appropriately restricted class of timed systems in TIOA, an interface to UPPAAL [18] has been developed. The `TIOA2XTA` tool [37] of the TIOA toolkit translates specifications written in TIOA into XTA—the input language of the UPPAAL model checker—and invokes UPPAAL on the generated XTA code to perform model checking.

Because of the richness of the TIOA specification language, TIOA specifications may describe continuous behavior or involve unbounded parameters, making them unsuitable for verification directly with a model checker. Thus, Tempo supports theorem proving to handle the more complex cases. Theorem proving is supported by translating the TIOA specification of a system and its properties to be verified into a PVS specification of the system and the properties, after which PVS can be applied. The translation scheme followed uses a PVS theory template which is instantiated with the states, actions and transitions of an automaton. A tool [19, 22] developed as part of Tempo performs the translation automatically. The PVS theory template used in the translation scheme is a new variant of the TAME (Timed Automata Modeling Environment) [3, 5] automaton template, whose original variants supported modeling and proving properties of MMT automata [29] and SCR automata [15]. Following this template allows previously developed TAME proof steps to be used when doing proofs with PVS.

The proof support in Tempo includes new TAME strategies as well. 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 [20]. In particular, a special aim of the design of the TAME template for TIOA is to support TAME proof steps (which are implemented as PVS strategies) for reasoning about trajectories. Table 1 describes the new TAME proof steps for reasoning about trajectories. Example proofs using `apply_traj_evolve` and `deadline_reason` can be seen in Sect. 4.1, Fig. 6 and Sect. 4.2, Fig. 14 respectively. Additional TAME proof steps and the descriptions of their purposes can be found in Appendix A; a TAME proof using a richer subset of the TAME steps than shown in Figs. 6 and 14 can be seen in Fig. 21 in Appendix C. A short sample interaction with the PVS theorem prover during reasoning about a trajectory is shown in Appendix D.

3 Overview of the Tempo proof methodology

3.1 Proving invariant and simulation properties

The TIOA mathematical model is useful for specifying timed distributed systems and for supporting analysis of the systems for properties that can be represented as invariants and simulation relations. By organizing the description of possible changes of state into discrete transitions resulting from actions and continuous transitions resulting from following a trajectory, the model furnishes the basis for organizing proofs of such properties by induction over the length of an automaton execution into a systematic case analysis with respect to these actions and trajectories. Here, the length of an execution is measured as the total number of discrete actions plus the total number of interludes between discrete actions during which some trajectory is followed.

To prove a state invariant property, one must prove that the property holds in every reachable state of an automaton. Since every reachable state is reached during some execution of the automaton, it is enough to establish that for every $n > 0$, the property holds in every state reached after some execution of length n. The standard approach to doing this is to use induction on n. Because each execution begins with the automaton in an initial state, proof of a property by induction over the length of an execution is equivalent to a proof that the property holds in every initial state ($n = 0$), and that whenever the property holds in some state (after an execution of length n), it also holds in any possible next state in an extension of this execution (to length $n + 1$). But any possible next state is reached by a discrete action or a trajectory. Thus, one can drop the reference to n and simply organize the proof into a proof of the base case (initial state) and a set of induction cases (one for each action and trajectory).

The proof organization in the case of a simulation relation property is similar, though there are a few differences. A major difference is that a simulation relation from an automaton **A** to an automaton **B** is defined in terms of the *traces* of **A** and **B**, which consist of the sequences of external actions in executions. **A** and **B** must have corresponding external actions, and the goal of a proof of simulation is to establish trace inclusion, i.e., that the traces of **A** are a subset of the traces of **B**. The simulation relation is specified as a relation on `states`(**A**) $\times$ `states`(**B**), and the proof of simulation is done by showing that, modulo internal actions, executions of **A** and **B** with the same traces lead to states in **A** and **B** in the specified relation. Thus, the proof is organized around a base case, in which each initial state of **A** is shown to correspond to some initial state of **B**, and a set of induction cases, one for each external action of **A**.

Because of the standard organization of proofs of invariant and simulation properties, it is possible to partially automate such proofs. The representations of both types of property also lend themselves to a standard organization, which can be used to advantage in the partial automation.

3.2 Using PVS with Tempo

The Tempo methodology for theorem proving is illustrated in Fig. 1, which we reproduce from [22]. The methodology, updated since publication of [22], 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 as an instantiation of the TAME4TIOA template, and then (3) proving the properties in PVS using TAME4TIOA strategies. 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

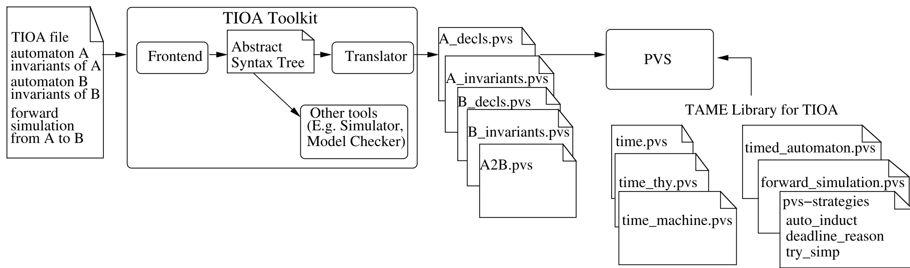

Fig. 1 TIOA framework for theorem-proving

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 TAME4TIOA libraries containing PVS definitions for timed I/O automata and any additional data type theories required, these generated files specify the automaton and its properties. For example, in Fig. 1, the file A_decls.pvs and A_invariants.pvs contain the TAME4TIOA representations of automaton **A** and its invariants; A2B.pvs contains the TAME4TIOA representation of the definition of the forward simulation relation from **A** to **B**; and the library theories time_machine.pvs and forward_simulation.pvs provide the lemmas supporting the case breakdown in induction proofs of invariants and forward simulation. The user then uses the TAME4TIOA strategies in pvs-strategies, which were adapted from TAME [3] or developed specifically 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, and obtains a PVS description in a form supporting the use of TAME4TIOA proof steps in the PVS theorem prover. Moreover, the translator also performs the tasks of translating (1) each transition definition, possibly given most naturally in TIOA by a sequence of program statements, into functional relations that capture the effects of the transition in PVS, and (2) each trajectory with differential equations into a time-passage action. A major additional benefit gained from specifying a system in TIOA rather than directly in PVS is that the user can also use other tools in the toolkit including the simulator, code generator and model checker.

TAME proof support is designed to allow a PVS user to prove a given property of an automaton by using proof steps that mimic the steps in a high level hand proof of the property. For example, the TAME step `auto_induct` (for "automaton induction") performs the initial case breakdown and simplifications (including the dismissing of trivial cases) of the proof by induction of an invariant property of an automaton, and presents the nontrivial cases as (labeled) subgoals; this mimics "the proof is by induction over the reachable states; only the following cases are nontrivial: ...". Thus, starting from a high level proof sketch of a property, a user can often simply choose corresponding TAME steps to follow this proof in order to easily construct a mechanized proof of the property. The TAME steps are also helpful in proof exploration, to users comfortable with interpreting the PVS sequents that represent subgoals in the proof. Another helpful feature of the TAME proof support is that the rerunnable, saved PVS scripts of proofs done using the TAME steps contain enough information to be understood without being rerun [5].

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 Tempo.

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 reach 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`, with `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_check` amount of time has elapsed. In the `check` phase, the process checks to see if `x` contains its process index. If so, it proceeds to the `critical` phase.

In the TIOA specification for `fischer` in Fig. 2, the state variable `pc[i]` represents the program counter, or phase, of process `i`, `last_set[i]` and `first_check[i]` are used to enforce the time bounds on `set` and `check`, and `now` represents the current time. The shared variable `x` is represented by the state variable `turn`, whose type `Null[process]` is the type `process` with an added "bottom" element `nil` to represent the case when `turn` is undefined.

The safety property we want to prove[1] is that no two processes are simultaneously in the `critical` phase. We also prove a set of auxiliary invariants to help us prove this main invariant. Auxiliary invariants are generally needed to prove a property that is not inductive. Rather than creating a complex, inductive invariant that combines an invariant with its auxiliary invariants, we find it more practical in an interactive theorem prover, where screen space is at a premium, to prove a sequence of simpler invariants, introducing earlier invariants as lemmas when they are needed in the proofs of later invariants. This approach also helps to make both the invariants and their proofs more human understandable. Figure 3

[1] Note that the `fischer` example is a clear candidate for the use of theorem proving rather than model checking to establish properties, since its specification applies to a set of processes that has unknown size.

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

 6   automaton fischer(l_check, u_set: Real) where
       u_set < l_check ∧ u_set ≥ 0 ∧ l_check ≥ 0
 8     imports fischer_types
       signature
10       output try(i: process)    internal test(i: process)
         output crit(i: process)   internal set(i: process)
12       output exit(i: process)   internal check(i: process)
         output rem(i: process)    internal reset(i: process)
14     states
         turn: Null[process] := nil,
16       now: Real := 0,
         pc: Array[process, PcValue] := constant(pc_rem),
18       last_set: Array[process, AugmentedReal] := constant(u_set),
         first_check: Array[process, Real] := constant(0)
20     transitions
         internal test(i)                    internal reset(i)
22         pre pc[i] = pc_test                 pre pc[i] = pc_reset
           eff if turn = nil then              eff pc[i] := pc_leaveexit;
24                pc[i] := pc_set;                  turn := nil;
                  last_set[i] :=
26                  now + u_set              output try(i)
               fi                              pre pc[i] = pc_rem
28                                             eff pc[i] := pc_test
         internal set(i)
30         pre pc[i] = pc_set                output crit(i)
           eff turn := embed(i);               pre pc[i] = pc_leavetry
32             pc[i] := pc_check;              eff pc[i] := pc_crit
               last_set[i] := \infty;
34             first_check[i] :=             output exit(i)
                 now + l_check;                pre pc[i] = pc_crit
36                                             eff pc[i] := pc_reset
         internal check(i)
38         pre pc[i] = pc_check ∧            output rem(i)
                 first_check[i] ≤ now         pre pc[i] = pc_leaveexit
40         eff if turn = embed(i) then         eff pc[i] := pc_rem;
                  pc[i] := pc_leavetry
42             else
                  pc[i] := pc_test
44             fi;
               first_check[i] := 0;
46
       trajectories
48       trajdef traj
           stop when
50           ∃ i: process (now = last_set[i])
           evolve
52           d(now) = 1
```

Fig. 2 TIOA specification for `fischer`

shows the sequence of invariants we proved for `fischer`, the last invariant being the safety property.

To illustrate how the various elements of an automaton specification in TIOA translate into TAME, Fig. 4 shows the TAME specification output by the TIOA-to-TAME transla-

```
1  invariant of fischer:
     ∀ k: process (pc[k] = pc_set ⇒ (last_set[k] ≤ (now + u_set)))
3
   invariant of fischer:
5    ∀ k: process (now ≤ last_set[k])

7  invariant of fischer:
     ∀ k: process (pc[k] = pc_set ⇒ last_set[k] ≠ \infty)
9
   invariant of fischer:
11   ∀ i: process ∀ j: process
       (pc[i] = pc_check ∧ turn = embed(i) ∧ pc[j] = pc_set
13      ⇒ first_check[i] > last_set[j])

15 invariant of fischer:
     ∀ i: process ∀ j: process
17     (pc[i] = pc_leavetry ∨ pc[i] = pc_crit ∨ pc[i] = pc_reset
        ⇒ turn = embed(i) ∧ pc[j] ≠ pc_set)
19
   invariant of fischer:
21   ∀ i: process ∀ j: process (i ≠ j ⇒ pc[i] ≠ pc_crit ∨ pc[j] ≠ pc_crit)
```

Fig. 3 TIOA invariants for `fischer`

tor applied to the TIOA specification in Fig. 2. The TAME specification has been edited slightly to save space. In the TAME specification, automaton parameters are translated as (uninterpreted) 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 `actions` data type in the TAME specification which has two parameters: `delta_t`, 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_evolve` capture the invariant, stopping condition and evolve clause of the trajectory definition respectively. The effect of the "trajectory action" `nu_traj` is constrained—and thus, effectively, captured—by the precondition of `nu_traj`, which asserts that (1) the invariant holds throughout the duration of the trajectory, (2) the stopping condition can hold in the trajectory only at the last state of the trajectory, and (3) the evolution of the state variables satisfies the evolve clause. The transition function for `nu_traj` simply returns the post-state obtained by applying the trajectory function `F` after an elapsed time of `delta_t`. (Following a technique of Luchangco [23], a slight variation of this trajectory representation method, in which the trajectory action has an additional, "new state" argument, appropriately constrained in the precondition, allows a trajectory action to have a nondeterministic effect, with `trans` remaining a function from actions and states to states.)

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 Fig. 3 to be proved interactively in PVS in a clear, high-level fashion. The TIOA-to-TAME translator trans-

```
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?
  exit(i: process): exit?
  rem(i: process): rem?
  test(i: process): test?
  set(i: process): set?
  check(i: process): check?
  reset(i: process): reset?
END actions
visible?(a:actions): bool = try?(a) OR crit?(a) OR exit?(a) OR rem?(a)
timepassageaction?(a:actions): bool = nu_traj?(a)
traj_invariant(a:(timepassageaction?))(s:states):bool = CASES a OF
  nu_traj(delta_t, F): TRUE 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,
  crit(i): pc(s)(i) = pc_leavetry,
  exit(i): pc(s)(i) = pc_crit,
  rem(i): pc(s)(i) = pc_leaveexit,
  test(i): pc(s)(i) = pc_test,
  set(i): pc(s)(i) = pc_set,
  check(i): pc(s)(i) = pc_check AND first_check(s)(i) <= now(s),
  reset(i): pc(s)(i) = pc_reset
  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?,
   lambda(a:(timepassageaction?)): dur(delta_t(a))]
END fischer_decls
```

Fig. 4 TAME representation of `fischer`

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

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

Fig. 5 TAME `lemma_5` for `fischer`

forms the six invariants in Fig. 3 into TAME invariants and lemmas numbered starting from 0. Thus, the goal safety property, the last invariant in Fig. 3, becomes the TAME invariant/lemma pair shown in Fig. 5.

Figure 6 shows a verbose TAME proof of `lemma_5` in Fig. 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 invariant of 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_traj(delta_t_action, F_action)` and `crit(i_action)` are trivial. The `nu_traj(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_evolve` 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 twice uses `apply_inv_lemma` to apply the invariant lemma 4: first to `i_theorem` and `j_theorem` (the skolem constants for the `i` and `j` under the universal quantifier in `Inv_5`), and then, symmetrically, to `j_theorem` and `i_theorem`. The PVS formulation of lemma 4, which corresponds to the fifth invariant in Fig. 3, is provided in the comment after each use of `apply_inv_lemma`. After the two appeals to lemma 4, only "obvious" reasoning with `try_simp` is needed to complete the proof.

4.2 A two task race

The two-task race system `TwoTaskRace` (see Fig. 7 for its TIOA description) increments a variable `count` repeatedly, after time passage of between `a1` and `a2` time units, `a1` $\leq$ `a2`, until it is interrupted by a `set` action. This `set` action can occur between `b1` and `b2` time from the start, where `b1` $\leq$ `b2`. After `set`, the value of `count` is decremented, again at time intervals between `a1` and `a2` units, and a `report` action is triggered when `count` reaches 0. The `report` action in `TwoTaskRace` sets the variable `reported` to `true`, and no `TwoTaskRace` action can reset this variable to `false`. Thus, the time of the `report` action is the same as the time in which `reported` becomes `true`. We want to show that the time bounds on the occurrence of the `report` action are: lower bound: $\texttt{if a2 < b1 then min(b1,a1) +}\ \frac{\texttt{(b1-a2)*a1}}{\texttt{a2}}\ \texttt{else a1}$, and upper bound: `b2 +`

```
;;; Proof lemma_5-1 for formula fischer_invariants.lemma_5
;;; developed with shostak decision procedures
(""
 (auto_induct)
 (("1" ;; Case nu_traj(delta_t_action, F_action)
   (apply_specific_precond)
   ;; Applying the precondition
   ;; (FORALL (t: (interval(zero, delta_t_action))):
   ;;    traj_invariant(nu_traj(delta_t_action, F_action))
   ;;                   (F_action(t)))
   ;;  AND
   ;;  (FORALL (t: (interval(zero, delta_t_action))):
   ;;     traj_stop(nu_traj(delta_t_action, F_action))
   ;;              (F_action(t))
   ;;      => t = delta_t_action)
   ;;   AND
   ;;   (FORALL (t: (interval(zero, delta_t_action))):
   ;;      F_action(t) =
   ;;       traj_evolve(nu_traj(delta_t_action, F_action))
   ;;                   (t, prestate))
   (apply_traj_evolve "delta_t_action")
   ;; Using the fact that
   ;;  F_action(delta_t_action) =
   ;;  prestate WITH
   ;;    [now := 1 * dur(delta_t_action) + now(prestate)]
   (try_simp))
  ("2" ;; Case crit(i_action)
   (apply_specific_precond)
   ;; Applying the precondition
   ;; pc(prestate)(i_action) = pc_leavetry
   (apply_inv_lemma "4" "i_theorem" "j_theorem")
   ;; Applying the lemma
   ;; FORALL (i: process, j: process):
   ;;  pc(prestate)(i) = pc_leavetry OR
   ;;  pc(prestate)(i) = pc_crit OR pc(prestate)(i) = pc_reset
   ;;  => turn(prestate) = up(i) AND pc(prestate)(j) /= pc_set
   (apply_inv_lemma "4" "j_theorem" "i_theorem")
   ;; Applying the lemma
   ;; FORALL (i: process, j: process):
   ;;  pc(prestate)(i) = pc_leavetry OR
   ;;  pc(prestate)(i) = pc_crit OR pc(prestate)(i) = pc_reset
   ;;  => turn(prestate) = up(i) AND pc(prestate)(j) /= pc_set
   (try_simp))))
```

Fig. 6 Verbose TAME proof of `lemma_5` in `fischer`

$\texttt{a2} + \frac{\texttt{b2*a2}}{\texttt{a1}}$. This property is proved by first specifying an appropriate abstract automaton `TwoTaskRaceSpec` that performs a `report` action within these bounds (see Fig. 8), then defining a relation from `TwoTaskRace` to `TwoTaskRaceSpec` (see Fig. 10) that maintains equality between the values of `reported` and the current time `now` in the two automata, and finally, establishing that the relation is a forward simulation.

The relation defined in Fig. 10 is expressed as a universally quantified implication which, when instantiated by the (shared) parameters of `TwoTaskRace` and `TwoTaskRaceSpec`,

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

4     signature
        internal increment
6       internal decrement
        internal set
8       output report
      states
10      count: Int := 0,
        flag: Bool := false,
12      reported: Bool := false,
        now: Real := 0,
14      first_main: Real := a1,
        last_main: AugmentedReal := a2,
16      first_set: Real := b1,
        last_set: AugmentedReal := b2
18    transitions
        internal increment
20        pre ¬flag ∧ now ≥ first_main
          eff count := count + 1;
22            first_main := now + a1;
              last_main := now + a2
24      internal set
          pre ¬flag ∧ now ≥ first_set
26        eff flag := true;
              first_set := 0;
28            last_set := \infty
        internal decrement
30        pre flag ∧ count > 0 ∧ now ≥ first_main
          eff count := count - 1;
32            first_main := now + a1;
              last_main := now + a2
34      output report
          pre flag ∧ count = 0 ∧ ¬reported ∧ now ≥ first_main
36        eff reported := true;
              first_main := 0;
38            last_main := \infty
      trajectories
40      trajdef traj
          stop when now = last_main ∨ now = last_set
42        evolve
            d(now) = 1
```

Fig. 7 `TwoTaskRace` in TIOA

has hypotheses that can be discharged by state invariants, and a conclusion in six parts. The first two parts of the conclusion guarantee the required correspondence between `now` and `reported` in the two automata. The remaining four parts support the proof that this relation is a simulation relation.

The abstract automaton `TwoTaskRaceSpec` has two trajectories: `pre_report` and `post_report`. The TAME representation of `TwoTaskRaceSpec` (see Fig. 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 Fig. 10 of the forward simulation relation into the PVS theory in Fig. 11 that asserts (as a theorem to be

```
   automaton TwoTaskRaceSpec(a1, a2, b1, b2: Real) where
2    a1 > 0 ∧ a2 > 0 ∧ b1 ≥ 0 ∧ b2 > 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1
     signature
4      output report
     states
6      reported: Bool := false,
       now: Real := 0,
8      first_report: Real :=
         if a2 < b1 then min(b1, a1) + (((b1 - a2) * a1) / a2) else a1,
10     last_report: AugmentedReal :=
         b2 + a2 + ((b2 * a2) / a1)
12   transitions
       output report
14       pre ¬reported ∧ now ≥ first_report
         eff reported := true;
16           first_report := 0;
             last_report := \infty
18   trajectories
       trajdef pre_report
20       invariant ¬reported
         stop when now = last_report
22       evolve
           d(now) = 1
24     trajdef post_report
         invariant reported
26       evolve
           d(now) = 1
```

Fig. 8 `TwoTaskRaceSpec` in TIOA

proved) the property `forward_simulation`. The theory in Fig. 11 follows the TAME template for formulating abstraction relations between automata described in [30]. The parameterized *theory* `forward_simulation`, imported with appropriate actual parameters in Fig. 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, the gist of which is described above in Sect. 3, is based on the definition in [26]. The proof of this property for `TwoTaskRace` and `TwoTaskRaceSpec` uses invariants of both automata. A high level view of the saved proof of this property can be seen in Fig. 25 in Appendix F.

The invariants of `TwoTaskRace` and `TwoTaskRaceSpec` needed for the forward simulation proof are a subset of the invariants shown in Figs. 12 and 13, all of which have been proved in TAME. Although the discovery of invariants is not always so simple, these invariants were straightforward to discover, either by a "bottom-up" approach based on an intuitive understanding of the specifications or by the "top-down" approach from proof goals at dead ends in a simulation proof attempt. The proofs of these invariants are all quite simple; in fact, most of the `TwoTaskRaceSpec` invariants can be proved using just the TAME induction strategy `auto_induct` followed by the sequence `(apply_specific_precond)`, `(apply_traj_evolve "delta_t_action")`, `(try_simp)` for each of the two (trajectory) actions. 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 Fig. 12, whose TAME proof is shown in Fig. 14. Invariant 4 essentially says that in the TIOA model of `TwoTaskRace`, the current time `now` cannot pass beyond the deadline `last_main`. In this

```
TwoTaskRaceSpec_decls : THEORY BEGIN
. . .
% Trajectory invariants
traj_invariant(a:(timepassageaction?)) (s:states):bool = CASES a OF
    nu_pre_report(delta_t,F): NOT reported(s),
    nu_post_report(delta_t,F): reported(s)
  ENDCASES
% Trajectory stopping conditions
traj_stop(a:(timepassageaction?)) (s:states):bool = CASES a OF
    nu_pre_report(delta_t,F): fintime(now(s))=last_report(s),
    nu_post_report(delta_t,F): true
  ENDCASES
% Trajectory evolve clauses
traj_evolve(a:(timepassageaction?)) (t:(fintime?),s:states):states = CASES a OF
    nu_pre_report(delta_t,F): s WITH [now := now(s) + 1 * dur(t)],
    nu_post_report(delta_t,F): s WITH [now := now(s) + 1 * dur(t)]
  ENDCASES
% Enabled
enabled(a:actions, s:states):bool = CASES a OF
    nu_pre_report(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_post_report(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)),
    report: NOT reported(s) AND now(s) >= first_report(s)
  ENDCASES
% Transition function
trans(a:actions, s:states):states = CASES a OF
    nu_pre_report(delta_t,F): F(delta_t),
    nu_post_report(delta_t,F): F(delta_t),
    report: s WITH [last_report := infinity,
                    reported := true,
                    first_report := 0]
  ENDCASES
. . .
END TwoTaskRaceSpec_decls
```

Fig. 9 `TwoTaskRaceSpec` trajectories in TAME

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_t_action,F_action)`. After using `apply_specific_precond` and `apply_traj_evolve` to compute the state after time `delta_t_action` and using `apply_inv_lemma` to use invariant 1 to establish that `now >= 0` at the beginning of the trajectory, the new TAME step `deadline_reason` argues that `now <= last_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 `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`. They also typically have fewer induction cases (see Sect. 3.1). For more details, see [30].

```
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 ∧ a2 ≥ a1 ∧ b2 ≥ b1
  ∧ last_set≥0 ∧ last_set=TwoTaskRace.last_set
  ∧ last_main≥0 ∧ last_main=TwoTaskRace.last_main
  ∧ last_report≥0 ∧ last_report=TwoTaskRaceSpec.last_report
  ⇒ TwoTaskRace.reported = TwoTaskRaceSpec.reported
   ∧ TwoTaskRace.now = TwoTaskRaceSpec.now
   ∧ (¬TwoTaskRace.flag ∧ last_main < TwoTaskRace.first_set
      ⇒ TwoTaskRaceSpec.first_report ≤
        (min(TwoTaskRace.first_set, TwoTaskRace.first_main)
        + ((TwoTaskRace.count
        + ((TwoTaskRace.first_set - last_main) / a2)) * a1)))
   ∧ (TwoTaskRace.flag ∨ last_main ≥ TwoTaskRace.first_set
      ⇒ TwoTaskRaceSpec.first_report ≤
        (TwoTaskRace.first_main + (TwoTaskRace.count * a1)))
   ∧ (¬TwoTaskRace.flag ∧ TwoTaskRace.first_main ≤ last_set
      ⇒ last_report ≥ (last_set + ((TwoTaskRace.count + 2
        + ((last_set - TwoTaskRace.first_main) / a1)) * a2)))
   ∧ (¬(TwoTaskRace.reported) ∧ (TwoTaskRace.flag ∨
          TwoTaskRace.first_main > last_set) ⇒ last_report
          ≥ (last_main + (TwoTaskRace.count * a2))))
```

Fig. 10 Forward simulation from `TwoTaskRace` to `TwoTaskRaceSpec`

4.3 A simple timeout system

A simple timeout system consists of a sender, a delay prone channel, and a receiver. The sender sends messages to the receiver, within `u1` time after the previous message has been sent. A timed message queue, in which messages are queued together with their delivery deadline, is used to model the fact that the delivery of each message is delayed 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. When the receiver times out, the sender is considered to be "suspected". Figure 15 shows the TIOA description of the abstract automaton `timeout` for this simple timeout system. In this description, the boolean valued variables `suspected` and `failed` indicate whether the sender is suspected or failed, respectively. The variable `queue` represents the timed message queue, and the variables `p_clock` and `t_clock` are used, respectively, to keep track of the current deadlines on the `send` and `timeout` actions. The variable `now`, as usual, represents the current time.

We are interested in proving the two 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. Because the flag `suspected` becomes true only when a timeout occurs, the Safety property can be captured by the system invariant `suspected ⇒ failed` on line 23 of Fig. 16 (invariant 7, numbering the invariants from 0). As in the two-task race example, to establish the Timeliness property, we first create an abstract automaton `timeout_spec` that times out within `u2 + b` time of occurrence of a failure, and then prove a forward simulation from the system `timeout` to its abstraction `timeout_spec`. The TIOA description of `timeout_spec` is shown in Fig. 17. The state variables `suspected`, `failed`, and `now` in this description have the same meanings as the analogous variables in the TIOA description of `timeout`; a new variable `last_timeout` is used to keep track of the current deadline on the `timeout` action. The definition of the actual forward simulation relation between `timeout` and `timeout_spec`, which is similar in form to the definition in Fig. 10, is given in Appendix E.

```
TwoTaskRace2TwoTaskRaceSpec: THEORY BEGIN
IMPORTING TwoTaskRace_invariants
IMPORTING TwoTaskRaceSpec_invariants
timed_auto_lib: LIBRARY = "../timed_auto_lib"
MA: THEORY = timed_auto_lib@timed_automaton
             :-> TwoTaskRace_decls
MB: THEORY = timed_auto_lib@timed_automaton
             :-> TwoTaskRaceSpec_decls
amap(a_A: {a: MA.actions |
   visible?(a) AND NOT timepassageaction?(a)}):MB.actions =
  CASES a_A of report: report ENDCASES
ref(s_A: MA.states, s_B: MB.states): bool =
  FORALL (last_set: real, last_main: real, last_report: real):
   a1>0 AND a2>0 AND b1>=0 AND b2>0 AND a2>=a1 AND b2>=b1
   AND last_set >= 0 AND fintime(last_set) = last_set(s_A)
   AND last_main >= 0 AND fintime(last_main) = last_main(s_A)
   AND last_report >= 0 AND fintime(last_report) = last_report(s_B)
   => reported(s_A) = reported(s_B) AND now(s_A) = now(s_B)
    AND (NOT flag(s_A) AND last_main < first_set(s_A) =>
        first_report(s_B) <= min(first_set(s_A), first_main(s_A))
        + count(s_A) + (first_set(s_A) - last_main)/a2*a1)
    AND (flag(s_A) OR last_main >= first_set(s_A) =>
         first_report(s_B) <= first_main(s_A) + count(s_A)*a1)
    AND (NOT flag(s_A) AND first_main(s_A) <= last_set =>
         last_report >= last_set + count(s_A) + 2
                         + (last_set - first_main(s_A))/a1*a2)
    AND (NOT reported(s_A)
         AND (flag(s_A) OR first_main(s_A) > last_set)
         => last_report >= last_main + count(s_A) * a2)
IMPORTING timed_auto_lib@forward_simulation[MA, MB, ref,
 (LAMBDA(a:MA.actions): timepassageaction?(a)),
 (LAMBDA(a:{a:MA.actions|timepassageaction?(a)}):dur(delta_t(a))),
 amap]
fw_simulation_thm: THEOREM forward_simulation
END TwoTaskRace2TwoTaskRaceSpec
```

Fig. 11 TwoTaskRace to TwoTaskRaceSpec simulation relation in TAME

Both the Safety and Timeliness properties have been proved using the TAME strategies in a manner analogous to the invariant and forward 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 uses a custom data type `timed_message_Queue`. TIOA provides a `vocabulary` syntax to allow the user to declare custom data types and operators. Figure 18 shows how the data type for `timed_message_Queue` and the associated operators are declared in TIOA. The actual definitions of the types and operators in the `timed_queue` vocabulary come from PVS, and are provided as part of a TIOA library of PVS data type theories; Fig. 19 shows a sample of these definitions. Aside from the PVS operator `enQ?` (which implements the TIOA operator `enQ_qn` for querying whether a `timed_message_Queue` is a nonempty queue), the PVS vocabulary is identical to the TIOA vocabulary. Properties of these types and operators have been proved in PVS, and have been used in proofs of the Safety and Timeliness properties. The `timed_queue` properties which were actually used in proofs of `timeout` and `timeout_spec` are shown in Appendix B.

```
1  invariant of TwoTaskRace:
     a1 ≥ 0 ∧ a2 > 0 ∧ b1 ≥ 0 ∧ b2 > 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1
3
   invariant of TwoTaskRace: now ≥ 0
5
   invariant of TwoTaskRace: (now + b2) ≥ 0
7
   invariant of TwoTaskRace: flag ⇒ last\_set = \infty
9
   invariant of TwoTaskRace: now ≥ 0 ⇒ last\_main ≥ now
11
   invariant of TwoTaskRace: now ≥ 0 ⇒ last\_set ≥ now
13
   invariant of TwoTaskRace:
15   b2 ≥ 0 ∧ now ≥ 0 ∧ last\_set ≠ \infty ⇒ last\_set ≤ (b2 + now)

17 invariant of TwoTaskRace:
     a2 ≥ 0 ∧ now ≥ 0 ∧ last\_main ≠ \infty ⇒ last\_main ≤ (a2 + now)
19
   invariant of TwoTaskRace: first\_main ≤ (a1 + now)
21
   invariant of TwoTaskRace: first\_set ≤ (b1 + now)
23
   invariant of TwoTaskRace: reported ⇒ flag
25
   invariant of TwoTaskRace: reported ⇒ count = 0
27
   invariant of TwoTaskRace: ¬reported ⇔ last\_main ≠ \infty
29
   invariant of TwoTaskRace:
31   (¬reported ⇔ last\_main ≠ \infty) ∧ (¬flag ⇔ last\_set ≠ \infty)

33 invariant of TwoTaskRace: ¬flag ⇒ last\_set ≠ \infty

35 invariant of TwoTaskRace:
     last\_main ≠ \infty ∧ ¬flag ⇒ ((last\_set + a2) - last\_main) ≥ 0
37
   invariant of TwoTaskRace:
39   a1 ≥ 0 ∧ ¬flag ∧ first\_main ≥ 0
       ⇒ ((2 * a1) + (last\_set - first\_main)) ≥ 0
41
   invariant of TwoTaskRace:
43   first\_main ≥ 0 ⇒ first\_main ≤ last\_main
```

Fig. 12 TwoTaskRace invariants 0–19

```
1  invariant of TwoTaskRaceSpec: now ≥ 0

3  invariant of TwoTaskRaceSpec: ¬reported ⇒ last_report ≠ \infty

5  invariant of TwoTaskRaceSpec:
     (now ≥ 0 ∧ ¬reported ⇒ last_report ≠ \infty) ∧
7    (TwoTaskRaceSpec.now ≤ last_report ∧ reported ⇒ last_report = \infty)

9  invariant of TwoTaskRaceSpec: reported ⇒ last_report = \infty
```

Fig. 13 TwoTaskRaceSpec invariants 0–3

Both the Queue and timed_message_Queue data types and theories have proved useful in other examples. Because developing PVS theories for complex data types is not trivial, a goal for Tempo is to populate the library tioa_types_lib with theories of data types that are apt to be reused.

```
;;; Proof lemma_4-1 for formula TwoTaskRace_invariants.lemma_4
;;; developed with shostak decision procedures
(""
 (auto_induct)
 (("1" ;; Base case
   (const_facts)
   ;; Applying the facts about the constants:
   ;; a1 > 0 AND a2 > 0 AND b1 >= 0 AND b2 > 0 AND a2 >= a1 AND b2 >= b1
   (try_simp))
  ("2" ;; Case nu_traj(delta_t_action, F_action)
   (apply_specific_precond)
   ;; Applying the precondition
   ;; (FORALL (t: (interval(zero, delta_t_action))):
   ;;    traj_invariant(F_action(t)))
   ;;  AND
   ;;  (FORALL (t: (interval(zero, delta_t_action))):
   ;;     traj_stop(F_action(t)) => t = delta_t_action)
   ;;   AND
   ;;   (FORALL (t: (interval(zero, delta_t_action))):
   ;;      F_action(t) = traj_evolve(t, prestate))
   (apply_traj_evolve "delta_t_action")
   ;; Using the fact that
   ;;  F_action(delta_t_action) =
   ;;  prestate WITH
   ;;   [now := 1 * dur(delta_t_action) + now(prestate)]
   (apply_inv_lemma "1")
   ;; Applying the lemma
   ;; now(prestate) >= 0
   (deadline_reason "last_main(prestate)")
   ;; Reasoning that time cannot pass beyond last_main(prestate)
   (try_simp))
  ("3" ;; Case increment
   (const_facts)
   ;; Applying the facts about the constants:
   ;; a1 > 0 AND a2 > 0 AND b1 >= 0 AND b2 > 0 AND a2 >= a1 AND b2 >= b1
   (try_simp))
  ("4" ;; Case decrement
   (const_facts)
   ;; Applying the facts about the constants:
   ;; a1 > 0 AND a2 > 0 AND b1 >= 0 AND b2 > 0 AND a2 >= a1 AND b2 >= b1
   (try_simp))
  ("5" ;; Case report
   (try_simp))))
```

Fig. 14 Proof of `TwoTaskRace` invariant 4

5 Discussion

5.1 Developing theorem proving support

Our approach to developing appropriate theorem proving support for Tempo 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

```
    automaton timeout(u1, u2, b: Real)
2     where u1 ≥ 0 ∧ u2 ≥ 0 ∧ b ≥ 0 ∧ u2 > (u1 + b)
      imports timed_queue
4     signature
        internal send(m: M)
6       internal receive(m: M)
        output fail
8       output timeout
      states
10      p_clock: AugmentedReal := 0,
        t_clock: AugmentedReal := u2,
12      suspected: Bool := false,
        failed: Bool := false,
14      now: Real := 0,
        queue: timed_message_Queue := mtQ
16    transitions
        internal send(m)
18        pre now ≥ 0 ∧ ¬failed ∧ p_clock = now
          eff if (now + u1) ≥ 0 then p_clock := now + u1 fi;
20            if (now + b) ≥ latest_deadline(queue) then
                 queue := enQ(MKtimed_message(m, now + b), queue)
22            fi;
        internal receive(m)
24        pre now ≥ 0 ∧ enQ_qn(queue) ∧ m = earliest_msg(queue)
          eff if (now + u2) ≥ 0 then t_clock := now + u2 fi;
26            if enQ_qn(queue) then queue := deQ(queue) fi
        output fail
28        pre ¬ failed
          eff failed := true;
30            p_clock := \infty
        output timeout
32        pre now ≥ 0 ∧ ¬suspected ∧ t_clock = now
          eff suspected := true;
34            t_clock := \infty
      trajectories
36      trajdef traj
          stop when now ≥ 0 ∧ (now = p_clock ∨ now = t_clock
38                  ∨ now = earliest_deadline(queue))
          evolve d(now) = 1
```

Fig. 15 TIOA description of the `timeout` system

a translator to PVS targets can, if chosen carefully, 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 Fig. 9 and the scheme for representing the start state predicate `start(s)` as an equality of the form `s = ...` (see, for example, Fig. 4), possibly in conjunction with additional restrictions. 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. A full description of the lessons we have learned about template details can be found in [21].

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 assertion that time cannot

```
1  invariant time_ordered of timeout: time_ordered(queue)

3  invariant of timeout: now ≥ 0

5  invariant of timeout:
     now ≥ 0
7      ⇒ now ≤ t_clock ∧ now ≤ p_clock ∧ now ≤ earliest_deadline(queue)

9  invariant of timeout:
      (now + u2) ≥ 0 ∧ ¬suspected ⇒ t_clock ≠ \infty ∧ t_clock ≤ (now + u2)
11
   invariant of timeout:
13    (now + u1) ≥ 0 ∧ ¬failed ⇒ p_clock ≠ \infty ∧ p_clock ≤ (now + u1)

15 invariant of timeout: ∀ n: Nat
      (n ≤ (lengthQ(queue) - 1) ⇒ deadline(nthQ(queue, n)) ≤ (now + b))
17
   invariant of timeout:
19   b ≥ 0 ∧ ¬ failed
       ⇒ (if enQ_qn(queue) then earliest_deadline(queue) < t_clock
21           else (p_clock + b) < t_clock)

23 invariant of timeout: suspected ⇒ failed
```

Fig. 16 Invariant `time_ordered` and invariants 0–6 of the `timeout` system

pass beyond a given deadline unless some discrete action occurs. This observation led us to include `deadline_reason` among our set of proof steps. Based on examples we have seen so far, we believe we are at least close to having a minimal set or proof steps sufficient for mechanizing proofs of invariant properties, but expect to discover additional proof steps useful in proofs of simulation.

5.2 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 [26]. 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. Alternatively, the user can find a TAME expert to mechanize the proof. Typically, the proof mechanization 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.

```
   automaton timeout_spec(u1, u2, b: Real)
2    where u1 ≥ 0 ∧ u2 ≥ 0 ∧ b ≥ 0 ∧ u2 > (u1 + b)
     signature
4      output fail
       output timeout
6    states
       last_timeout: AugmentedReal := \infty,
8      now: Real := 0,
       suspected: Bool := false,
10     failed: Bool := false
     transitions
12     output fail
         pre ¬failed
14       eff if (now + u2 + b) ≥ 0 then last_timeout := now + u2 + b fi;
             failed := true
16     output timeout
         pre failed ∧ ¬suspected
18       eff suspected := true;
             last_timeout := \infty
20   trajectories
       trajdef beforeFailed_or_afterSuspected
22       invariant ¬failed ∨ (failed ∧ suspected)
         evolve
24         d(now) = 1
       trajdef failed_and_notSuspected
26       invariant failed ∧ ¬suspected
         stop when now ≥ 0 ∧ now = last_timeout
28       evolve
           d(now) = 1
```

Fig. 17 TIOA description of `timeout_spec`

```
vocabulary timed_queue
types M, timed_message_Queue, timed_message
operators
  mtQ: -> timed_message_Queue
  enQ_qn: timed_message_Queue -> Bool
  deQ: timed_message_Queue -> timed_message_Queue
  enQ: timed_message, timed_message_Queue -> timed_message_Queue
  MKtimed_message: M, Real -> timed_message
  earliest_msg: timed_message_Queue -> M
  earliest_deadline: timed_message_Queue -> AugmentedReal
  latest_deadline: timed_message_Queue -> Real
  time_ordered: timed_message_Queue -> Bool
  nthQ: timed_message_Queue, Nat -> M
  lengthQ: timed_message_Queue -> Nat
  deadline: M -> Real
```

Fig. 18 TIOA declaration of custom data types and operators used in `timeout`

5.3 Scalability

For the three examples described in Sect. 4, the Tempo theorem proving support has acceptable time performance. Translation of each of the TIOA specifications executes almost instantaneously. The TAME proofs of invariants run in times ranging from about 1 second to about 15 seconds, with most proofs executing in 1 to 5 seconds. The most time consuming proof steps are, unsurprisingly, those that perform the largest amount of automated reasoning, such as `auto_induct` and `try_simp`. The two TAME simulation proofs consume

```
Queue[T:TYPE]: DATATYPE
 BEGIN
  mtQ: mtQ?
  enQ(last:T, before_last:Queue): enQ?
 END Queue

Queue_thy[T:type]: THEORY
 BEGIN
  IMPORTING Queue[T]
  lengthQ(q:Queue): RECURSIVE nat =
   IF mtQ?(q) THEN 0 ELSE lengthQ(before_last(q)) + 1 ENDIF
  MEASURE reduce_nat(0, (LAMBDA (x:T), (n:nat): n+1));
  deQ(q:(enQ?)): RECURSIVE Queue =
    IF mtQ?(before_last(q)) THEN mtQ
    ELSE enQ(last(q),deQ(before_last(q))) ENDIF
  MEASURE lengthQ(q);
  nthQ(q:(enQ?), n:{i:nat| 0<=i & i<=lengthQ(q)-1}): RECURSIVE T =
   IF n=0 THEN last(q) ELSE nthQ(before_last(q),n-1) ENDIF
  MEASURE n;
  . . .
 END Queue_thy

timed_message_thy[M:TYPE]: THEORY
 BEGIN
  timed_auto_lib: LIBRARY = "../timed_auto_lib"
  IMPORTING timed_auto_lib@time_thy
  timed_message: TYPE = [# message:M, deadline:nonneg_real #]
  MKtimed_message(m:M, d:nonneg_real):timed_message =
    (# message := m, deadline := d #);
  . . .
 END timed_message_thy

timed_message_Queue_thy[M:TYPE] : THEORY
 BEGIN
  . . .
  IMPORTING timed_message_thy[M]
  IMPORTING Queue_thy[timed_message]
  timed_message_Queue: TYPE = Queue[timed_message];
  time_ordered(q:timed_message_Queue): bool =
    FORALL (i:[upto(lengthQ(q)-1)], j:{n:nat | n>=i & n<=lengthQ(q)-1}):
    deadline(nthQ(q,i)) >= deadline(nthQ(q,j));
  earliest_deadline(q:timed_message_Queue): time = CASES q OF
      mtQ: infinity,
      enQ(x,y): fintime(deadline(frontQ(q)))
    ENDCASES;
  latest_deadline(q:timed_message_Queue): nonneg_real = CASES q OF
      mtQ: 0,
      enQ(x,y): deadline(last(q))
    ENDCASES;
  . . .
 END timed_message_Queue_thy
```

Fig. 19 PVS definitions of custom data types and operators used in `timeout`

more time: on the order of 60 to 90 seconds. This may partly be because the current versions of these proofs make extensive use of `deduce`, which uses the PVS strategy `grind` (which makes a serious attempt to discharge its current proof goal automatically) in a hidden way. Individual proof steps in the simulation proofs execute in times ranging from a few seconds to a fraction of a second. Thus, interactive use of the theorem prover is reasonably efficient in terms of human time.

To explore the scalability of Tempo's theorem proving support, we have begun experimenting with Tempo and TAME 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 [9]. TIOA versions of both the discrete and hybrid models have been formulated, and the discrete model has been verified directly in PVS [36]. In addition, a candidate forward simulation relation from the hybrid TIOA model to the discrete TIOA model has been formulated and proved by hand to be a forward simulation [35]. We have used the TIOA-to-TAME translator to represent the discrete TIOA model in TAME, and have redone many of the invariant proofs using the TAME strategies. So far, the TAME proofs have been easier to construct, and their saved scripts much easier to understand and significantly shorter than the "raw PVS" versions. A mechanical verification of the hybrid model can be obtained by creating a TAME version of the forward simulation proof. From our experience with the invariant proofs for the discrete model of SATS, we are hopeful of being able to accomplish this.

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 [7, 11], PVS [8], and Isabelle [33, 38]. Our implementation of the TIOA-to-PVS translator described in [22] builds upon [7]. The target PVS specifications of this translator 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 [22]. The TIOA-to-TAME translator is essentially a version of the TIOA-to-PVS translator of [22] 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 translation scheme and strategies described in [22] can be found in [21]. In [13], 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 Tempo toolset is given in [4].

A hybrid model of the SATS system has been verified by encoding it as a discrete system and then applying a combination of explicit state model checking (implemented as a PVS strategy) and automated theorem proving in PVS [31, 32]. The argument for the correctness of the encoding is given in natural language. By contrast, the "encoding correctness" argument in the TIOA approach is done in terms of a forward simulation that can be verified mechanically. There is much more automation in the proof process described in [31, 32] than in the TIOA approach using TAME; however, this proof process involves first defining and implementing several application-specific techniques and strategies, and the extent to which these techniques and strategies can be reused in other applications remains to be clarified.

From `timed_message_Queue_thy.pvs`**:**

```
earliest_latest_lem: LEMMA FORALL(q:(nonemptyqueue?)):
  time_ordered(q) => dur(earliest_deadline(q)) <= latest_deadline(q);

enQ_ordered: LEMMA FORALL(q:timed_message_Queue, msg:timed_message):
  (time_ordered(q) & deadline(msg) >= latest_deadline(q))
  =>
  time_ordered(enQ(msg,q));

deQ_ordered: LEMMA FORALL(q:(nonemptyqueue?)):
  time_ordered(q) => time_ordered(deQ(q));

before_last_ordered: LEMMA FORALL(q:(nonemptyqueue?)):
  time_ordered(q) => time_ordered(before_last(q));

deQ_earliest: LEMMA FORALL(q:(nonemptyqueue?)):
  time_ordered(q) => earliest_deadline(q) <= earliest_deadline(deQ(q));
```

From `Queue_thy.pvs`**:**

```
last_deQ_queue: LEMMA (FORALL (q:(enQ?)):
  (lengthQ(q) >= 2) => (last(deQ(q)) = last(q)));

length_deQ_queue: LEMMA (FORALL (q:(enQ?)):
  lengthQ(deQ(q)) = lengthQ(q) - 1);

nth_elt_deQ_queue: LEMMA (FORALL (q:(enQ?),
                                  n:{i:nat| 0<=i & i<=lengthQ(q)-1}):
  (lengthQ(q) >= 2 & n <= lengthQ(q) - 2)
  =>
  (n <= lengthQ(deQ(q)) - 1 & nthQ(deQ(q),n) = nthQ(q,n)));

frontQ_nth_queue: LEMMA (FORALL (q:(enQ?)):
  frontQ(q) = nthQ(q, lengthQ(q)-1));

length_enQ_queue: LEMMA (FORALL (e:T,q:Queue):
  lengthQ(enQ(e,q)) = lengthQ(q) + 1);
```

Fig. 20 Lemmas about timed queues needed in proofs of `timeout` properties

7 Conclusion

Tempo 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 Tempo 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 to use experimentation with more complex examples, such as `SATS` or the Dynamic Host Configuration Protocol `DHCP` (using models based on the work described in [14]), to explore extensions and improvements to our proof support.

Acknowledgements We wish to thank the anonymous reviewers of earlier versions of this paper for helpful suggestions for improvements.

Appendix A: TAME proof steps for TIOA

Table 2 lists and describes most of the TAME proof steps (strategies) provided for TIOA, including all of those appearing in TAME proof scripts in this paper.

Table 2 Major TAME proof steps for TIOA

TAME proof step	Effect
(auto_induct)	Set up the proof by induction of an invariant and perform initial simplifications
(direct_proof)	Set up the non-induction proof of an invariant and perform initial simplifications
(apply_specific_precond)	Introduce the specific precondition of the current action in an induction step of a proof by induction
(prove_fwd_sim)	Set up the proof by induction of a forward simulation and perform initial simplifications
(apply_ind_hyp *args*)	Apply the inductive hypothesis in the current induction proof to the (non-default) values *args*
(apply_inv_lemma *lemma-tag args*)	Introduce the invariant lemma with tag *lemma-tag* instantiated with the values *args*
(use_defs *(formula-label) names*)	Use the definitions of the entities named in *names,* optionally only in formulas labeled *by formula-label,* and perform TAME-appropriate simplifications
(apply_lemma *lemma_name args*)	Introduce the lemma *lemma-name* instantiated with the values *args*
(skolem_in *formula-label skolem-names*)	Skolemize an imbedded quantifier in formula *formula-label* with the names *skolem-names*
(inst_in *formula-label values*)	Instantiate an embedded quantifier in formula *formula-label* with the values *values*
(suppose assertion)	Suppose *assertion* is true, and label it Suppose; then suppose it is false, and label it Suppose not
(partition_proof *comments*)	Split the proof into separate branches for disjuncts in conclusion or inductive-conclusion formula and label branches with *comments*
(deduce assertion)	Attempt to deduce *assertion* with PVS's strategy grind; if successful, label new assertion Deduce
(apply_traj_evolve t)	Compute state time *t* from now
(apply_traj_stop t)	Deduce that the stopping condition cannot hold after time *t* in a trajectory T unless T ends at *t*
(apply_traj_invariant t)	Deduce trajectory invariant holds time *t* from now
(deadline_reason t)	Deduce trajectory cannot evolve more than time *t* if a deadline is reached time *t* from now
(try_simp)	Attempt to complete the current proof branch automatically

```
;;; Proof lemma_4-1 for formula timeout_invariants.lemma_4
;;; developed with shostak decision procedures
(""
 (auto_induct)
 (("1" ;; Base case
   (use_defs "lengthQ"))
  ("2" ;; Case nu_traj(delta_t_action, F_action)
   (apply_specific_precond)
   (apply_traj_evolve "delta_t_action")
   (try_simp))
  ("3" ;; Case send(m_action)
   (suppose "now(prestate) + b >= latest_deadline(queue(prestate))")
   (("1" ;; Suppose now(prestate) + b >= latest_deadline(queue(prestate))
     (use_defs "inductive-conclusion" "nthQ")
     (suppose "n_theorem = 0")
     (("1" ;; Suppose n_theorem = 0
       (try_simp))
      ("2" ;; Suppose not [n_theorem = 0]
       (use_defs "inductive-conclusion" "lengthQ")
       (apply_ind_hyp "n_theorem - 1")
       (try_simp))))
    ("2" ;; Suppose not [now(prestate) + b >= latest_deadline(queue(prestate))]
     (try_simp))))
  ("4" ;; Case receive(m_action)
   (suppose "enQ?(queue(prestate))")
   (("1" ;; Suppose enQ?(queue(prestate))
     (apply_lemma "length_deQ_queue" "queue(prestate)")
     (apply_lemma "nth_elt_deQ_queue" "queue(prestate)" "n_theorem")
     (("1" (try_simp))
      ("2" (try_simp))))
    ("2" ;; Suppose not [enQ?(queue(prestate))]
    (try_simp))))))
```

Fig. 21 TAME proof of `timeout lemma_4`

Appendix B: Lemmas about queues

Although TAME provides steps for proving invariant and simulation properties of timed I/O automata models, it is often necessary to use lemmas from theories about the data types used in specifying the models. TAME does not provide support for proving data type lemmas; however, such lemmas can be accumulated in appropriate theories and placed in the TIOA TAME library `tioa_types_lib` for reuse.

Figure 20 shows the lemmas about queues that were used directly or indirectly in proofs of `timeout` invariants and the forward simulation between `timeout` and `timeout_spec`. The lemmas come from two theories in `tioa_types_lib`: `timed_message_Queue_thy` and `Queue_thy`.

Appendix C: A more complex invariant proof

Figure 21 shows the proof of `timeout lemma_4`, which illustrates the use of several TAME proof steps beyond those used in Figs. 6 and 14: `suppose`, `use_defs`, `apply_ind_hyp`, and `apply_lemma`. Unlike the proofs in Figs. 6 and 14, the proof in Fig. 21 is in nonverbose form. In verbose form, comments describing introduced facts would have been printed after each use of `apply_specific_precond`, `apply_traj_evolve`, `apply_ind_hyp`, and `apply_lemma`.

```
Rule? (apply_specific_precond)
Applying the specific precondition,
this simplifies to:
lemma_4.2 :
;;; Case nu_traj(delta_t_action, F_action)

[-1,(poststate-definition)]
      trans(nu_traj(delta_t_action, F_action), prestate) = poststate
[-2,(pre-state-reachable)]
      reachable(prestate)
[-3,(inductive-hypothesis)]
      n_theorem <= lengthQ(queue(prestate)) - 1 =>
       deadline(nthQ(queue(prestate), n_theorem)) <= now(prestate) + b
{-4,(specific-precondition_part_1 specific-precondition)}
      FORALL (t: (interval(zero, delta_t_action))):
        traj_invariant(nu_traj(delta_t_action, F_action))(F_action(t))
{-5,(specific-precondition_part_2 specific-precondition)}
      FORALL (t: (interval(zero, delta_t_action))):
        traj_stop(nu_traj(delta_t_action, F_action))(F_action(t)) =>
         t = delta_t_action
{-6,(specific-precondition_part_3 specific-precondition)}
      FORALL (t: (interval(zero, delta_t_action))):
        F_action(t) =
         traj_evolve(nu_traj(delta_t_action, F_action))(t, prestate)
[-7,(post-state-reachable)]
      reachable(poststate)
[-8,(inductive-conclusion_part_1 inductive-conclusion)]
      n_theorem <= lengthQ(queue(F_action(delta_t_action))) - 1
  |-------
[1,(inductive-conclusion_part_2 inductive-conclusion)]
      deadline(nthQ(queue(F_action(delta_t_action)), n_theorem)) <=
       now(F_action(delta_t_action)) + b

Rule? (apply_traj_evolve "delta_t_action")
Evolving the trajectory for delta_t_action time,
this simplifies to:
lemma_4.2 :
;;; Case nu_traj(delta_t_action, F_action)

[-1,(poststate-definition)]
      trans(nu_traj(delta_t_action, F_action), prestate) = poststate
[-2,(pre-state-reachable)]
      reachable(prestate)
[-3,(inductive-hypothesis)]
      n_theorem <= lengthQ(queue(prestate)) - 1 =>
       deadline(nthQ(queue(prestate), n_theorem)) <= now(prestate) + b
[-4,(specific-precondition_part_1 specific-precondition)]
      FORALL (t: (interval(zero, delta_t_action))):
        traj_invariant(nu_traj(delta_t_action, F_action))(F_action(t))
[-5,(specific-precondition_part_2 specific-precondition)]
      FORALL (t: (interval(zero, delta_t_action))):
        traj_stop(nu_traj(delta_t_action, F_action))(F_action(t)) =>
         t = delta_t_action
[-6,(post-state-reachable)]
      reachable(poststate)
{-7,(inductive-conclusion_part_1 inductive-conclusion)}
      n_theorem <= lengthQ(prestate`queue) - 1
  |-------
{1,(inductive-conclusion_part_2 inductive-conclusion)}
      deadline(nthQ(prestate`queue, n_theorem)) <=
       dur(delta_t_action) + now(prestate) + b

Rule?
```

Fig. 22 Sample interaction with PVS from proof of `timeout` `lemma_4`

```
1  forward simulation from timeout to timeout_spec:
     ∀ u2: Real
3      (timeout.failed = timeout_spec.failed
        ∧ timeout.suspected = timeout_spec.suspected
5       ∧ timeout.now = timeout_spec.now
        ∧ (if ¬(timeout_spec.failed)
7            then timeout_spec.last_timeout = \infty
             else
9              if enQ_qn(timeout.queue)
               then
11               (timeout_spec.last_timeout)
                   ≥ (latest_deadline(timeout.queue) + u2)
13             else (timeout_spec.last_timeout) ≥ (timeout.t_clock)))
```

Fig. 23 Forward simulation from `timeout` to `timeout_spec`

```
;;; Proof fw_simulation_thm-1 for formula
;;; timeout2timeout_spec.fw_simulation_thm
;;; developed with shostak decision procedures
(""
 (prove_fwd_sim)
 (("1" ;; Base case: User must instantiate with an abstract start state
   (inst "abstract-start-state"
    "(#  last_timeout := infinity,
         now := 0,
         suspected := false,
         failed := false#)")
   (try_simp))
  ("2" ;; Case nu_traj(delta_t_C_action, F_C_action)
    . . . )
  ("3" ;; Case send(m_C_action)
    . . . )
  ("4" ;; Case receive(m_C_action)
    . . . )
  ("5" ;; Case fail
    . . . )
  ("6" ;; Case timeout
    . . . )))
```

Fig. 24 Skeleton of forward simulation proof for `timeout` example

Appendix D: An example interaction with PVS

Figure 22 shows part of the user interaction with the PVS prover during execution of the proof in Fig. 21. The first sequent in Fig. 22 represents the current proof goal in the induction case `nu_traj(delta_t_action, F_action)` after the `(apply_specific_precond)` step in the proof, and the second sequent shows the resulting proof goal after the subsequent `apply_traj_evolve` step. In these sequents, every formula is labeled with a number and one or more names. Curly braces (as opposed to square brackets) around the list of labels for a formula indicate the formula has been changed (or added) by the last proof step. The formulas above the turnstile (i.e., `|----`) are hypotheses, all of which can be assumed known; to discharge the proof goal, one must prove one of the formulas below the turnstile.

```
;;; Proof fw_simulation_thm-13 for formula
;;; TwoTaskRace2TwoTaskRaceSpec.fw_simulation_thm
;;; developed with shostak decision procedures
(""
 (prove_fwd_sim)
 (("1" ;; Base case: User must instantiate with an abstract start state
   (inst "abstract-start-state"
    "(# reported := false,
        now := 0,
        first_report := IF a2 < b1 THEN min(b1, a1) + (b1 - a2) * a1 / a2
                        ELSE a1 ENDIF,
        last_report := fintime(b2 + a2 + b2 * a2 / a1) #)")
   (const_facts)
   . . . )
  ("2" ;; Case nu_traj(delta_t_C_action, F_C_action)
    . . . )
  ("3" ;; Case increment
    . . . )
  ("4" ;; Case decrement
    . . . )
  ("5" ;; Case set
    . . . )
  ("6" ;; Case report
    . . . )))
```

Fig. 25 Skeleton of forward simulation proof for `TwoTaskRace` example

Appendix E: TIOA forward simulation specification for `timeout`

The TIOA specification of the forward simulation relation between `timeout` and `timeout_spec` is shown in Fig. 23.

Appendix F: Structure of saved forward simulation proofs

Figures 24 and 25 show the forward simulation proof skeletons for `timeout` and `TwoTaskRace` after the TAME step `start_fwd_sim_proof` is applied and the instantiation required in the base case is provided. This instantiation must be an abstract start state related to the concrete start state by the simulation relation. For `timeout`, the instantiation is trivially proved correct using `try_simp`. For `TwoTaskRace`, the proof of correctness for the instantiation requires facts about the constants `a1`, `a2`, `b1`, and `b2` followed by additional reasoning about nonlinear real arithmetic.

The proofs of the remaining branches (not shown) appeal to the concrete and abstract preconditions, and apply invariants of the concrete and abstract specifications. They also appeal to lemmas from appropriate supporting theories. For example, the `TwoTaskRace` proof uses lemmas from the theory of real arithmetic, and the `timeout` proof uses lemmas from the theory of queues.

References

1. Alur R (1999) Timed automata. In: Proceedings of the 11th international conference on computer aided verification (CAV '99). Lecture notes in computer science, vol 1633. Springer, Berlin, pp 8–22
2. Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126:183–235
3. Archer M (2000) TAME: Using PVS strategies for special-purpose theorem proving. Ann Math Artif Intell 29(1-4):139–181
4. Archer M (2006) Basing a modeling environment on a general purpose theorem prover. Technical report NRL/MR/5546–06-8952, NRL, Washington, DC, December 2006. Presented at the Monterey Workshop on Software Engineering Tools: Compatibility and Integration, Baden, Austria, October 2004
5. Archer M, Heitmeyer C, Riccobene E (2002) Proving invariants of I/O automata with TAME. Autom Softw Eng 9(3):201–232
6. Archer M, Lim H, Lynch N, Mitra S, Umeno S (2006) Specifying and proving properties of timed I/O automata in the TIOA toolkit. In: Formal methods and models for codesign (MEMOCODE 2006), pp 129–138
7. Bogdanov A, Garland S, Lynch N (2002) 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 international conference, Houston, TX, USA, November 2002, pp 364–368
8. Devillers M (1999) Translating IOA automata to PVS. Technical report CSI-R9903, Computing Science Institute, University of Nijmegen, February 1999
9. Dowek G, Muñoz C, Carreño V (2004) Abstract model of the SATS concept of operations: Initial results and recommendations. Technical report NASA/TM-2004-213006, NASA Langley Research Center, Hampton, VA
10. Garland S (2006) TIOA user guide and reference manual. Technical report, MIT CSAIL, Cambridge, MA. URL http://tioa.csail.mit.edu
11. Garland S, Guttag J (1991). A guide to LP, the Larch prover. Technical report, DEC Systems Research Center. URL http://nms.lcs.mit.edu/Larch/LP
12. Garland S, Lynch N, Tauber J, Viziri M (2004) IOA user guide and reference manual. Technical report MIT-LCS-TR-961, MIT CSAIL, Cambridge, MA
13. Gebremichael B, Vaandrager FW (2005) Specifying urgency in timed I/O automata. In: Proceedings of the 3rd IEEE international conference on software engineering and formal methods (SEFM 2005). Koblenz, Germany, 5–9 September 2005. IEEE Computer Society, Los Alamitos, pp 64–73
14. Griffeth N, Cantor Y, Djouvas C (2006) Testing a network by inferring rerpresentative state machines from network traces. In: Proceedings of the international conference on software engineering advances 2006. IEEE Computer Society, Los Alamitos
15. Heitmeyer C, Archer M, Bharadwaj R, Jeffords R (2005) Tools for constructing requirements specifications: the SCR toolset at the age of ten. Int J Comput Syst Sci Eng 20(1):19–35
16. Kaynar D, Lynch NA, Segala R, Vaandrager F (2003) A mathematical framework for modeling and analyzing real-time systems. In: The 24th IEEE international real-time systems symposium (RTSS), Cancun, Mexico, December 2003
17. Kaynar D, Lynch NA, Segala R, Vaandrager F (2005) The theory of timed I/O automata. In: Synthesis lectures on computer science. Morgan Claypool
18. Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a nutshell. Int J Soft Tools Tech Transf 1(1–2):134–152
19. Lim H (2006) Translating timed I/O automata specifications for theorem proving in PVS. Master's thesis, Mass Inst of Tech, Cambridge, MA. URL http://tioa.csail.mit.edu/
20. Lim H, Archer M (2006) Translation templates to support strategy development in PVS. In: Proceedings of the 6th international workshop on strategies in automated deduction (STRATEGIES06), Seattle, USA, August 2006
21. Lim H, Archer M (2007) Translation templates to support strategy development in PVS. Electron Notes Theor Comput Sci 174(1):59–79
22. Lim H, Kaynar D, Lynch N, Mitra S (2005) Translating timed I/O automata specifications for theorem proving in PVS. In: Formal modeling and analysis of timed systems (FORMATS), Uppsala, Sweden, September 2005, pp 17–31
23. Luchangco V (1996) Personal communication. MTT Computer Science Laboratory
24. Lynch N, Tuttle M (1989) An introduction to input/output automata. CWI-Quarterly 2(3):219–246
25. Lynch N, Vaandrager F (1996) Forward and backward simulations, part II: timing-based systems. Inf Comput 128(1):1–25
26. Lynch NA (1996) Distributed algorithms. Morgan Kaufmann, San Mateo
27. Lynch NA, Garland SJ, Kaynar D, Michel L, Shvartsman A (2007) The Tempo language user guide and reference manual. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, October 2007. URL http://www.veromodo.com/tempo

28. Mavromattis PP (2006) TIOA simulator manual. February 15, 2006. URL http://tioa.csail.mit.edu/public/Tools/simulator/
29. Merritt M, Modugno F, Tuttle MR (1991) Time constrained automata. In: Baeten JCM, Goote JF (eds) CONCUR'91: 2nd international conference on concurrency theory. Lecture notes in computer science, vol 527. Springer, Berlin
30. Mitra S, Archer M (2005) PVS strategies for proving abstraction properties of automata. Electron Notes Theor Comput Sci 152(2):45–65
31. Muñoz C, Carreño V, Dowek G (2006) Formal analysis of the operational concept for the small aircraft transportation system. In: Rigorous engineering of fault-tolerant systems. Lecture notes in computer science, vol 4157. Springer, Berlin, pp 306–325
32. Muñoz C, Dowek G (2005) Hybrid verification of an air traffic operational concept. In: Proceedings of IEEE ISoLA workshop on leveraging applications of formal methods, verification, and validation, Columbia, MD
33. Paulson LC (1994) In: Isabelle: a generic theorem prover. Lecture notes in computer science, vol 828. Springer, Berlin
34. Shankar N, Owre S, Rushby JM, Stringer-Calvert DWJ (2001) PVS Prover Guide, Version 2.4. Technical report, Comp Sci Lab, SRI Int, Menlo Park, CA, November 2001
35. Umeno S (2006) Proving safety properties of an aircraft landing protocol using timed and untimed I/O automata: a case study. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA
36. Umeno S, Lynch N. Proving safety properties of an aircraft landing protocol using I/O automata and the PVS theorem prover: a case study (submitted); long version to appear as an MIT Technical report
37. VeroModo (2006) TIOA model checker user guide and reference manual. 30 August 2006. URL http://www.veromodo.com
38. Win TN (2003) Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2003