Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5546--06-8952

Basing a Modeling Environment on a General Purpose Theorem Prover

Myla Archer

Center for High Assurance Computer Systems
Information Technology Division

December 29, 2006

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 29-12-2006	**2. REPORT TYPE** Memorandum Report	**3. DATES COVERED** *(From - To)*
4. TITLE AND SUBTITLE Basing a Modeling Environment on a General Purpose Theorem Prover		**5a. CONTRACT NUMBER** N0001406WX20708
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 61153N
6. AUTHOR(S) Myla Archer		**5d. PROJECT NUMBER** 55-8956-0-7
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory, 4555 Overlook Avenue, SW, Washington, DC 20375-5320		**8. PERFORMING ORGANIZATION REPORT NUMBER** NRL/MR/5546--06-8952
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research, One Liberty Center, 875 North Randolph Street, Suite 1425, Arlington, VA 22203-1995		**10. SPONSOR / MONITOR'S ACRONYM(S)**
		11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A general purpose theorem prover can be thought of as an extremely flexible modeling environment in which one can define and analyze almost any kind of model. A disadvantage to the full flexibility of a general purpose theorem prover is the lack of any guidance on how to construct a model and how then to apply the theorem prover to analyzing the model. In the general environment supplied by the prover, much time can be consumed in deciding how to specify a model and in interacting with and understanding feedback from the prover. However, specification templates, together with proof strategies whose design follows certain principles, can be used in many general purpose provers to create specialized modeling environments that address these difficulties. A specialized modeling environment created in this way can be further extended and/or further specialized by drawing on the underlying theorem prover for additional capabilities, and provides a means of integrating powerful theorem proving capabilities into existing software development environments by way of appropriate translation schemes. This paper will use TAME (Timed Automata Modeling Environment) to illustrate the creation, extension, and specialization of a modeling environment based on PVS, and its integration into several software development environments.

15. SUBJECT TERMS

Modeling environments	Tool compatibility	Automata models	Verification	Automated deduction
Software engineering tools	Tool integration	Specification	Theorem proving	

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Myla Archer
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UL	24	**19b. TELEPHONE NUMBER** *(include area code)* (202) 404-6304

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

CONTENTS

Basing a Modeling Environment on a General Purpose Theorem Prover ⋆

Myla Archer

Code 5546, Naval Research Laboratory,
Washington, DC 20375
archer@itd.nrl.navy.mil

Abstract. A general purpose theorem prover can be thought of as an extremely flexible modeling environment in which one can define and analyze almost any kind of model. A disadvantage to the full flexibility of a general purpose theorem prover is the lack of any guidance on how to construct a model and how then to apply the theorem prover to analyzing the model. In the general environment supplied by the prover, much time can be consumed in deciding how to specify a model and in interacting with and understanding feedback from the prover. However, specification templates, together with proof strategies whose design follows certain principles, can be used in many general purpose provers to create specialized modeling environments that address these difficulties. A specialized modeling environment created in this way can be further extended and/or further specialized in a straightforward way by drawing on the underlying theorem prover for additional capabilities. Moreover, a specialized modeling environment derived from a general purpose theorem prover provides a means of integrating powerful theorem proving capabilities into existing software development environments by way of appropriate translation schemes. This paper uses TAME (Timed Automata Modeling Environment) to illustrate the creation, extension, and specialization of a modeling environment based on PVS, and its integration into several software development environments.

1 Introduction

For establishing properties of systems, the use of a mechanical theorem prover is often considered impractical because of the effort required to represent a system in the language of the theorem prover and to create proofs of properties of the system using the proof primitives provided in the prover. Attention tends to turn instead to model checking, because of its reputation for being "automatic" and because significant progress has recently been made towards increasing its efficiency, scope, and the degree to which it truly is automatic, through the use of predicate abstraction [12, 5] and abstraction refinement [7, 19]. Even with these advances, however, model checking still does not provide a complete answer to

⋆ This research is funded by ONR.

Manuscript approved November 2, 2006.

the verification problem. Abstraction refinement may still produce large models. Model checking arbitrary properties of concurrent systems or infinite state systems is still problematic. User guidance can still be needed.

Moreover, the use of theorem proving can provide some advantages: 1) infinite state systems can be handled as easily as finite state systems, 2) abstraction is not generally needed to allow verification of properties, and 3) by constructing a proof in a theorem prover, one can both establish the truth of a property and, at the same time, obtain an explanation of *why* a property is true of a system. While the third advantage may not come automatically (due to obscurity in the significance of the proof steps of a mechanical prover), it is possible in many provers to design high level proof steps that advance the proof in a way that can be easily understood. Although model checkers are better than theorem provers at finding counterexamples, when one generates proof explanations along with proofs, the feedback from an incomplete proof in the prover can be meaningful enough to aid in the search for counterexamples, for example in a simulator. The TAME (Timed Automata Modeling Environment) interface to PVS [33] is designed to provide the kind of high level proof steps that allow proof explanation.

A general purpose theorem prover, through its specification language and proof support, constitutes a very general purpose modeling environment by itself. A specialized modeling environment such as TAME that is based on a general purpose theorem prover can "organize" that prover for application to a particular class of models. When the class of models is broad enough (e.g., state machines) to apply in many settings, the specialized modeling environment can be used as an intermediate layer between toolsets designed for particular settings and the general purpose prover. This layer can supply an organization for representing specifications, possibly in other languages, in the language of the prover. Further, it can provide a set of meaningful, user-friendly proof steps that can be built on and extended, if necessary, to provide user-friendly proof support appropriate in other settings.

Using TAME as an example, this paper illustrates how a modeling environment based on a general purpose theorem prover can be used to facilitate the integration of that prover, along with a set of helpful proof steps, into other tools. The paper is organized as follows. Section 2 gives an overview of TAME itself. The next three sections illustrate how TAME is being used to provide user-friendly PVS proof support in three settings. The first setting, described in Section 3, is SCR (Software Cost Reduction), a tabular specification method particularly suited to specifying requirements for control systems, initially developed at the Naval Research Laboratory (NRL) [17,18]. SCR is supported by a toolset developed and maintained by NRL. Section 4 describes the second setting: TIOA (Timed I/O Automata), an automaton model [21] for which a specification language and toolkit initially developed at MIT [20] is currently under commercial development. Finally, Section 5 describes the third setting: policy analysis for SELinux (Security-Enhanced Linux) [22,34], a version of Linux initially developed at NSA that provides a policy definition language and enforcement mechanism intended to allow (and encourage) Linux application developers to incorporate certain security guarantees in their software. Section 6 briefly discusses related work, and Section 7 provides some conclusions and a preview of future work.

2 TAME

As noted in the introduction, the modeling environment TAME [1, 3] is based on the general purpose theorem prover PVS. TAME specifications represent automata in the MMT timed automaton model [28], which extends the untimed I/O automata model [25] to include timed I/O automata (see also [26, 27]). The I/O automata model is a very general model in which many other automaton models can be represented.

The motivation for TAME is the desire to facilitate proving properties of automata using PVS, and, at the same time to make the resulting proofs saved by PVS human-understandable. Towards accomplishing this goal, TAME provides a set of proof steps that resemble high-level reasoning steps used by humans. Figure 1 shows the proof steps TAME currently supplies for proving state invariant properties of automata interactively. As noted in [2], the steps in Figure 1, when

TAME Strategy	Purpose
AUTO_INDUCT	Set up a reachable-state induction proof
DIRECT_PROOF	Set up a non-induction proof
DIRECT_INDUCTION	Set up a mathematical induction proof
APPLY_SPECIFIC_PRECOND	Introduce the specified precondition
APPLY_GENERAL_PRECOND	Introduce the timing constraints
APPLY_IND_HYP	Apply the inductive hypothesis
APPLY_INV_LEMMA	Apply an invariant lemma
APPLY_LEMMA	Apply any general lemma
SUPPOSE	Do a case split and label the cases
COMPUTE_POSTSTATE	Compute the poststate of the current transition
SKOLEM_IN	Skolemize an embedded quantified formula
INST_IN	Instantiate an embedded quantified formula
TRY_SIMP	Try to complete the proof automatically

Fig. 1. TAME proof steps for invariants.

supplemented by a small number of PVS proof steps including EXPAND (which expands definitions) and INST (which instantiates formulas), has turned out to be sufficient for proving the invariants in the great majority of examples to which TAME has been applied.

TAME steps are provided with names intended to indicate their significance. The steps also save information in the form of comments used to identify the meanings of the various proof branches. These features help to support natural language translations of saved proofs. Figure 2 shows an example (tree structured) saved TAME proof and its automatically generated natural language translation, reproduced from [1]. The natural language proof in Figure 2 can be understood without reference to the PVS theorem prover—one only needs access to the automaton specification and the statement of the theorem being proved.

To make translations such as that in Figure 2 possible, it is necessary for every proof step in the saved PVS proof to have a meaning which depends only on elements in the specification of an automaton and its properties, and not on any details of proof execution. Thus a major desire for TAME is to provide a set of proof steps that are execution-independent semantically while being sufficient to prove all automaton properties of interest.

The need to instantiate formulas is a challenge to the desire for an execution-independent semantics. The PVS step (INST?), which instantiates an unspecified formula (chosen by PVS) with an unspecified value (also chosen by PVS) is an example of a semantically non-execution-independent proof step: its effect can be changed by an appropriate PVS (HIDE <*formula-number*>) step that hides a quantified formula that might otherwise have been instantiated. The PVS step (INST -2 0), whose meaning (to instantiate formula numbered -2 in the current proof goal with the value 0) in a given proof goal is more specific, still has the same problem. In TAME, execution-independent semantics for proof steps using (e.g.) INST is more closely approximated by maintaining labels on formulas that indicate their significance, as well as labels (comments) on subgoals that track the current proof branch. Thus, in the induction branch of an invariant proof corresponding to an action <*action*>, the meaning of (INST `"specific-precondition_part_1"` <*value*>) is "instantiate the first conjunct of the specific precondition of <*action*> with <*value*>" on its first use in any proof branch, and the meanings of its later uses in that proof branch, if any, will also be clear from the form of <*action*>'s specific precondition. When possible, TAME proof steps (e.g., **APPLY_INV_LEMMA**, **APPLY_IND_HYP**) remove any ambiguities related to instantiation by permitting instantiations to be provided as arguments.

Recent additions to the set of TAME proof steps include **PROVE_REFINEMENT** and **PROVE_FWD_SIM**, which perform the initial steps in the proofs of refinement and forward simulation, two abstraction properties that can relate pairs of automata (see [29, 30]). The strategy **PROVE_REFINEMENT**, in combination with the proof steps listed in Figure 1, is sufficient for the refinement proofs attempted so far. Research is continuing into additional TAME steps for completing proofs of forward simulation and more complex refinement properties.

In addition to (PVS) proof strategies for implementing proof steps, TAME's implementation uses a set of templates and a set of supporting theories. TAME has templates for specifying systems (e.g., the automaton specification), for specifying properties (e.g., the abstraction property specification template), and for stating lemmas and theorems (e.g., invariant property lemmas and abstraction property lemmas). The supporting theories in TAME serve to provide defini-

```
Inv_5(s:states): bool = (FORALL (e:Edges): length(mq(e,s)) <= 1);
    (""
    (AUTO_INDUCT)
    (("1" ;;Case add_child(addE_action)
          (APPLY_SPECIFIC_PRECOND)
          (SUPPOSE "e_theorem = addE_action")
          (("1.1" ;;Suppose e_theorem = addE_action
                (TRY_SIMP))
           ("1.2" ;;Suppose not [e_theorem = addE_action]
                (TRY_SIMP))))
     ("2" ;;Case children_known(childV_action)
          (SUPPOSE "source(e_theorem) = childV_action")
          (("2.1" ;;Suppose source(e_theorem) = childV_action
                (APPLY_SPECIFIC_PRECOND)
                (APPLY_INV_LEMMA "2" "e_theorem")
                (TRY_SIMP))
           ("2.2" ;;Suppose not [source(e_theorem) = childV_action]
                (TRY_SIMP))))
     ("3" ;;Case ack(ackE_action)
          (SUPPOSE "e_theorem = ackE_action")
          (("3.1" ;;Suppose e_theorem = ackE_action
                (APPLY_SPECIFIC_PRECOND)
                (TRY_SIMP))
           ("3.2" ;;Suppose not [e_theorem = ackE_action]
                (TRY_SIMP))))))
```

Proof. The proof is by induction. The base case is trivial. There are 3 nontrivial action cases.

• **Consider the action** `add_child(addE_action)`**.** The proof in this case is as follows. Apply the precondition of the action `add_child(addE_action)`. Suppose first that `e_theorem` = `addE_action`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `e_theorem` = `addE_action`. The rest of the proof in this case is obvious. This completes the proof for the action `add_child(addE_action)`.

• **Consider the action** `children_known(childV_action)`**.** The proof in this case is as follows. Suppose first that `source(e_theorem)` = `childV_action`. Apply the precondition of the action `children_known(childV_action)`. Apply Invariant *2* to `e_theorem`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `source(e_theorem)` = `childV_action`. The rest of the proof in this case is obvious. This completes the proof for the action `children_known(childV_action)`.

• **Consider the action** `ack(ackE_action)`**.** The proof in this case is as follows. Suppose first that `e_theorem` = `ackE_action`. Apply the precondition of the action `ack(ackE_action)`. The rest of the proof in this case is obvious. Suppose, on the other hand, that it is not true that `e_theorem` = `ackE_action`. The rest of the proof in this case is obvious. This completes the proof for the action `ack(ackE_action)`. □

Fig. 2. A saved TAME proof and its translation.

tions, capture common reasoning steps (such as reachable-state induction), serve as theory parameter "types" (e.g., the theory `automaton` used as a parameter type in the specification theory for the refinement property), and to support reasoning about data of types commonly used in a domain (e.g., communication protocols).

Central to the integration of TAME into other development environments is its automaton template to which specifications must be matched. The major ingredients of this template can be summarized as follows:

- `MMTstates`: A product type whose elements are vectors of values for the non-time-related variables in a system, i.e., elements of the "basic state";
- `actions`: A PVS `DATATYPE` whose elements are a set of (parameterized) actions that trigger the system's state transitions;
- `trans(a:actions,s:states):states` : The transition function that computes the effect of action `a` on state `s`;
- `enabled(a:actions,s:states):bool` : The full precondition for action `a` on state `s`, decomposed into four parts:

  ```
  enabled(a,s) = enabled_general(a,s) & enabled_specific(a,s) &
                 OKstate?(trans(a,s)) & OKtrans?(s,trans(a,s))
  ```

  where these four parts are:
  - `enabled_specific(a:actions,s:states):bool` : The specific precondition for action `a` on state `s`;
  - `enabled_general(a:actions,s:states):bool` : The general timing precondition for action `a` on state `s`, whose definition is system-independent;
  - `OKstate?(s:states):bool` : A predicate limiting the allowable post-states in the system (i.e., very close to a state invariant by fiat);
  - `OKtrans?(s,s_new:states):bool` : A predicate limiting the allowable state transitions in the system (i.e., a transition invariant by fiat);
- `start`: A predicate defining the possible start states of the system.

In any specification that follows this template, `start(s) OR OKstate?(s)` will be a state invariant. Not requiring that `start(s)` $\Rightarrow$ `OKstate?(s)` allows one to model systems whose initial state may be unpredictable.

3 Example: SCR

3.1 Specifications in SCR

SCR (Software Cost Reduction) is a tabular requirements specification method first developed at NRL (Naval Research Laboratory). NRL's SCR toolset provides tools for creating and analyzing requirements specifications based on the SCR method. The toolset has been extensively documented (see, e.g., [16, 15, 14]), and has a growing number of users in industry, government, and academia.

Every SCR specification defines a state machine in which a state is determined by an assignment of values to state variables. We will refer to the state machine defined by an SCR specification as an *SCR machine*. The state variables fall into four classes: *monitored* variables, representing inputs, *controlled* variables, representing outputs, *mode classes*, natural for representing the current mode (or modes) of operation of the machine, and *terms*, which can be used to capture certain interesting state information concisely, and which can also, along with mode classes, be used to capture history information. The controlled variables, mode classes, and terms comprise the *dependent variables.* Transitions in an SCR machine are triggered by an *input event*, a change to a monitored variable. The *one input assumption* ensures that only one monitored variable at a time can change. When an input event occurs, the dependent variables are updated in some order consistent with the "new state dependencies" order that can be deduced from their definitions. The SCR toolset's *consistency checker* [16] checks that there are no circularities in this dependence.

An SCR specification consists of a set of tables. Special tables called *dictionaries* list variables, types, environmental assumptions, assertions to be verified, and so on. The values assigned to the dependent variables on a state transition are computed from tables (or in simple cases, expressions), one for each variable. The SCR toolset supports three types of variable definition tables (see Figure 3): *mode transition* tables, which are used to define mode classes, and *condition* tables and *event* tables, which are used to define the other dependent variables (terms and controlled variables).

Mode Transition Table		
$\texttt{mode}_1$	$\texttt{event}_1$	$\texttt{mode}_2$
. . .	. . .	. . .
$\texttt{mode}_1$	$\texttt{event}_i$	$\texttt{mode}_k$
. . .	. . .	. . .
$\texttt{mode}_n$	$\texttt{event}_j$	$\texttt{mode}_m$

Condition Table			
$\texttt{mode}_1$	$\texttt{cond}_{1,1}$	. . .	$\texttt{cond}_{1,m}$
. . .	. . .	. . .	. . .
$\texttt{mode}_n$	$\texttt{cond}_{n,1}$	. . .	$\texttt{cond}_{n,m}$
var =	$\texttt{val}_1$	. . .	$\texttt{val}_m$

Event Table			
$\texttt{mode}_1$	$\texttt{event}_{1,1}$	. . .	$\texttt{event}_{1,m}$
. . .	. . .	. . .	. . .
$\texttt{mode}_1$	$\texttt{event}_{n,1}$	. . .	$\texttt{event}_{n,m}$
var′ =	$\texttt{val}_1$	. . .	$\texttt{val}_m$

Fig. 3. SCR's three variable definition table types.

As illustrated below in Figure 5 on page 10, the use in Figure 3 of modes $\texttt{mode}_i$ to index the rows of condition and event tables is flexible: e.g., one can list several modes in one row, if desired; one can also omit them in the table altogether. The $\texttt{cond}_{i,j}$ entries in the condition table are boolean expressions involving the variables of one state; the $\texttt{event}_i$ and $\texttt{event}_{i,j}$ entries in the event table are boolean expressions representing *events.* Events in SCR can be defined using one of the constructs `@T`, `@F`, or `@C`. Following the usual convention of using unprimed names to represent values in the old state of a transition, and primed names for new state values, these constructs can be defined by:

$$\begin{aligned}
\texttt{@T(c)} &\iff \neg\texttt{c} \land \texttt{c}' \\
\texttt{@F(c)} &\iff \texttt{c} \land \neg\texttt{c}' \\
\texttt{@C(c)} &\iff \texttt{c}' \neq \texttt{c}.
\end{aligned}$$

Using `WHEN`, events defined by `@T`, `@F`, and `@C` can also be *conditioned*:

$$\texttt{@T(c) WHEN d} \iff \neg\texttt{c} \land \texttt{c}' \land \texttt{d}. \qquad (*)$$

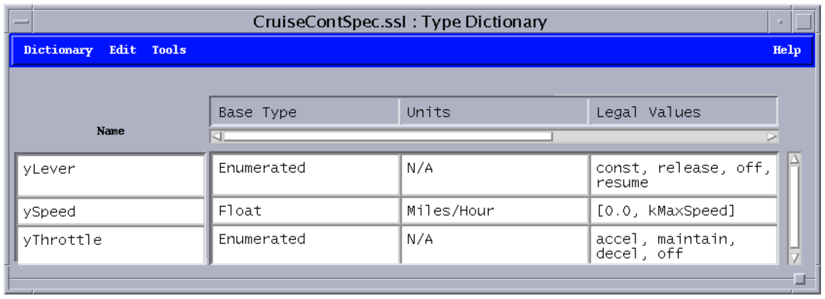
Figures 4 and 5 show example dictionaries and tables from the SCR specification of a cruise control system (CCS) that is used as a running example in [14]. Figure 6 shows the variable dependency graph for CCS. In the SCR tool set, the dependency provides perhaps the best unified overview of an SCR specification—for example, the definition table of any dependent variable can be brought up with an appropriate mouse click. The variable dependencies are computed during consistency checking in the SCR tool set. Any given variable can depend directly on the old state value or the new state value of another variable (or both); thus, dependencies can be identified as either *old state dependencies* or *new state dependencies*. The SCR consistency checker checks that there is no circularity in the new state dependencies.

An additional type of dependency is an *update dependency*, where it is possible for a change in the value of a variable to be triggered as the result of a change in the variable on which it depends; the update dependencies form a subset of the new state dependencies. An example of a new state dependency that is *not* an update dependency can be seen in the event table for `tDesiredSpeed` (Figure 5): `tDesiredSpeed` has a new state dependency on `mSpeed` (since its new value is copied from the new value of `mSpeed`); however, a change in `mSpeed` will not *cause* a change in `tDesiredSpeed`. In fact, as the `tDesiredSpeed` event table shows, the only variable on which `tDesiredSpeed` has a direct update dependency is the monitored variable `mLever`. As the dependency graph in Figure 6 shows, `mLever` has no further dependencies, and is therefore the unique variable on which `mSpeed` has an update dependency.

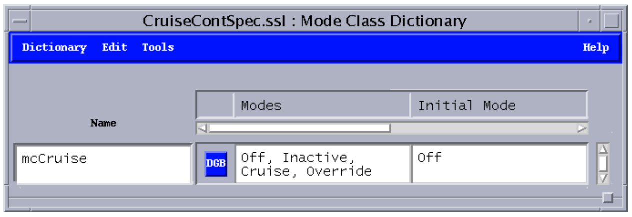
Note that because of the one input assumption, a change in `mLever` will never cause a change in `mSpeed`, so that when a change in `tDesiredSpeed` is triggered by a change in `mLever`, the value of $\texttt{mSpeed}'$ is the same as that of `mSpeed`. Thus, replacing $\texttt{mSpeed}'$ by `mSpeed` in the table for `tDesiredSpeed` would yield an equivalent specification. However, non-trivial update dependencies can arise when the condition `d` in some conditioned event of the form $(*)$ above uses a new state value not used in the condition `c`.

3.2 Representing SCR specifications in TAME

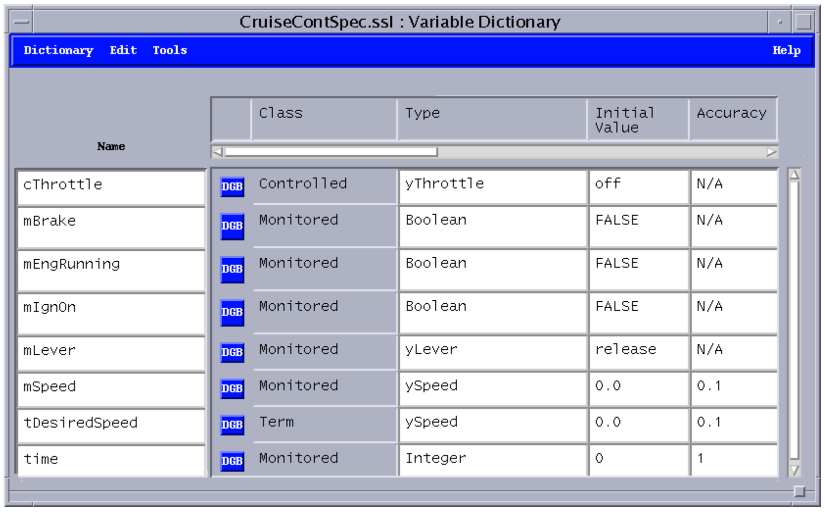
Representing an SCR machine as an I/O automaton mathematically is straightforward: there is one action for each monitored variable, all the actions are input actions, and the transitions triggered by actions update all the dependent variables. In SCR, this updating is done by passing through the dependent variables



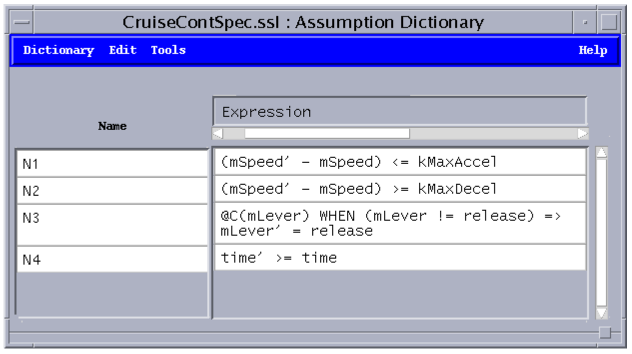
Name	Base Type	Units	Legal Values
yLever	Enumerated	N/A	const, release, off, resume
ySpeed	Float	Miles/Hour	[0.0, kMaxSpeed]
yThrottle	Enumerated	N/A	accel, maintain, decel, off



Name		Modes	Initial Mode
mcCruise	DGB	Off, Inactive, Cruise, Override	Off

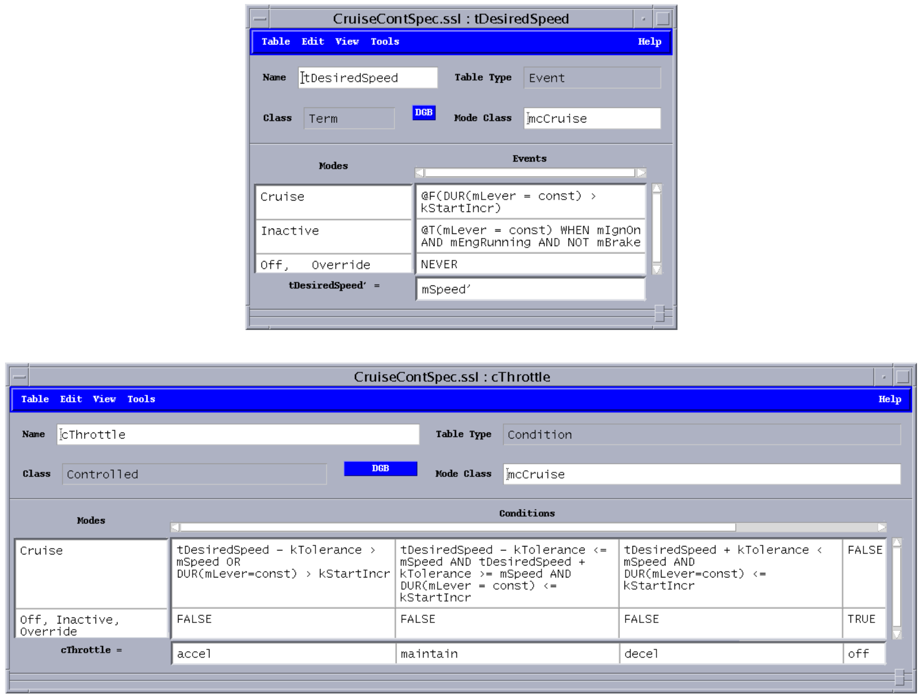


Name		Class	Type	Initial Value	Accuracy
cThrottle	DGB	Controlled	yThrottle	off	N/A
mBrake	DGB	Monitored	Boolean	FALSE	N/A
mEngRunning	DGB	Monitored	Boolean	FALSE	N/A
mIgnOn	DGB	Monitored	Boolean	FALSE	N/A
mLever	DGB	Monitored	yLever	release	N/A
mSpeed	DGB	Monitored	ySpeed	0.0	0.1
tDesiredSpeed	DGB	Term	ySpeed	0.0	0.1
time	DGB	Monitored	Integer	0	1



Name	Expression
N1	(mSpeed' - mSpeed) <= kMaxAccel
N2	(mSpeed' - mSpeed) >= kMaxDecel
N3	@C(mLever) WHEN (mLever != release) => mLever' = release
N4	time' >= time

Fig. 4. The type, mode class, variable, and assumption dictionaries for CCS.



CruiseContSpec.ssl : tDesiredSpeed

Table Edit View Tools Help

Name	tDesiredSpeed	Table Type	Event
Class	Term	Mode Class	mcCruise

Modes	Events
Cruise	@F(DUR(mLever = const) > kStartIncr)
Inactive	@T(mLever = const) WHEN mIgnOn AND mEngRunning AND NOT mBrake
Off, Override	NEVER
tDesiredSpeed' =	mSpeed'

CruiseContSpec.ssl : cThrottle

Table Edit View Tools Help

Name	cThrottle	Table Type	Condition
Class	Controlled	Mode Class	mcCruise

Modes	Conditions			
Cruise	tDesiredSpeed - kTolerance > mSpeed OR DUR(mLever=const) > kStartIncr	tDesiredSpeed - kTolerance <= mSpeed AND tDesiredSpeed + kTolerance >= mSpeed AND DUR(mLever = const) <= kStartIncr	tDesiredSpeed + kTolerance < mSpeed AND DUR(mLever=const) <= kStartIncr	FALSE
Off, Inactive, Override	FALSE	FALSE	FALSE	TRUE
cThrottle =	accel	maintain	decel	off

Fig. 5. The `tDesiredSpeed` event table and `cThrottle` condition table for CCS.

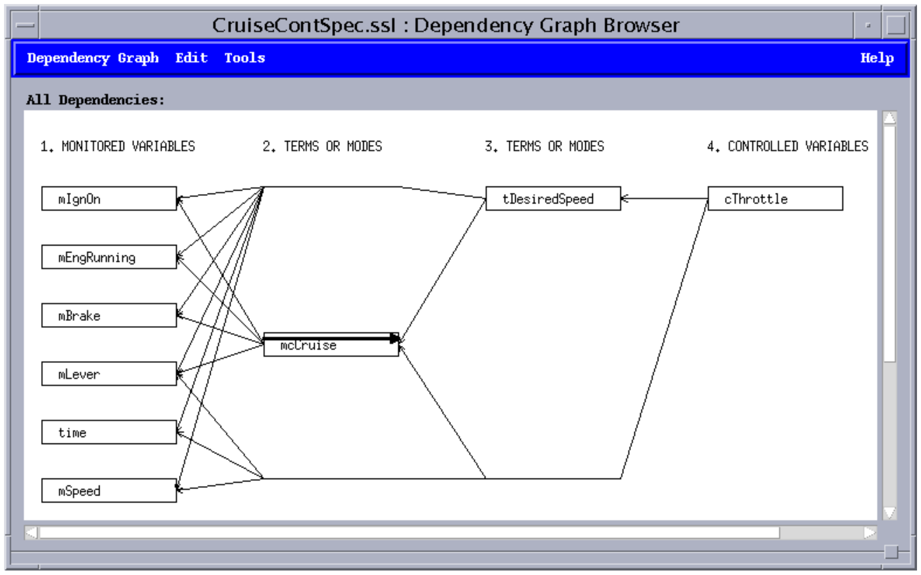


Fig. 6. The variable dependency graph for CCS.

in an order consistent with the new state dependencies. One choice of representing the transitions concretely in TAME is to mimic this pass through the variables using a nesting of LET constructs that is as deep as the dependency graph. This approach allows the new state s′ to be computed as a function of old state s and the input action a. However, experience has shown that this is very inefficient for theorem proving (at least in PVS). A more efficient choice is to represent the transitions as a *relation* trans(s,a,s′):bool. The SCR-to-TAME translator in the SCR toolset uses this representation.

A fragment of the TAME representation of the transition relation for CCS is shown in Figure 7. This fragment, which shows the transition triggered by the

```
trans(s : states, A: actions, new_s : states): bool =
  new_s =
    CASES A OF
      ...
      mSpeed(mSpeed_value):
        s WITH [basic := basic(s) WITH
          [mSpeed_part := mSpeed_value,
           cThrottle_part := update_cThrottle(new_s)]],
      ...
    ENDCASES
```

Fig. 7. Transition relation for CCS in TAME.

input event when mSpeed acquires a new value mSpeed_value, illustrates the use of update dependency information by the SCR-to-TAME translator: although tDesiredSpeed has a new state dependency on mSpeed, it need not be updated as a result of a change in mSpeed. However, mSpeed is updated to its new value, and cThrottle, which *is* update dependent on mSpeed, is updated in accordance with its (condition) table, represented by the function update_cThrottle.

The representation of variable value tables in TAME takes advantage of the disjointness checks performed by the consistency checker, and employs some simple optimizations. Figure 8 shows the definition in TAME of update_cThrottle. Each i-th guard cThrottle_discr_0_i in the IF construct represents the expression in the i-th cell of row 0 of the table (where numbering starts from 0). The fact that consistency checking in SCR showed these guards to be disjoint guarantees that the IF expression captures all possible behaviors of cThrottle in the SCR semantics (which does allow nondeterminism). Because the expressions in the second row of the table are all boolean constants, the SCR-to-TAME translator does not construct any cThrottle_discr_1_i functions. Instead, it pre-simplifies the IF expressions that would have corresponded to the modes Inactive, Off, and Override.

```
update_cThrottle(s : states) : yThrottle =
  CASES mcCruise(s) OF
    Cruise:
      IF cThrottle_discr_0_0(s) THEN accel
      ELSIF cThrottle_discr_0_1(s) THEN maintain
      ELSIF cThrottle_discr_0_2(s) THEN decel
      ELSE cThrottle(s)
      ENDIF,
    Inactive:
      off,
    Off:
      off,
    Override:
      off
  ENDCASES
```

Fig. 8. TAME representation of the `cThrottle` condition table.

The preconditions on actions come from the assumption dictionary, which contains assumptions, in the form of predicates on the monitored and controlled variables, about the environment in which the SCR machine will operate. These assumptions can be categorized according to their mixes of old state and new state values. All of the assumptions in the assumption dictionary in Figure 4 involve the new state value of only a single monitored variable. Therefore, each of them translates naturally into a component of the definition of the specific precondition `enabled_specific(a,s)` in the case when `a` is the action which sets the monitored variable to new value. If all variable values in an assumption are either old state or new state values, then that assumption translates to a conjunct in the predicate `OKstate?(s)`. All other assumptions are gathered into the relation `OKtrans?(s,s')`. Because `trans` was recast as a relation for SCR, the `OKtrans?` conjunct of the full precondition `enabled` for action `a` on state `s` cannot be rendered as `OKtrans?(s,trans(a,s))`. Instead, `enabled` must also be recast as a relation `enabled(s,a,s')`:`bool`.

3.3 TAME proof support in SCR

Because liveness properties are irrelevant for SCR machines, which are input driven, safety properties are the properties of most interest for SCR specifications. In practice, the most important of these are *state invariants*—properties that hold for every reachable state—and *transition invariants*—properties that hold for every reachable transition. Currently, TAME supplies two major strategies that can be used in verifying state and transition invariants of SCR specifications: **SCR_INV_PROOF** and **ANALYZE**. The implementation of **SCR_INV_PROOF** builds on the existing TAME steps **AUTO_INDUCT**, **DIRECT_PROOF**, **APPLY_SPECIFIC_PRECOND**, and **APPLY_INV_LEMMA**. Based on the first three of these TAME steps, **SCR_INV_PROOF** first attempts an automatic proof of the invariant. If the proof does not succeed,

SCR_INV_PROOF then uses **APPLY_INV_LEMMA** to find, for each unproved subgoal, a minimal combination of the known state invariants, which include invariants generated automatically in the SCR toolset, that will complete the proof (if one exists).

To some extent, the user can provide guidance in the verification process. An optional, natural number argument to **SCR_INV_PROOF** allows the user to limit the size of invariant combinations explored on each unproved subgoal. Any unproved subgoals that remain correspond to some set of prestate-poststate pairs that cannot be shown automatically to preserve the state invariant inductively, or satisfy the transition invariant (as the case may be). For any unproved subgoal, the strategy **ANALYZE** can be used to make explicit all known information about the relevant set of prestate-poststate pairs. This set can guide the search for counterexamples, or suggest additional invariants that may be needed as lemmas.

4 Example: TIOA

The I/O automata model [25] is a very general automata model particularly suited for the specification of distributed systems and algorithms. The IOA toolset developed at MIT [10, 9] supports the specification, simulation, and verification of I/O automata models of systems. The TIOA (Timed I/O Automata) toolkit currently being developed at MIT [20] is an extension of the IOA toolset to include the timed I/O automata model described in [21]. TAME is being integrated into the TIOA toolkit to provide theorem proving support.

The timed automata model in [21] differs in two respects from the MMT timed automata model that has until now been supported in TAME: 1) in place of *first* and *last* time bounds constraining when an action can occur, the timeliness of actions is now ensured by either *stopping conditions* (as in [21]) or by *urgency predicates* (as in [11]) on the current state `s`, and 2) changes in the state during time passage between discrete actions is now defined in terms of *trajectories*, which are continuous or piecewise continuous—and possibly nondeterministic—paths through the state space describing how the state evolves with time. Associating an urgency predicate `P` with an action `a` ensures that when `P(s)` is true, time cannot advance until some action (possibly, but not necessarily `a`) causes `P(s)` to become false. Trajectories can be defined by providing a *state model*, an (optional) *stopping condition*, and an (optional) *trajectory invariant*. A state model can be given in terms of algebraic and differential equations and inclusions.

Very few adaptations are required in TAME to fit it to this new timed automata model. Although trajectories and transitions are described as separate entities in TIOA specifications, all trajectories that have arisen so far in practice can be handled by representing them as actions parameterized by the amount `delta_t` of time passage and some possibly nondeterministic function `F` describing the trajectory. The higher order nature of PVS makes this kind of conflation of trajectories with transitions possible; this is a convenience because when this is done, no change to the TAME template is needed to handle trajectories.

For the MMT model, TAME represents the first and last time bounds on actions by a pair of functions first and last of type [[actions,states]->time]. The precondition of the time passage action ν(delta_t) checks that now+delta_t is less than the last deadline of any action; the effect of the action is to increase te value of now by an increment delta_t. One way the TIOA model differs from the MMT model is that it allows for multiple time passage actions, one for each trajectory. The effect on the current time of each of these actions on now is still to increase its value by the increment delta_t. However, the other state variables can also change with time, in the manner described in the trajectory's state model. In TAME, the state model of a trajectory is captured as the function F (of type [time->states]) that, along with the time increment delta_t, is passed as a parameter to corresponding time passage action a. The limits on any nondeterministic behavior of F are captured in the precondition of the action a(delta_t,F). The state resulting from the action can then be represented simply as F(delta_t). This approach to capturing the possibly nondeterministic nature of trajectories is an adaptation of the method (suggested by V. Luchangco [24]) of specifying an action with a nondeterministic result by using the nondeterministic parts of its result as parameters to the action.

The parts of the preconditions of time passage actions that capture urgency predicates or stopping conditions are somewhat similar to those previously used to enforce first and last time bounds. In particular, urgency predicates can be represented using a function urgent of type [[actions,states]->bool], and the precondition of any action involving passage of delta_t time must check that urgent(a,s) is false for all discrete actions a and for all states s = F(t) on the trajectory F for all times t between now and now+delta_t. The analog for stopping conditions is similar but slightly simpler, because it does not require quantification over actions; in particular stopping conditions can be represented by a function stop of type [[actions,states]->bool], and the precondition of any action a involving passage of delta_t time must check that stop(a,s) is false for all states s = F(t) on the trajectory F for all times t after now and before (but not including) now+delta_t.

A TIOA example that includes trajectories is provided in Figure 9, which shows the representation as a TIOA specification of a simple train braking system from [35]. A possible TAME representation for the preconditions and effects of brakeOn and the trajectory braking is shown in Figure 10. This representation shows how the nondeterminism of the state model parameter F of the (trajectory) action braking defined by the evolve clause for braking in Figure 9 can be constrained in TAME using the precondition enabled. Note that a parameter (c) and the enabled predicate are similarly used to handle the nondeterminism in the action brakeOn in Figure 9. The TAME representation in Figure 10 also illustrates how the trajectory invariant for braking and the urgency predicate (which is trivial in this example) are incorporated as parts of the precondition.

While the effects of actions represented in the definition of trans in Figure 10 are quite simple, for other TIOA specifications, they can be more complex. In particular, it is possible to have variable dependencies in TIOA specifications

```
automaton TRAIN(c_s, c1_s, c2_s, c2_min, c2_max: Real)
signature
  input brakeOn, brakeOff
states
  x: Real := c_s,
  x1: Real := c1_s,
  x2: Real := c2_s,
  b: Bool := false,
  now: Real := 0
transitions
  input brakeOn
    eff
      b := true;
      x2 := choose c where (c >= c2_min /\ c <= c2_max)
  input brakeOff
    eff
      b := false;
      x2 := 0
trajectories
  trajdef braking
    invariant b
    evolve
      d(now) = 1;
      d(x) = x1;
      d(x1) = x2;
      c2_min <= x2;
      x2 <= c2_max
  trajdef not_braking
    invariant ~b
    evolve
      d(now) = 1;
      d(x) = x1;
      d(x1) = x2;
      x2 = 0
```

Fig. 9. A TIOA specification for a train braking system.

of the effects of actions analogous to the variable dependencies in SCR specifications. In such cases, there is a choice to be made between the use of nested LET constructs and symbolic computation of expressions representing values that depend on values assigned to "earlier" variables in the same transition. The situation in TIOA is slightly different from that in SCR, in which the specification structure lends itself to lazy evaluation of the new state. Which is the better choice in TIOA from the standpoint of efficiency in theorem proving remains to be determined.

The existing theorem proving support in TAME for establishing invariant and abstraction properties of automata can be carried over to TIOA with only minimal additions to handle any reasoning about urgency predicates or stopping conditions. Additional TAME-style proof support to simplify reasoning about the evolution of trajectories is likely to be needed, because this reasoning can involve nonlinear arithmetic, reasoning about trigonometric functions, etc. It is

```
enabled(a:actions, s:states):bool =
  CASES a OF
    brakeon(c): c >= c2_min AND c <= c2_max

    braking(delta_t,F):
      FORALL(t:time): zero <= t AND t <= delta_t =>

        (FORALL(a:actions): t < delta_t => NOT(urgent(a,F(t)))) AND

        b(F(t) AND

        c2_min <= x2(F(t)) AND x2(F(t)) <= c2_max AND

        (FORALL(t0:time): zero <= t0 AND t0 <= t =>
               c2_min*(t-t0) <= x1(F(t))-x1(F(t0)) AND
               x1(F(t))-x1(F(t0)) <= c2_max*(t-t0)) AND

        (FORALL(t0:time): zero <= t0 AND t0 <= t =>
               c2_min*(t-t0)*(t-t0)/2 <= x(F(t))-x(F(t0)) AND
               x(F(t))-x(F(t0)) <= c2_max*(t-t0)*(t-t0)/2) AND

        now(F(t)) = now(s) + t,
     ...
   ENDCASES

trans(a:actions, s:states):states =
  CASES a OF
    brakeon(c): s WITH [b:=true, x2:=c],

    braking(delta_t, F): F(delta_t)
    ...
  ENDCASES
```

Fig. 10. Part of a possible PVS representation of the TIOA specification in Figure 9.

anticipated that part of the proof support for reasoning about arithmetic can be based on existing packages developed at NASA Langley: Field [31] and Manip [8].

5 Example: SELinux

A different approach to the use of TAME to support modeling and verification of systems in PVS has been used in NRL's tool for modeling and analyzing SELinux security policies [4]. In this application, the specifications to be modeled are security policies represented in the SELinux policy definition language [22, 34]. Policies in this language are defined by a set of rules such as type enforcement (TE) `allow` rules of the form:

```
allow <type_s> <type_t>:<obj_class> <perm>
```

and `type_transition` rules of the form:

```
type_transition <type_s> <type_t>:<obj_class> <type_n>.
```

In these rules, the names `<type_s>`, `<type_t>`, and `<type_n>` are chosen to suggest the roles of the TE types for which they are placeholders: a source type, a target type, and (for `type_transition`) a new type. The form of the `allow` rules suggests that an object of the source type is given permission `<perm>` to any object of class `<obj_class>` of the target type, while the form of the `type_enforcement` rules suggests that an object of the source type can interact with an object of the target type to produce a new object of class `<obj_class>` that is assigned the new type. However, the actual semantics of any given rule cannot be defined independently from the code; the exact meaning of the rule is determined by the way policy checks of the rule are used in the code.

Because the effects of policies are determined by explicit reference to rules in the system code, obtaining an accurate representation of the effect of a policy requires modeling the system at some appropriate level of abstraction. The system can be modeled in TAME by choosing to represent the state in terms of the set of objects (processes, files, file descriptors, etc.) currently present in the system, and the set of actions in terms of the available system calls needed to allow one object in the system to affect another. Rules in the policy affect both the preconditions and effects of actions; for example, `allow` rules are used to determine whether an action is enabled, and `type_transition` rules affect the type of the result of an action. However, the effects of the rules can be abstracted in such a way that it is possible to use a standard TAME template instantiation in modeling arbitrary policies.

This instantiation uses a standard set of names for the types, functions, and predicates that represent the policy-specific details of an SELinux policy specification. For example, the enumerated types `TE_type`, `Permission`, and `Objectclass` represent the sets of TE types, permissions, and object classes defined in the policy; the function `GetTE_type` returns the TE type of a system object such as a process or a file, and the predicate `Allowed`, of type `[[TE_type,TE_type,Objectclass,Permission] -> bool]`, captures the policy's set of `allow` rules. Because actions (i.e., system calls) in the model are parameterized by the process making the system call, it is possible to capture the embedding of rules in the code as a predicate `PermissionGranted`, of type `[[actions,states] -> bool]`.

In this situation, it is not necessary to fit each policy to the TAME template; it is sufficient to generate appropriate supporting theories and strategies. In particular, to model any given policy, one can simply compile the policy as defined in the SELinux policy specification language into a set of PVS theories containing the definitions for each of the standard names. In addition, a few policy-dependent special purpose strategies are generated when a policy is compiled: these strategies simply expand the definitions of sets of elements (such as permissions). These compiled strategies are used to support a fixed set of special purpose strategies that has been developed to simplify type checking SELinux policy specifications and to check the consistency of the `neverallow` assertions in a policy with the generated predicate `Allowed`. The standard set of TAME strategies described in Section 2 is available for proving deeper policy properties.

The initial policy modeled in TAME is a subset of the policy that accompanied the initial release, which was very large—over 80 pages. This policy uses, for example, 28 object classes, 115 permissions, and 253 type names of which 21 are parameterized—meaning a potentially unbounded number of type names. Thus, full policy specifications, which build on top of the basic policy that accompanies SELinux, tend to be quite lengthy, complex, and full of low-level detail. When the macros in the initial basic policy are fully expanded, the result is several tens of thousands of rules, mostly `allow` rules. Some specific examples provide the flavor:

```
allow file_t file_t:file transition
allow init_t file_t:process execute
type_transition cardmgr_t tmp_t:chr_file cardmgr_dev_t
type_transition cardmgr_t device_t:lnk_file cardmgr_lnk_t
```

In practice, rather than modeling the full policy, one can identify a slice of the policy that is most relevant to a particular aspect of the system.

6 Related Work

Most closely related to the work described in this paper is work on theorem proving support for automata models. STeP [6], which is based on first-order logic, supports the proof of temporal logic properties of transition systems, including timed systems. The Larch prover [13] is used to provide theorem proving support in IOA; some examples of its use in this setting are described in [23, 36]. Müller [32] developed a verification environment for I/O automata in Isabelle that includes a meta-theory of automata and their properties. This work has also been built on in order to provide theorem proving with Isabelle in IOA [36]. In both the Larch and Isabelle proof support in IOA, the required user effort is reduced by automating many of the proof steps.

7 Conclusions and Future Work

This paper has illustrated how a modeling environment based on specification templates, supporting theories, and a set of high level proof steps implemented as strategies (or tactics) in a general purpose theorem prover can be used to provide access to the prover in a variety of settings in a usefully structured way. For this purpose, it helps when the central model of the modeling environment is a sufficiently general one such as the automata model used in TAME. Then, it is likely that specifications in the languages of many other tools can be represented in the model, and hence translations of these specifications can be matched to specification templates in the modeling environment. Since many sub-steps performed automatically in high-level proof steps depend on name, type, and other structural conventions in the templates, translation into the templates allows high level proof steps from the modeling environment to be used as a basis for high level proof steps in the new settings.

For purposes of illustration, the paper has described in detail how TAME is being used to supply proof support in three different settings: SCR, a specification language and toolset intended primarily for modeling embedded control systems; TIOA, a specification language and toolkit that is suited to modeling distributed systems and protocols; and a set of tools for analyzing SELinux policy specifications. Each of these settings has it own unique challenges with regard to finding the most preferable translation scheme for specifications; nevertheless, the generality of TAME's automaton model makes it possible to represent specifications from all three settings in TAME with at most minor modifications to TAME. In all of these cases, the theorem proving support in TAME provides the basis for theorem proving in the new setting, but requires some extensions. One advantage to basing a modeling environment such as TAME on a general purpose theorem prover is that such extensions are relatively cheap: rather than (say) implementing the extensions in low level code and having to establish their soundness, one needs only to develop additional strategies in the general purpose prover.

The major challenge for the near future is to complete the integration of TAME with TIOA. In particular, the best representations of preconditions and effects of trajectories and transitions need to be determined. Hybrid I/O automata (HIOA) are similar to TIOA, the difference being that trajectories in HIOA can represent continually changing inputs or outputs rather than internal changes, as in TIOA. This becomes something to take into account when two automata are composed by matching outputs to inputs. Once PVS extensions under development to facilitate reasoning about compositions of I/O automata are complete, the distinction between TIOA and HIOA may become important in any future TAME support for HIOA.

Acknowledgements

I wish to thank Ralph Jeffords, Sayan Mitra, Nancy Lynch, and Tevfik Bultan for helpful discussions. I especially thank Sayan Mitra and HongPing Lim for providing the TIOA code for the example in Figure 9.

References

1. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb., 2001.
2. Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, OR, Jan. 14–15 2002.
3. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
4. Myla Archer, Elizabeth Leonard, and Matteo Pradella. Analyzing Security-Enhanced Linux policy specifications. In *Proc. IEEE 4th Int'l Wkshp. on Policies for Distr. Systems and Networks (POLICY 2003)*. IEEE Comp. Soc. Press, 2003.
5. Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, Redmond, WA, June 2002.

6. Nikolaj Bjørner, Zohar Manna, Henny B. Sipma, and Tomas E. Uribe. Deductive verification of real-time systems using STeP. In *Proceedings of ARTS'97*, volume 1231 of *Lect. Notes in Comp. Sci.*, pages 22–43. Springer-Verlag, May 1997.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th International Conference on Computer-Aided Verification (CAV)*, pages 154–169. Springer-Verlag, 2000.
8. B. Di Vito. A PVS prover strategy package for common manipulations. Technical Memorandum NASA/TM-2002-211647, NASA Langley Research Center, Hampton, VA, April 2002.
9. S. J. Garland and N. A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Technical Report MIT/LCS/TR-762, MIT Laboratory for Computer Science, August 1998.
10. Stephen Garland, Nancy Lynch, Joshua Tauber, and Mandana Viziri. IOA User Guide and Reference Manual. Technical Report MIT-LCS-TR-961, MIT CSAIL, Cambridge, MA, 2004.
11. B. Gebremichael and F. W. Vaandrager. Specifying urgency in timed I/O automata. In *Proc. 3rd IEEE Intern. Conf. on Softw. Eng. and Form. Meths. (SEFM 2005)*, pages 64–73, Koblenz, Germany, September 5-9 2005. IEEE Comp. Soc.
12. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *9th International Conference on Computer-Aided Verification (CAV'97)*. Springer-Verlag, 1997.
13. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.
14. C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal on Computer System Science and Engineering*, 20(1):19–35, January 2005.
15. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11):927–948, November 1998.
16. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, Apr.–June 1996.
17. Kathryn Heninger, David L. Parnas, John E. Shore, and John W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.
18. Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, January 1980.
19. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *15th International Conference on Computer-Aided Verification (CAV)*, pages 262–274. Lecture Notes in Computer Science 2725, Springer-Verlag, 2003.
20. D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. A mathematical framework for modeling and analyzing real-time systems. In *The 24th IEEE Intern. Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.
21. D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O automata.* Synthesis Lectures on Computer Science. Morgan Claypool Publishers, 2005.
22. Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. Technical report, National Security Agency, Jan. 2, 2001.

23. V. Luchangco, E. Söylemez, S. Garland, and N. Lynch. Verifying timing properties of concurrent algorithms. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII: Proc. of the 7th IFIP WG6.1 Intern. Conference on Formal Description Techniques* (FORTE'94, Berne, Switzerland, Oct. 1994), pages 259–273. Chapman and Hall, 1995.
24. Victor Luchangco. Personal communication. 1996.
25. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.
26. N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.
27. N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.
28. M. Merritt, F. Modugno, and M. R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, eds., *CONCUR'91: 2nd Intern. Conference on Concurrency Theory*, vol. 527 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1991.
29. S. Mitra and M. Archer. Developing strategies for specialized theorem proving about untimed, timed, and hybrid I/O automata. In *Proc. 1st Intern. Wkshp. on Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, Rome, Italy, Sept. 8 2003. NASA Proceedings NASA/CP-2003-212448; also, NRL Memorandum Rept. NRL/MR/5540–0308722.
30. Sayan Mitra and Myla Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theor. Comp. Sci.*, 125(2):45–65, 2005.
31. C. Muñoz and M. Mayero. Real automation in the field. Technical Report Interim ICASE Report No. 39, NASA/CR-2001-211271, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, December 2001.
32. Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universität München, Sept. 1998.
33. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Comp. Sci. Lab., SRI Intl., Menlo Park, CA, Nov. 2001.
34. Stephen Smalley and Timothy Fraser. A security policy configuration for Security-Enhanced Linux. Technical report, National Security Agency, Jan. 2, 2001.
35. Henri B. Weinberg. Correctness of vehicle control systems: A case study. Master's thesis, Massachusetts Institute of Technology, Feb. 1996.
36. Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2003.