Fundamenta Informaticae 70 (2006) 75–110
IOS Press

Towards Integrated Verification of Timed Transition Models

Mark Lawford*

Vera Pantelic*

Hong Zhang

Software Quality Research Lab, McMaster University

1280 Main St W., Hamilton, ON, Canada L8S 4K1

lawford@mcmaster.ca, pantelv@mcmaster.ca, zhangh5@cas.mcmaster.ca

Abstract. This paper describes an attempt to combine theorem proving and model-checking to formally verify real-time systems in a discrete time setting. The Timed Automata Modeling Environment (TAME) has been modified to provide a formal model for Time Transition Models (TTMs) in the PVS proof checker. Strong and weak state-event observation equivalences are formalized in PVS for state-event labeled transition systems (SELTS), the underlying semantic model of TTMs. The state-event equivalences form the basis of truth value preserving abstractions for a real-time temporal logic. When appropriate restrictions are placed upon the TTMs, their PVS models can be easily translated into input for the SAL model-checker. A simple real-time control system is specified and verified using these theories. While these preliminary results indicate that the combination of PVS and SAL could provide a useful environment to perform equivalence verification, model-checking and compositional model reduction of real-time systems, the current implementation in the general purpose SAL model-checker lags well behind state of the art real-time model-checkers.

Keywords: Real-time, equivalence verification, theorem proving, PVS, model-checking, model reduction, SAL

1. Introduction

Timed Transition Models (TTMs) are a form of guarded transition systems that can be used to conveniently model real-time systems in a discrete time setting [23, 26]. In particular one may model a

Address for correspondence: M. Lawford, Software Quality Research Lab, Dept. of Computing and Software, McMaster University, 1280 Main St W., Hamilton, ON, Canada L8S 4K1
*This work was partially support by the Natural Sciences and Engineering Research Council of Canada

system's desired behavior or specification using one TTM and the actual implementation of the system using another, more detailed TTM [13]. One can then verify that the implementation is in some sense equivalent to the specification. For an appropriately defined equivalence relation, one can "reduce before use", performing compositional model reduction where concurrent component subsystems are replaced with smaller equivalent subsystems before they are composed [14, 13]. To be of practical use, such equivalence verification techniques require some form of mechanized support. This paper describes how the Timed Automata Modeling Environment (TAME) [2, 3] has been modified to provide a formal model for TTMs in the PVS automated proof assistant [28]. Strong and weak versions of state-event equivalences are formalized in PVS for state-event labeled transition systems (SELTS), the underlying semantic model of TTMs, thus providing mechanized support for equivalence verification of TTMs.

If, in addition to equivalence verification, one wishes to model-check the specification and implementation, PVS does have limited model-checking facilities for the branching time Computational Tree Logic (CTL), though the model-checker is not considered to be state of the art and does not have a much needed counter example generation and simulation capabilities. Fortunately the SAL 2 model-checker [22] has a similar, though more restrictive, type system and similar input syntax to PVS while providing state of the art BDD and SAT-solver based model checkers for Linear Temporal Logic (LTL) [7]. By restricting ourselves to finite state TTMs on types with operations supported by SAL, the PVS specifications can be easily translated into input for SAL, providing state of the art model checking capabilities for our real-time setting.

As we will see in the example of section 5, to prove two TTMs are weakly equivalent, we first specify the two TTMs in PVS using the modeling environment, define a relation between the states of these two TTMs and prove that the relation is a weak state-event bisimulation relating the initial states of these two TTMs. If we wish to model-check our system we can translate the more abstract (specification) TTM into SAL. Provided the model-checking formulas of interest satisfy a form of stuttering invariance, the model-checking results can be used to infer the results for the implementation.

The remainder of this section discusses related work. Section 2 gives a brief description of TTMs, SELTS and a simple real-time state-event temporal logic. The formalization of TTMs and SELTS in PVS and SAL is described in section 3. Section 4 gives the definitions of strong and weak state-event equivalences, describes how the equivalences can be used to perform compositional model reduction and then outlines their formalization in PVS. The results of a mechanized verification of an industrial real-time controller modeled using TTMs in PVS and SAL is given in section 5. As a basis for comparison of the model-checking results, timed automata models of the controllers are partially verified using the UPPAAL real-time model-checker [11]. Finally section 6 summarizes the method's benefits, limitations and possible extensions.

1.1. Related Work

The recent survey article [31] provides an extensive overview of the various techniques and tools that can be used for the formal specification and verification of real-time systems. It covers equivalence verification, model checking and model reduction techniques that have been applied to both continuous and discrete time settings. While each of these topics have been addressed previously, there is no single tool suite that lets the user combined these methods in a particular formalism. This paper illustrates some potential benefits of combining methods and outline how the methods can be integrated using a combination of theorem proving and model-checking.

The Timed Automata Modeling Environment (TAME) [2, 3] is a special-purpose interface to PVS designed to support developers of software systems in proving invariants. It supports the creation of PVS descriptions of three different automata models: Lynch-Vaandrager (LV) timed automata [18], I/O automata [17], and the automata model that underlies SCR specifications [10]. It does not include support for verifying different types of equivalences on pairs of automata models, nor does it support automated composition of automatons. The user must combine the individual automaton descriptions to produce a single TAME specification by extracting the common variables to produce a single TAME specification.

TAME does not support TTMs directly and its representation of time as part of the state variables is not suitable for TTMs. In TAME, the time variable *now* is explicitly changed in the LV timed automata by a special *time-passage* action ν. The time requirement for other *non-time-passage* actions are checked against the *first* and *last* value of the corresponding action. TAME uses the real numbers extended with ∞ to represent time values. In our TTM model, we use the extended natural numbers to represent time values up to the resolution of a global clock tick. A special *tick* action is needed to update the clocks associated with *non-tick* actions. The actions also need to satisfy the state variable requirements appearing in guard conditions, a common situation in control systems. The PVS theories underlying TAME provide the basis for our formalization of TTMs in PVS. We make use of some of the basic theories and follow a similar template based method to make the theories easier to use. As we have used TAME as the initial basis for our TTM models in PVS, our method currently also requires manual composition of TTMs. Most significantly, we have add theories defining equivalences between pairs of models, a feature previously absent in TAME.

In [27] Ostroff outlines a compositional method for proving Real-Time Temporal Logic properties of TTM modules. The work makes extensive use of the results of [13, 14] to provide model reduction based upon state-event equivalences. He uses the Delayed Reactor Trip (DRT) example of [13, 15] to illustrate the proof methodology using a combination of the StateTime tool [25] for modeling and the STeP theorem prover and model-checker [19]. We examine a variation of the same DRT example in section 5. The main distinction between [27] and the current work is that here we provide a means of rigorously verifying equivalence of TTMs to provide provably correct abstractions which can then be used for compositional model reduction as in [13, 14] or compositional reasoning as done in [27].

Verifying the state-event equivalences described in this paper for finite state TTMs reduces to solving the relational coarsest partition problem on the underlying transition structure [13], and hence can be solved using model-checking techniques. While there exist model-checking tools such as MOCHA [1] and UPPAAL [11], these tools do not directly support the verification of user defined equivalence relations and they do not directly support the semantics of TTMs. Our experience using an interactive theorem prover such as PVS to verify systems like the example in section 5 indicates that a combination of theorem proving to decompose the problem and model-checking to discharge parts of the proof obligation would be the most effective combination. Unfortunately soundness problems with PVS' built-in model-checker limited our ability to test this hypothesis. The recently released SAL 2 model-checker [22], with a type system similar, though more restrictive than PVS, has provided an opportunity to use model-checking techniques on TTM specification, though this currently requires a manual translation of the PVS model to a SAL 2 model. In this paper SAL is used to verify temporal logic properties of TTMs that have been reduced using equivalences verified in PVS. We have not yet tried exporting equivalence proof subgoals from PVS to SAL, though this should be possible for a suitably restricted class of TTMs to deal with SAL's type restrictions.

2. Preliminaries

This section introduces the TTMs that will be used as high level representations of systems that motivate the state-event approach taken in this work. The SELTS described later will be used as our underlying semantic model.

2.1. Timed Transition Models

We use a modified version of the Timed Transition Models (TTMs) employed in [26]. To simplify the problem of equivalence verification, the initial condition is limited to specifying a unique initial state instead of (possibly) multiple initial states.

A *Timed Transition Model* (TTM) $M := (\mathcal{V}, \Theta, \mathcal{T})$, where $\mathcal{V}$ is a set of variables, Θ is an initial condition, and $\mathcal{T}$ is a finite set of transitions.

$\mathcal{V}$ always includes two special variables: the global time variable t and an activity variable which we will usually denote by x. A TTM's activities typically corresponds to its modes, so x is used to track the system's current mode. For $v \in \mathcal{V}$ the range space of v is $Range(v)$ (e.g. $Range(t) = \mathbb{N}$ where $\mathbb{N} := \{0, 1, 2, \ldots\}$). We define the set of *state assignments of* M to be $\mathcal{Q} := \times_{v_i \in \mathcal{V}} Range(v_i)$. For a state assignment $q \in \mathcal{Q}$ and a variable $v \in \mathcal{V}$, we will denote the value of v in state assignment q by $q(v)$ where $q(v) \in Range(v)$. This notation can be extended to expressions over $\mathcal{V}$ in a natural way.

$\mathcal{T}$ is the transition set. A transition α is a 4-tuple

$$\alpha := (e_\alpha, h_\alpha, l_\alpha, u_\alpha)$$

where e_α is the transition's enablement condition (a boolean valued expression in the variables of $\mathcal{V}$), h_α is the operation function, and $l_\alpha \in \mathbb{N}$ and $u_\alpha \in \mathbb{N} \cup \{\infty\}$ are the lower and upper time bounds respectively with $l_\alpha \leq u_\alpha$. We say that α is *enabled* when $q(e_\alpha) = true$. The operation function $h_\alpha : \mathcal{Q} \to \mathcal{Q}$ is a partial function, defined when $q(e_\alpha) = true$, that maps the current state assignment to the new state assignment when the transition occurs. $\mathcal{T}$ always contains the special transition $tick := (true, [t : t+1], -, -)$ which represents the passage of time on the global clock. It is the only transition that affects the time variable t and also has no lower or upper time bound.

Θ is a boolean valued expression in the variables of $\mathcal{V}$ that identifies a unique initial state of the system.

TTM Semantics A *trajectory* of a TTM is any infinite string of the TTM state assignments connected by transitions, of the form $q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \ldots$. The interpretation is that q_i goes to q_{i+1} via the transition α_i. A state trajectory $\sigma := q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \ldots$ is a *legal* trajectory of a TTM M if it meets the following four requirements:

1. **Initialization:** The initial state assignment satisfies the initial condition - i.e. $q_0(\Theta) = true$.
2. **Succession:** $\forall i, q_{i+1} = h_{\alpha_i}(q_i) \wedge q_i(e_{\alpha_i}) = true$.
3. **Ticking:** There is an infinite number of $\alpha_i = tick$. This eliminates the possibility of "clock stoppers" in the trajectory where an infinite number of non-$tick$ transitions occur consecutively without being interleaved with any $tick$s.

4. **Time Bounds:** To determine if the trajectory σ satisfies the time bound requirements of M, we associate with each non-$tick$ transition α, a counter variable c_α with $Range(c_\alpha) = \mathbb{N}$. We denote the set of transition counters by $C := \{c_\alpha : \alpha \in \mathcal{T} - \{tick\}\}$. From the trajectory σ we derive the *full trajectory* $\bar{\sigma} := \bar{q}_0 \xrightarrow{\alpha_0} \bar{q}_1 \xrightarrow{\alpha_1} \bar{q}_2 \xrightarrow{\alpha_2} \ldots$, where each $\bar{q}_i \in \bar{\mathcal{Q}} = \mathcal{Q} \times \mathbb{N}^C$ is obtained from σ by extending each q_i as follows:

   For all $c_\alpha \in C$, $\bar{q}_0(c_\alpha) = 0$ and for $i = 0, 1, 2, \ldots$

$$\bar{q}_{i+1}(c_\alpha) = \begin{cases} \bar{q}_i(c_\alpha) + 1, \text{ if } q_i(e_\alpha) \wedge \alpha_i = tick \\ 0, \text{ if } \neg q_{i+1}(e_\alpha) \vee \alpha_i = \alpha \\ \bar{q}_i(c_\alpha), \text{ otherwise} \end{cases}$$

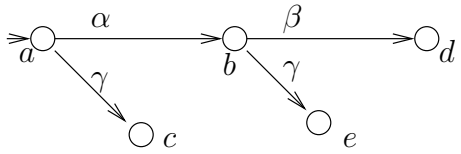
   The trajectory σ satisfies the time bounds of M iff the following conditions hold in $\bar{\sigma}$ for all $i \in \mathbb{N}$:

   (i) $\alpha_i = tick$ iff for all $\alpha \in \mathcal{T} - \{tick\}$, $q_i(e_\alpha) = true$ implies $\bar{q}_i(c_\alpha) < u_\alpha$, and

   (ii) $\alpha_i = \alpha$, $\alpha \in \mathcal{T} - \{tick\}$ iff $l_\alpha \leq \bar{q}_i(c_\alpha) \leq u_\alpha$.

Condition (i) means that upper time bounds on transitions represent hard time bounds by which time the transitions are guaranteed to occur if they are not preempted. Note that any loop of transitions in a TTM must have at least one transition with a non-zero upper time bound.

As a small example, consider the TTM M shown in Fig. 1 together with its legal trajectories. The



$$\begin{aligned} \mathcal{V} &:= \{u, v, t, x\} \\ \Theta &:= u = 0 \wedge v = 1 \wedge x = a \\ \mathcal{T} &:= \{\alpha, \beta, \gamma, tick\}, \text{ where} \\ \alpha &:= (u \geq 0, [u : u + v], 0, 2) \\ \beta &:= (true, [u : u + 1, v : v - 1], 2, \infty) \\ \gamma &:= (v \geq 0, [\,], 2, 2) \end{aligned}$$

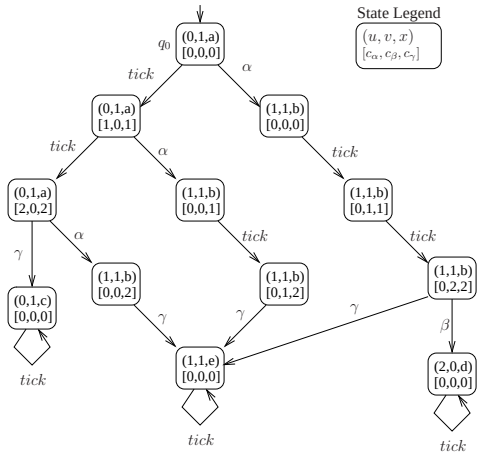


Figure 1. Example TTM M (left) and its legal trajectories (right).

full enablement conditions for the transitions should also include conditions derived from the graph. For instance, in the case of γ, the full enablement condition is $e_\gamma := v \geq 0 \wedge (x = a \vee x = b)$. When describing TTM transitions we will usually omit these activity variable conditions since they are obvious from the transition diagram. Finally, the special transition $tick$ is declared to implicitly be in $\mathcal{T}$ and hence is omitted from the list of M's transitions.

In writing out the operation functions of the transitions of M we employ Ostroff's assignment format [23]. When a transition occurs, the new value of the activity variable x is obtained from the transition diagram. The other variables that are affected by the transition are listed in the form $[v_1 : expr_1, v_2 :$

$expr_2, \ldots, v_n : expr_n]$ with the interpretation that variables v_1 to v_n are assigned the new values given by the simultaneous evaluations of expressions $expr_1$ to $expr_n$ respectively. The operation function acts as the identity on variables not listed in the assignment statement.

If we let the current state assignment be represented by a 4-tuple of the form (u, v, x, t), then a legal trajectory of M would be $q_0 \overset{tick}{\rightarrow} q_1 \overset{\alpha}{\rightarrow} q_2 \overset{tick}{\rightarrow} q_3 \overset{\gamma}{\rightarrow} q_4 \overset{tick}{\rightarrow} \ldots$, i.e. $(0,1,a,0) \overset{tick}{\rightarrow} (0,1,a,1) \overset{\alpha}{\rightarrow} (1,1,b,1) \overset{tick}{\rightarrow} (1,1,b,2) \overset{\gamma}{\rightarrow} (1,1,e,2) \overset{tick}{\rightarrow}$ where from q_4 onward the trajectory is continued by an infinite string of $tick$s. Note that after the second occurrence of $tick$, γ is forced to occur. A $tick$ could not take place from q_3 since γ has $u_\gamma = 2$ and, upon reaching q_3, e_γ has been true for two $tick$s already.

If the initial condition for M is changed to $\Theta := (u = 0 \wedge v = -1 \wedge x = a)$, then a legal trajectory is $(0,-1,a,0) \overset{\alpha}{\rightarrow} (-1,-1,b,0) \overset{tick}{\rightarrow} (-1,-1,b,1) \overset{tick}{\rightarrow} (-1,-1,b,2) \overset{tick}{\rightarrow}$ where again this trajectory is continued by an infinite number of $tick$ transitions. This trajectory illustrates our interpretation of $u_\beta = \infty$. We do not insist on "fairness", allowing trajectories such as the one above where β is a possible next transition for an infinitely long time, although it does not occur. Thus an upper time bound of ∞ means that a transition is possible but is not forced to occur in a legal trajectory.

To be useful for designing real systems, a formalism must provide a means of decomposing large systems into smaller, more manageable subsystems. Complex systems are then typically constructed from interacting components running in parallel. In [24] Ostroff defines a TTM parallel composition operator that allows for shared variables and synchronous (shared) transitions. We extend this TTM parallel composition operator to handle nondeterministic operation functions. In the following definition we denote the state assignments over a set of variables $\mathcal{V}$ by $\mathcal{Q}_\mathcal{V} := \times_{v \in \mathcal{V}} Range(v)$. For $\mathcal{U} \subseteq \mathcal{V}$ the *natural state assignment projection* $P_\mathcal{U} : \mathcal{Q}_\mathcal{V} \rightarrow \mathcal{Q}_\mathcal{U}$ maps a state assignment over $\mathcal{V}$ to its corresponding state assignment over $\mathcal{U}$. In order to allow us to distinguish between a transition and its label, for $\mathcal{T}$, a given set of transitions (labeled 4-tuples), let $\Sigma(\mathcal{T})$ denote the set of transition labels. For the example TTM of Figure 1, $\Sigma(\mathcal{T}) = \{\alpha, \beta, \gamma, tick\}$.

In order to deal with the possibility of nondeterministic transition functions in TTM composition, to model for example an input variable, we need the *set wise functional product* of $h_1 : Q_1 \rightarrow \mathcal{P}(R_1)$ and $h_2 : Q_2 \rightarrow \mathcal{P}(R_2)$ to be the function:

$$h_1 \otimes h_2 : Q_1 \times Q_2 \rightarrow \mathcal{P}(R_1) \times \mathcal{P}(R_2)$$

such that $(q_1, q_2) \mapsto f_1(q_1) \times f_2(q_2)$. Thus if $R_i' \subset R_i$ and $f_i(q_i) = R_i'$ for $i = 1, 2$ then $f_1 \otimes f_2(q_1, q_2) = R_1' \times R_2' = \{(r_1, r_2) : r_1 \in R_1' \text{ and } r_2 \in R_2'\}$ while $f_1 \times f_2(q_1, q_2) = (R_1', R_2')$. We can extend the set wise product operator to handle functions that range over elements instead of sets. For example with f_1 as above, if $f_2 : Q_1 \rightarrow R_2$ then define $f_1 \otimes f_2(q_1, q_2) = f_1(q_1) \times \{f_2(q_2)\}$.

Definition 2.1. Given two TTMs $M_i := \langle \mathcal{V}_i, \Theta_i, \mathcal{T}_i \rangle, i = 1, 2$, the *parallel composition of* M_1 *and* M_2 is given by $M_1 \| M_2 := \langle \mathcal{V}_1 \cup \mathcal{V}_2, \Theta_1 \wedge \Theta_2, \mathcal{T}_1 \| \mathcal{T}_2 \rangle$, where the composite transition set $\mathcal{T}_1 \| \mathcal{T}_2$ is defined as follows.

(i) If $\alpha := (e, h, l, u) \in \mathcal{T}_1$ with operation function $h : \mathcal{Q}_{\mathcal{V}_1} \rightarrow \mathcal{P}(\mathcal{Q}_{\mathcal{V}_1})$ and $\alpha \notin \Sigma(\mathcal{T}_2)$, then $\alpha := (e, h', l, u) \in \mathcal{T}_1 \| \mathcal{T}_2$ where $h' : \mathcal{Q}_{\mathcal{V}_1 \cup \mathcal{V}_2} \rightarrow \mathcal{P}(\mathcal{Q}_{\mathcal{V}_1 \cup \mathcal{V}_2})$ is the extension of h given by $h' := h \otimes id_{Q_{\mathcal{V}_2 \setminus \mathcal{V}_1}}$. The reverse case when $\alpha := (e, h, l, u) \in \mathcal{T}_2$ and $\alpha \notin \Sigma(\mathcal{T}_1)$ is similarly defined.

(ii) If α is a shared transition, i.e. $\alpha \in \Sigma(\mathcal{T}_1) \cap \Sigma(\mathcal{T}_2)$, with $\alpha := (e_1, h_1, l_1, u_1) \in \mathcal{T}_1$ and $\alpha := (e_2, h_2, l_2, u_2) \in \mathcal{T}_2$ and operation functions $h_i : \mathcal{Q}_{\mathcal{V}_i} \to \mathcal{P}(\mathcal{Q}_{\mathcal{V}_i}), i = 1, 2$ then $\alpha := (e', h', l', u') \in \mathcal{T}_1 \| \mathcal{T}_2$ where

$e' := e_1 \wedge e_2$ is the enablement condition.

$h' : \mathcal{Q}_{\mathcal{V}_1 \cup \mathcal{V}_2} \to \mathcal{P}(\mathcal{Q}_{\mathcal{V}_1 \cup \mathcal{V}_2})$ is the function such that $h'(q) := \{q' \in \mathcal{Q}_{\mathcal{V}_1 \cup \mathcal{V}_2} : P_{\mathcal{V}_1}(q') \in h_1 \circ P_{\mathcal{V}_1}(q)$ and $P_{\mathcal{V}_2}(q') \in h_2 \circ P_{\mathcal{V}_2}(q)\}$.

$l' := max(l_1, l_2)$ is the lower time bound.

$u' := min(u_1, u_2)$ is the upper time bound.

Condition (i) states that if the transition $\alpha := (e, h, l, u)$ of M_1 is not a shared transition then the new operation function in the composite system is given by $h'(q) = \{q' \in \mathcal{Q}_{\mathcal{V}_1 \cup \mathcal{V}_2} : P_{\mathcal{V}_1}(q') \in h \circ P_{\mathcal{V}_1}(q) \wedge P_{\mathcal{V}_2 \setminus \mathcal{V}_1}(q') = P_{\mathcal{V}_2 \setminus \mathcal{V}_1}(q)\}$. The value of variables not in M_1's variable set (i.e. $v \in \mathcal{V}_2 \setminus \mathcal{V}_1$) are left unchanged by a transition occurring only in M_1. Condition (ii) requires that any new assignment to the shared variables ($\mathcal{V}_1 \cap \mathcal{V}_2$) made by a shared α transition must be possible assignments by α in both M_1 and M_2.

2.2. State-Event Labeled Transition Systems

SELTS extend Labeled Transition Systems (LTS) by adding a state output map [13]. We further add an event output map that is used in the definition of equivalence of two SELTS. SELTS provides a convenient way of illustrating the combination of state and event dynamics of TTMs. Rather than using an equivalent purely state-based or event-based formalism, SELTS explicitly retain the separation of state and event information that provides the intuition for the equivalences definitions. As we will see, the equivalence kernel of the state output map provides an initial state partition which is further refined using the event dynamics to obtain state-event equivalences.

Definition 2.2. A *State-Event Labeled Transition System (SELTS)* is an 8-tuple $\mathbb{Q} := \langle Q, Q', \Sigma, \Sigma', R_\Sigma, q_0, ps, pa \rangle$ where Q and Q' are an at most countable set of states and state outputs, respectively, Σ and Σ' are a finite set of elementary events (actions) and event outputs, respectively, $R_\Sigma = \{\xrightarrow{\alpha} : \alpha \in \Sigma\}$ is a set of binary relations on Q, $q_0 \in Q$ is the initial state, $ps : Q \to Q'$ is the state output map, and $pa : \Sigma \to \Sigma'$ is the event output map.

In the above definition, $q \xrightarrow{\alpha} q'$ (where $\alpha \in \Sigma$ and $q, q' \in Q$) means the SELTS can move from state q to q' by executing elementary action α. TTMs can be expanded to a corresponding SELTS so that we can analyze it. The legal trajectories of the TTM from Figure 1 are reproduced on the left of Figure 2. The top line of each state in the graph contains the state assignments of the system variables in the format (u, v, x). The second line of each state contains the current values of each transition's counter variable in the format $[c_\alpha, c_\beta, c_\gamma]$. The states of the graph are elements of M's set of extended state assignments $\overline{\mathcal{Q}}$, which include all state variable, activity variable and counter variable information. The initial state q_0 of the graph is indicated by an entering arrow. A TTM's legal trajectories are all infinite sequences and as can be seen from Figure 1, every path starting from q_0 can be extended to an infinite path. The transitions' counter variables are only used to obtain the structure of the graph. They are not part of the system's observed timed behavior. The counter variables are hidden variables, the values of

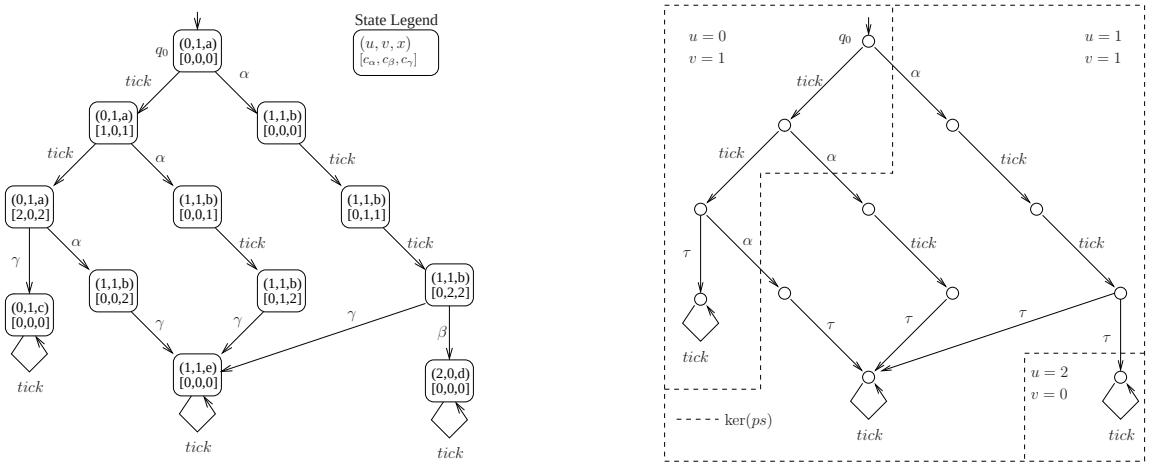


Figure 2. Legal trajectories of TTM M (left) and SELTS for its timed behavior of α and u, v (right)

which determine the Markovian dynamics of the structure. Thus if we were to treat the graph on the left of Figure 2 as a SELTS for M, the state output map would be the canonical projection from extended state assignments to state assignments $ps : \overline{\mathcal{Q}} \to \mathcal{Q}$. Assuming all of the events were observable, then $pa := id$, the identity map on transition labels.

Often a TTM's activity variable x plays a role similar to the counter variables in that it is only used to keep track of when transitions might possibly be enabled. Similarly, not all transition labels may be of significance. For instance M may be designed to share α and $tick$ transitions while β and γ represent transitions that are internal to M. If one's real interest in the TTM M was the timed behavior of the variables u and v and the occurrence of α transitions, then this could be represented by the SELTS on the right of Figure 2 where $ps(\overline{q}) = (u, v)$ and $pa(\beta) = pa(\gamma) = \tau$.

2.3. A Simple Real-Time State-Event Temporal Logic

In this section we introduce state-event temporal logics as an abstract method for reasoning about SELTS behavior with particular attention being paid to a simple real-time logic. In general when discussing SELTS throughout this section $AP, AP_1, AP_2, \ldots$ will represent sets of atomic propositions and the SELTS state output map will map each state to the set of atomic propositions satisfied by the state (i.e. $ps : Q \to \mathcal{P}(AP)$). We now give a brief summary of temporal logic and refer the reader to [4, 20, 23] for the full details. Following [23], the state-event sequences defined above will play the role of the state sequences in [20]. This will allow us to distinguish state formulas and state-event formulas. RTTL, as an example of a state-event temporal logic, is based upon Manna-Pnueli temporal logic with additional proof rules for dealing with real-time ($tick$ event) properties. To allow us to express simple real-time properties we add a bounded "until" operator.

Before defining the computations of a SELTS, we will introduce some notation to aid in our discussion of generated and observed state-event sequences. We are interested in sequences of both states and events so for notational convenience we define $\Sigma_- := \Sigma \cup \{-\}$, the event set extended with the "null" event symbol $-$, and $S := Q \times \Sigma_-$. For $s = (q, \alpha) \in S$, in addition to the set of atomic propositions found in $ps(q)$ we associate the atomic proposition $\eta = \alpha$. We refer to η as the (next) transition variable.

The computations of the SELTS $\mathbb{Q}$ will then be a subset of the union of the set of all finite, non-empty, state-event sequences S^+, and the set of all infinite state-event sequences S^ω. As a notational convenience, we introduce the notation $|\sigma|$, which for $\sigma = s_0 s_1 s_2 \ldots s_n \in S^+$ is defined as $|\sigma| = n$ and for $\sigma = s_0 s_1 s_2 \ldots \in S^\omega$, $|\sigma| = \omega$.

Definition 2.3. Given a SELTS $\mathbb{Q}$, the set of *computations* of $\mathbb{Q}$, denoted $\mathcal{M}(\mathbb{Q})$, is the largest subset of $S^+ \cup S^\omega$ such that for all $\sigma \in \mathcal{M}(\mathbb{Q})$,

$$\sigma = \begin{cases} s_0 s_1 \ldots s_n &= (q_0, \alpha_0)(q_1, \alpha_1) \ldots (q_n, -) \in S^+, \quad \text{or,} \\ s_0 s_1 \ldots &= (q_0, \alpha_0)(q_1, \alpha_1) \ldots \in S^\omega \end{cases}$$

and

(i) Initialization: q_0 is the initial state of $\mathbb{Q}$.

(ii) Succession: $0 \leq i < |\sigma|$ implies $\alpha_i \in \Sigma$ and $q_i \xrightarrow{\alpha_i} q_{i+1}$ in $\mathbb{Q}$.

(iii) Diligence: $\alpha_i = -$ iff $i = |\sigma|$ and for all $\alpha \in \Sigma$ and $q \in Q$, $q_i \overset{\alpha}{\not\rightarrow} q$.

Condition (iii) above states that the only finite sequences in $\mathcal{M}(\mathbb{Q})$ are those which terminate in a state where no transitions are possible and hence the final "event" of the state-event sequence is denoted by $-$. This diligence condition differs from that of [20] in that there is no idling transition in our setting so we allow finite sequences of states to be computations and modify our definition of temporal semantics accordingly [4].

State-event formulas are arbitrary boolean combinations of atomic predicates. We say that a state-event formula is a *state formula* if is does not include any transition predicates such as $\eta = \alpha$. For example, $(y \leq 10 \wedge x = atdelay) \vee t = 5$ is both a state formula and a state-event formula while $\eta = \alpha \vee y = 3$ is a state-event formula but not a state formula. State-event formulas (and hence state formulas) do not contain any temporal operators. For a state formula F_s and a state q, we use the standard inductive definition of satisfaction and write $q \models F_s$ when F_s is true in state q. Similarly the definition of satisfaction can be extended to any state-event pair $s \in S$ and any state-event formula F_{se}.

In the following inductive definition of satisfaction of temporal state-event formulas we will consider an arbitrary (possibly finite) state-event sequence $\sigma = s_0 s_1 \ldots = (q_0, \alpha_0)(q_1, \alpha_1) \ldots$. Henceforth σ^k will be used to denote the k-shifted suffix of σ,

$$\sigma^k := s_k s_{k+1} \ldots = (q_k, \alpha_k)(q_{k+1}, \alpha_{k+1}) \ldots$$

when it exists (i.e. when $|\sigma| \geq k$). When talking about projections of computations we will denote the prefix of σ up to position k by $\sigma^{-k} = (q_0, \alpha_0)(q_1, \alpha_1) \ldots (q_k, \alpha_k)$. For each $\alpha \in \Sigma$ we use the notation $\#\alpha(\sigma, i)$ to denote the number of α transitions that occur between q_0 and q_i in the state-event sequence σ. If $|\sigma| < i$ then $\#\alpha(\sigma, i)$ is undefined.

Definition 2.4. For temporal formulas F, F_1, F_2 and state-event sequence σ, the *satisfaction relation* $\models$ is defined as follows:

- If $F \in AP$ is an atomic predicate, then $\sigma \models F$ iff $s_0 \models F$ (i.e. $F \in ps(q_0)$)

- If $F := (\eta = \alpha)$, then $\sigma \models F$ iff $pa(\alpha_0) = \alpha$
- $\sigma \models F_1 \vee F2$ iff $\sigma \models F_1$ or $\sigma \models F_2$
- $\sigma \models F_1 \wedge F2$ iff $\sigma \models F_1$ and $\sigma \models F_2$
- $\sigma \models \neg F$ iff $\sigma \not\models F$
- $\sigma \models \bigcirc F$ iff σ^1 exists and $\sigma^1 \models F$
- $\sigma \models F_1 \mathcal{U} F_2$ iff $\sigma \models F_2$ or $\exists k > 0$ such that σ^k is defined, $\sigma^k \models F_2$ and $\forall i, 0 \leq i < k, \sigma^i \models F_1$.
- $\sigma \models F_1 \mathcal{U}^{\alpha}_{[l,u]} F_2$ iff $\sigma \models F_2$ or $\exists k > 0$ such that σ^k is defined, $\sigma^k \models F_2$ and $\forall i, 0 \leq i < k, \sigma^i \models F_1$ and $l \leq \#\alpha(\sigma, k) \leq u$.

Given a SELTS $\mathbb{Q}$ and a temporal formula F, we say that F is $\mathbb{Q}$-*valid*, written $\mathbb{Q} \models F$, iff for all $\sigma \in \mathcal{M}(\mathbb{Q})$, $\sigma \models F$.

The "next" operator $\bigcirc$ and "until" operator $\mathcal{U}$ are typically used to define additional operators. In particular the "eventually" ("future") operator $\Diamond F$, which denotes $(true)\mathcal{U}F$, and the "henceforth" ("always") operator $\Box F$, which is an abbreviation of $\neg\Diamond\neg F$. As an example of a temporal formula, consider $F := \Box \bigcirc true$. F is satisfied only by those σ such that $|\sigma| = \omega$. The $\mathcal{U}^{\alpha}_{[l,u]}$ operator is just the until operator subject to the restriction that for a formula $F_1 \mathcal{U}^{\alpha}_{[l,u]} F_2$, F_2 must become true after the lth occurrence of events producing α observations and before the $(u+1)$th occurrence of an α observation. In systems in which time is represented by discrete $tick$ events the $\mathcal{U}^{tick}_{[l,u]}$ operator can be used to specify that a system meets hard time bounds. For example, any system satisfying the formula $(true)\mathcal{U}^{tick}_{[0,2]}(\eta = \beta)$ will produce a β event before 3 time units have passed. We will use $\mathcal{U}^{tick}_{\leq k}$ as an abbreviation for $\mathcal{U}^{tick}_{[0,k]}$. For example the above formula can be written as $(true)\mathcal{U}^{tick}_{\leq 2}(\eta = \beta)$.

Fairness Typically when a given transition structure is used as the model for a system, a designer specifies some fairness constraints which a computation must satisfy if it is to be considered a "legal" computation of the system. For example, all systems in RTTL have the fairness constraint that the $tick$ event must occur infinitely often ($\Box\Diamond(\eta = tick)$), that is the system must not stop the clock or permit an infinite number of non-$tick$ transitions to occur between successive clock $tick$s. Given a specification as a temporal formula F, one then is not so much interested in verifying that *all* the computations of the transition structure satisfy F but rather in verifying that all the *legal* computations satisfy F. That is $\mathbb{Q} \models \neg F_{fair} \vee F$, where F_{fair} is the conjunction of all formulas that are to be satisfied by the system's legal computations. In performing such a verification one implicitly assumes that the set of legal computations considered is non-empty (i.e. $\exists \sigma \in \mathcal{M}(\mathbb{Q}), \sigma \models F_{fair}$).

3. Formalization of TTMs in PVS

PVS stands for "Prototype Verification System," and as the name suggests, it is an environment for specification and verification. The system consists of a specification language, a parser, a type checker, and an interactive theorem prover with a powerful collection of inference procedures that are applied interactively under user guidance within a sequent calculus framework. The specification language is based on higher-order logic with a richly expressive type system [28].

3.1. PVS Theories for the Timed Transition Model

Following the lead of the TAME system, we introduce several PVS theories and a pair of templates which support the specification of TTMs directly in PVS. When combined with selected theories from TAME, these theories can provide us with a modeling environment in which the software developer can produce specifications of TTMs in a straightforward way. It also provides us with a validated formal specification that, under appropriate restrictions, can be easily translated into input for the SAL model-checker.

Datatype `time`**:** In a TTM, each `action` (transition) has an associated timer with values in $\mathbb{N}$. The value of the timer is compared with the `lower_bound` and `upper_bound` to decide the `enabled_time` condition for each action. The `upper_bound` ∞ represents the case where there is no final deadline on an action. So the `time` type in our model is the union type of natural numbers and $\{\infty\}$, shown at the top of Figure 3.

The datatype `time` has two constructors. The first constructor, `fintime`, has a natural number parameter `dur` and the recognizer `fintime?`, and the second constructor, `infinity`, has no parameters and the recognizer `inftime?`. We can then reuse the theory `time_thy` from TAME [9] which provides the definitions of the standard arithmetic operators and predicates for time values.

Theory `states`**:** Appearing in the middle of Figure 3, this theory provides a standard record structure for the Timed Transition Model. The theory has four type parameters. They are `activity`, `nt_action`, `internal_state` and `time`. The `nt_action` is the set of all actions excluding the action `tick`. The `states` type defines the record type used to represent the system state. The first field is `activity` which specifies the activity label of the state. The second field is `basic` which represent all the non-time state information. The third field is `action_time` which is a function from `nt_action` to `time`. It associates each non-tick action with a time value. Thus each action is associated with a timer.

Theory `ttm`**:** The theory `ttm` specifies the common time operations of TTMs. It appears in the bottom section of Figure 3. The theory requires seven parameters to define a TTM. The first three have been described above. The parameters `lower_bound` and `upper_bound` are functions from `nt_action` (non-$tick$ actions) to `time` which associate the time bounds with each action. The parameter `enabled_state` is instantiated by a predicate on `action` and `internal_state` that is true only when the action is enabled, based on the value of `internal_state`. The parameter `graph` is instantiated by a predicate on `action` and `activity` that is true only when the action is enabled based on the `activity` label of the state. These parameters are all defined in the TTM template in section 3.2 where they are instantiated according to the specifics of the TTM being modeled.

3.2. PVS Templates for Timed Transition Models

We provide two templates that can be instantiated to simplify the process of specifying a TTM in PVS. In the theory `actions`, we define `action` as the type of all the possible actions in the TTM. The template for the theory `actions` is shown in Figure 4 where it has been instantiated for the TTM in Figure 1. Here the lines append with the comment "%* user *" indicates the TTM specific information supplied by a user of the template. Non-*tick* actions are distinguished with the `nt_action` subtype.

```
time: DATATYPE
  BEGIN
    fintime(dur: nat): fintime?
    infinity: inftime?
  END time

states [activity,nt_action,internal_state:TYPE,time:TYPE]: THEORY
  BEGIN
    states: TYPE = [# activity: activity,
                      basic:internal_state,
                      action_time: [nt_action -> time] #]
  END states

ttm [activity,internal_state, nt_action: TYPE,
    (IMPORTING time_thy, states[activity,nt_action,internal_state,time])
     lower_bound,upper_bound:[nt_action->time],
     enabled_state: [nt_action,internal_state -> bool],
     graph:[nt_action, activity -> bool] ]: THEORY
BEGIN
  s: VAR states
  alpha: VAR nt_action

  enabled_general(alpha,s):bool =
    enabled_state(alpha,s`basic) & graph(alpha,s`activity)

  update_clocks(s): [nt_action->time] =
    (LAMBDA (alpha):
      IF enabled_general(alpha,s) THEN s`action_time(alpha) + one
      ELSE zero ENDIF)

  reset_clocks(ac:nt_action,s): [nt_action->time] =
    (LAMBDA (beta:nt_action):
      IF (enabled_general(beta,s) & beta/=alpha) THEN s`action_time(beta)
      ELSE zero ENDIF)

  enabled_time(alpha, s): bool =
    s`action_time(alpha) >= lower_bound(alpha) &
    s`action_time(alpha) <= upper_bound(alpha)

  enabled_tick(s): bool =
    FORALL alpha: enabled_general(alpha,s) =>
      (s`action_time(alpha) < upper_bound(alpha))
END ttm
```

Figure 3. Datatype `time` and `states` and `ttm` theories

```
actions : THEORY
  BEGIN
  action:DATATYPE
  BEGIN
    tick:tick?
    alpha:alpha?   % * user *
    beta:beta?     % * user *
    gamma:gamma?   % * user *
  END action

  nt_action:TYPE={action:action |action/=tick}

  END actions
```

Figure 4. Instantiated `actions` for M in Fig. 1

Appendix A contains the `ttm_decls` template where a TTM's main declarations are provided. Once again it has been instantiated for the TTM in Figure 1 with the "%* user *" comment indicating user supplied lines specific to this TTM. In the template we import the fixed theory `time_thy` and the instantiated `actions` theory from Figure 4. The `time_thy` contains the definition of all of the constants of datatype `time` (eg. `twenty_nine` is the constant of datatype `time` for the natural number 29). Then we define `activity`, the set of TTM activities. The interpretations of type `internal_state`, functions `lower_bound` and `upper_bound`, `enabled_state` and `graph` have all been discussed above.

Once the values of each of these are filled in for the TTM in Fig. 1, the `ttm` theory can be imported to define the common operations. The function `enabled` combines the `enabled_general`, `enabled_time` and `enabled_tick` conditions to get the final `enabled` condition for all actions. In the transition function `trans`, the definition of `tick`'s effect as well as the resetting of clocks according to the new state variables assigned by other events is the same for all TTMs. The effects of non-tick actions on state variables are specified by the user to complete the `trans` function. Finally, the function `start` specifies valid initial states of the TTM. After defining these functions, we import the TAME `machine` theory [2, 3] which allows us to specify and inductively prove reachability invariants and other properties of the TTM.

3.3. Translation of TTMs to SAL Model-Checker

Theories defining state, transitions and initialization of the system are rewritten into the module language of SAL. We did not need the `machine` theory from PVS that recursively defines the set of reachable states, since SAL is designed for the modular specification of state machines. The `state` type remains largely the same, although SAL offers the option of defining `activ` as a local variable, which would be beneficial in case of the system containing more than one module. In fact all files `ttm.sal`, `time_thy.sal`, `states.sal` are very similar to the corresponding PVS files. To provide an example of how "close" the SAL versions of the TTM files are to the PVS files, the SAL version of the instantiated `ttm_decls.pvs` template of Appendix A appears in Appendix B. In specifying transition relations

SAL's guarded commands style is employed rather than its invariant style to more closely match the TTM formalism and PVS model.

The most significant difference is the `update_clocks` function in `ttm.sal`. In the current version of the PVS model, the counter values associated with any non-*tick* action (`action_time`) is unbounded if its upper bound is ∞. However, since we are using the SAL model-checkers for finite state systems, we want `action_time` to be a finite subrange of $\mathbb{N}$.

In the example of Fig. 1, although the β transition of M has an upper time bound of ∞, the SELTS M in Fig. 2 is finite state since γ preempts β, preventing an infinite number of $tick$s from causing c_β from becoming unbounded. What if γ also had an upper time bound of ∞? How do we generate a finite state representation of the timed behavior of M?

The set of extended state assignments is reduced to produce a finite state set by redefining the $Range$ of the counter variables as follows. For $M := \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ and $\alpha := (e, h, l, u) \in \mathcal{T}$

$$Range_M(c_\alpha) := \begin{cases} \{n \in \mathbb{N} : n \leq l\}, & \text{if } u = \infty \\ \{n \in \mathbb{N} : n \leq u\}, & u < \infty \end{cases}$$

If α has a finite upper time bound u_α, then TTM semantics prevent c_α from being incremented to a value exceeding u_α. For transitions with lower time bound l_α and upper time bound $u_\alpha = \infty$, we redefine the the clock update effect of $tick$ to cease incrementing c_α once it reaches l_α since all values of $c_\alpha \geq l_\alpha$ have the same effect of enabling the transition. We then redefine the set of extended state assignments to use the reduced clock ranges.

Other differences between the PVS and SAL files are mainly due to the limited capabilities of the current version of SAL. For example `states.sal` has more than one state type defined because the current version of SAL does not allow parameters of context to be a function of more than one parameter.

4. State-Event Bisimulation and Equivalence

In this section, we first briefly justify our discrete time setting and choice of equivalence, then give the definitions of strong and weak state-event equivalence together with their model reduction properties and finally describe the theories and templates that formalize the equivalences in PVS.

Discrete time models such as TTMs are sufficiently accurate in many instances, particularly when dealing with digital control systems that sample their inputs. In [15] the authors argue that discrete time models such as TTMs allow for a straight forward application of well known process algebraic equivalences such as observation (bisimulation) equivalence from Milner's CCS [21]. State-event equivalences of SELTS were introduced in [14] and used as the basis for equivalence of TTMs in [13]. Here we use an equivalent state-event bisimulation (a generalization of (event) bisimulations [29, 21]) characterization of the state-event equivalences rather than the homomorphism based characterization of [14, 13].

4.1. Strong State-Event Equivalence and Model Reduction

In this section and the following, let $\mathbb{Q}_i = \langle Q_i, Q, \Sigma_i, \Sigma, R_{\Sigma_i}, q_{i0}, ps_i, pa_i \rangle$, $i = 1, 2$ be SELTSs where $ps_i : Q_i \to Q$ and $pa_i : \Sigma_i \to \Sigma$.

Definition 4.1. A relation $S \subseteq Q_1 \times Q_2$ is a *strong state-event bisimulation* for $\mathbb{Q}_1$ and $\mathbb{Q}_2$ iff $(q_1, q_2) \in S$ implies

(i) $\forall \alpha_1 \in \Sigma_1$, whenever $q_1 \stackrel{\alpha_1}{\rightarrow} q_1'$ then $\exists q_2' \in Q_2, \alpha_2 \in \Sigma_2$ such that ($q_2 \stackrel{\alpha_2}{\rightarrow} q_2'$ and $(q_1', q_2') \in S$ and $ps_1(q_1') = ps_2(q_2')$ and $pa_1(\alpha_1) = pa_2(\alpha_2)$).

(ii) $\forall \alpha_2 \in \Sigma_2$, whenever $q_2 \stackrel{\alpha_2}{\rightarrow} q_2'$ then $\exists q_1' \in Q_1, \alpha_1 \in \Sigma_1$ such that ($q_1 \stackrel{\alpha_1}{\rightarrow} q_1'$ and $(q_1', q_2') \in S$ and $ps_1(q_1') = ps_2(q_2')$ and $pa_1(\alpha_1) = pa_2(\alpha_2)$).

We say that the SELTS are *strongly state-event equivalent*, denoted $\mathbb{Q}_1 \backsim_{se} \mathbb{Q}_2$, iff there exists a strong state-event bisimulation S for $\mathbb{Q}_1$ and $\mathbb{Q}_2$ such that $(q_{10}, q_{20}) \in S$.

For finite state systems $\mathbb{Q}_1$ and $\mathbb{Q}_2$, it is possible to compute the largest state-event bisimulation by solving a version of the Relational Coarsest Partition problem. Further, similar to the results of [8] for the event only case, abstractions based upon strong state-event equivalence preserve truth values under parallel composition [13, 14].

4.1.1. Strong State-Event Model Reduction

We are assuming that only partial state information is provided via the state output map and for the partial event information provided by the event output map all event outputs are observable. In this setting, one of the main results of [13, 14], restated below, is that strongly state-event equivalent systems satisfy the same temporal formulas.

Theorem 4.1. Given two SELTS as above, if $\mathbb{Q}_1 \sim_{se} \mathbb{Q}_2$ then for any temporal formula F, we have $\mathbb{Q}_1 \models F$ iff $\mathbb{Q}_2 \models F$.

4.2. Weak State-Event Equivalence and Model Reduction

In some cases, strong equivalence is more discriminating than we would like because it "observes" unobservable transitions. Therefore we will introduce weak state-event equivalence.

Given SELTS $\mathbb{Q}_1$ and $\mathbb{Q}_2$ as defined above, assume the special event τ represents unobservable events in their common event output set Σ. If an action $\alpha \in \Sigma_i$ maps to $\tau \in \Sigma$ through pa_i, we consider α to be an unobservable τ transition. In this case, when α happens, it does not produce an observable event output, though it may produce an observable change in the state output. For $q, q' \in Q_i$, if $pa_i(\alpha) = \tau$ then when $q \stackrel{\alpha}{\rightarrow} q'$ and $ps_i(q) = ps_i(q')$, there is no change in the state output, but if $ps_i(q) \neq ps_i(q')$, there is a change in state output even though no event output is observed!

The *unobservable state invariant transitive closure* for a given SELTS $\mathbb{Q}_i$ is defined as the relation $\Rightarrow_{se}$ such that for $q, q' \in Q_i$, $q \Rightarrow_{se} q'$ iff $q = q'$ or for some $n > 0$, $\exists q_0, q_1, \ldots, q_n \in Q_i$ and $\alpha_0, \alpha_1, \ldots, \alpha_{n-1} \in \Sigma_i$ such that

(i) $q = q_0 \stackrel{\alpha_0}{\rightarrow} q_1 \stackrel{\alpha_1}{\rightarrow} \cdots \stackrel{\alpha_{n-2}}{\rightarrow} q_{n-1} \stackrel{\alpha_{n-1}}{\rightarrow} q_n = q'$, and

(ii) $ps_i(q_j) = ps_i(q) = ps_i(q')$, for $j = 0, 1, \ldots, n$, and

(iii) $pa_i(\alpha_j) = \tau$, for $j = 0, \ldots, n-1$.

Thus the unobservable state invariant transitive closure is the reflexive and transitive closure within each cell of the equivalence kernel of the state output map of the union of transition relations that produce the silent event output τ.

We use $q \stackrel{\beta}{\Rightarrow}_{se} q'$, where $\beta \in \Sigma_i$, to denote $q \Rightarrow_{se} q_1 \stackrel{\beta}{\rightarrow} q_2 \Rightarrow_{se} q'$, where $q, q', q_1, q_2 \in Q_i$ for a given SELTS $\mathbb{Q}_i$. It will be used in our definition of *weak state-event bisimulation*. The definition uses event output maps in addition to the state output maps used originally in [13]. Formally, *weak state-event bisimulation* is defined as follows.

Definition 4.2. A relation $S \subseteq Q_1 \times Q_2$ is a weak state-event bisimulation for $\mathbb{Q}_1$ and $\mathbb{Q}_2$ iff $(q_1, q_2) \in S$ implies

(i) $\forall \alpha_1 \in \Sigma_1$, whenever $q_1 \stackrel{\alpha_1}{\rightarrow} q_1'$ then

- $(\exists q_2' \in Q_2, \alpha_2 \in \Sigma_2$ where $q_2 \stackrel{\alpha_2}{\Rightarrow}_{se} q_2'$ and $(q_1', q_2') \in S$ and $ps_1(q_1') = ps_2(q_2')$ and $pa_1(\alpha_1) = pa_2(\alpha_2))$, OR
- $(\exists q_2' \in Q_2$ where $q_2 \Rightarrow_{se} q_2'$ and $(q_1', q_2') \in S$ and $ps_1(q_1') = ps_2(q_2')$ and $pa_1(\alpha_1) = \tau)$

(ii) $\forall \alpha_2 \in \Sigma_2$, whenever $q_2 \stackrel{\alpha_2}{\rightarrow} q_2'$ then

- $(\exists q_1' \in Q_1, \alpha_1 \in \Sigma_1$ where $q_1 \stackrel{\alpha_1}{\Rightarrow}_{se} q_1'$ and $(q_1', q_2') \in S$ and $ps_1(q_1') = ps_2(q_2')$ and $pa_1(\alpha_1) = pa_2(\alpha_2))$, OR
- $(\exists q_1' \in Q_1$ where $q_1 \Rightarrow_{se} q_1'$ and $(q_1', q_2') \in S$ and $ps_1(q_1') = ps_2(q_2')$ and $pa_2(\alpha_2) = \tau)$

The two SELTS are said to be *weakly state-event equivalent*, denoted $\mathbb{Q}_1 \approx_{se} \mathbb{Q}_2$, iff there exists a weak state-event bisimulation for $\mathbb{Q}_1$ and $\mathbb{Q}_2$ such that $(q_{10}, q_{20}) \in S$.

4.2.1. Weak State-Event Model Reduction

We now define a projection from computations to weakly observed computations that deletes a state-event pair from a computation if the event output is an unobservable τ transition and the state output remains unchanged in the next state (i.e. there is no way to observe whether we remain in the current state or take the transition to the next state). Since weak state-event equivalence suppresses system information regarding sequences of unobservable events that do not cause state changes, the equivalence can only be used for model reduction with a restricted set of temporal formulas. This restricted class, which we will call the class of State-Event Stuttering-Invariant (SESI) formulas, is characterized as those formulas that are satisfied by a computation iff the projected computation satisfies the formula.

In [20] the authors use a state-based projection operator to develop a state-only version of weak satisfaction. They define the *reduced behavior* of a computation σ via a two step process that amounts to first applying ps to each state in the sequence and then replacing uninterrupted sequences of identical states with a single copy of the state. In our case we are dealing with sequences of state-event pairs rather than just sequences of states. We cannot simply apply $ps \times pa$ to each of the state-event pairs in the sequence and then replace subsequences of uninterrupted state-event output pairs by a single state-event output pair since in this case important information relating state changes and event observations would be lost.

Consider the three state-event sequences shown below where $tick$ is the event representing the passage of one second on the global clock.

$$(q_0,\tau)(q_0,\tau)(q_0,tick)(q_0,\alpha)(q_1,tick)\ldots$$
$$(q_0,\tau)(q_0,tick)(q_0,tick)(q_0,\alpha)(q_1,tick)\ldots$$
$$(q_0,tick)(q_0,\tau)(q_0,tick)(q_0,\tau)(q_0,\alpha)(q_1,tick)\ldots$$

If we assume that the output maps ps and pa are the identity map on their respective domains, then following [20] the first and second sequences would result in the same reduced computation:

$$(q_0,\tau)(q_0,tick)(q_0,\alpha)(q_1,tick)\ldots$$

while the third sequence is its own reduced computation. This would lead us to believe that in the first two cases the system delays for one second and then changes state from q_0 to q_1 via an α transition when, in fact, the second and third computations do not make the α transition until after 2 seconds. While we want our projection operator to distinguish the first case from the other two, the second and third computations differ only by unobservable transitions that do not change the state output. Upon rewriting the three sequences in terms of the notation of weak state-event observation equivalence, the differences and similarities in observed behaviors become apparent:

$$\left.\begin{array}{c} q_0 \xrightarrow{\tau} q_0 \xrightarrow{\tau} q_0 \xrightarrow{tick} q_0 \xrightarrow{\alpha} q_1 \xrightarrow{tick} \cdots \\ q_0 \xrightarrow{\tau} q_0 \xrightarrow{tick} q_0 \xrightarrow{tick} q_0 \xrightarrow{\alpha} q_1 \xrightarrow{tick} \cdots \\ q_0 \xrightarrow{\tau} q_0 \xrightarrow{tick} q_0 \xrightarrow{\tau} q_0 \xrightarrow{tick} q_0 \xrightarrow{\alpha} q_1 \xrightarrow{tick} \cdots \end{array}\right\} \mapsto \left\{\begin{array}{c} q_0 \overset{tick}{\Rightarrow}_{se} q_0 \overset{\alpha}{\Rightarrow}_{se} q_1 \overset{tick}{\Rightarrow}_{se} \cdots \\ q_0 \overset{tick}{\Rightarrow}_{se} q_0 \overset{tick}{\Rightarrow}_{se} q_0 \overset{\alpha}{\Rightarrow}_{se} q_1 \overset{tick}{\Rightarrow}_{se} \cdots \\ q_0 \overset{tick}{\Rightarrow}_{se} q_0 \overset{tick}{\Rightarrow}_{se} q_0 \overset{\alpha}{\Rightarrow}_{se} q_1 \overset{tick}{\Rightarrow}_{se} \cdots \end{array}\right.$$

To an external observer the second and third computations would produce the same observed state-event sequence: $(q_0,tick)(q_0,tick)(q_0,\alpha)(q_1,tick)\ldots$. The projection defined below has the effect of replacing all the state-event pairs making up an observed transition $q_1 \overset{\alpha}{\Rightarrow}_{se}$, with a single state-event pair $q_1 \xrightarrow{\alpha}$. The following weak state-event sequence projection operator produces a system's weakly observed computations.

Definition 4.3. Given a SELTS $\mathbb{Q}$ with state output map $pa : Q \to \mathcal{P}(AP)$, $ps : \Sigma \to \Sigma'$ and $\sigma = (q_0,\alpha_0)(q_1,\alpha_1)\ldots$, $\sigma \in \mathcal{M}(\mathbb{Q})$, the *weakly observed behavior* of σ is denoted, with a slight abuse of notation, by $\approx(\sigma)$ which is defined inductively as follows:

$$\begin{array}{rcl} \approx(q_0) & = & ps(q_0) \\ \approx(q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots q_n \xrightarrow{\alpha_n} q_{n+1}) & = & \left\{\begin{array}{l} \approx(q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots q_n), \textit{if } pa(\alpha_n) = \tau \wedge ps(q_n) = ps(q_{n+1}) \\ \approx(q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots q_n) \xrightarrow{pa(\alpha_n)} ps(q_{n+1}), \textit{otherwise} \end{array}\right. \end{array}$$

For C a set of computations, we define $\approx(C) := \{\approx(\sigma) : \sigma \in C\}$.

Example 4.1. In this example we consider the weak state-event observations generated by an SELTS with identity output maps $ps := id_Q$ and $pa := id_\Sigma$.

$$\begin{array}{rcl} \sigma_1 & = & (q_0,\tau)(q_0,\alpha)(q_0,\tau)(q_1,\tau)(q_1,\beta)(q_2,\alpha)\ldots = q_0 \xrightarrow{\tau} q_0 \xrightarrow{\alpha} q_0 \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_1 \xrightarrow{\beta} q_2 \xrightarrow{\alpha} \ldots \\ \approx(\sigma_1) & = & q_0 \xrightarrow{\alpha} q_0 \xrightarrow{\tau} q_1 \xrightarrow{\beta} q_2 \xrightarrow{\alpha} \ldots = (q_0,\alpha)(q_0,\tau)(q_1,\beta)(q_2,\alpha)\ldots \\ \sigma_2 & = & (q_0,\tau)(q_0,\tau)(q_0,\tau)\ldots = q_0 \xrightarrow{\tau} q_0 \xrightarrow{\tau} q_0 \xrightarrow{\tau} \ldots \\ \approx(\sigma_2) & = & q_0 = (q_0,-) \end{array}$$

In $\approx(\sigma_1)$ all the τ transitions are eliminated except for the $q_0 \xrightarrow{\tau} q_1$ transition since this τ transition can be inferred from the external observer's observation of a state change from q_0 to q_1 without any observed event. In this case we say that τ is an *implicitly observed transition*. The computation σ_2 is initially observed to be in state q_0 and then produces no state change or event observations. This is reflected in $\approx(\sigma_2)$ as $(q_0, -)$, the observed state output with no defined transition. Thus an infinite state-event sequence can result in a finite weakly observed sequence. This is why the effort was made earlier to extend the definition of temporal operators to finite as well as infinite sequences, allowing us to define weak satisfaction of temporal formulas below. We now consider those formulas with truth values that are robust with respect to unobservable τ transitions.

Definition 4.4. Given a state-event temporal formula F over the set of atomic predicates AP, we say that F is *State-Event Stuttering-Invariant* (SESI) if for all SELTS $\mathbb{Q}$ with state output map $P : Q \to \mathcal{P}(AP)$, for all computations $\sigma \in \mathcal{M}(\mathbb{Q})$, the following equation holds:

$$\sigma \models F \text{ iff } \approx(\sigma) \models F \tag{1}$$

Formulas composed solely of state predicates together with the $\vee, \wedge, \mathcal{U}, \mathcal{U}^{\alpha}_{[l,u]}$ operators (i.e. that do not contain the next operator $\bigcirc$ or next transition variable η) are SESI. Additionally, a formula of the form $\Box\Diamond(\eta = tick)$ is SESI since $\Diamond(\eta = tick)$ is SESI and $\Box F = \neg\Diamond\neg F$. The following theorem from [13, 14] allows us to model-check SESI formulas on a system's quotient system or other reduced equivalent systems and infer the result for the original system.

Theorem 4.2. Let F be an SESI formula. If $\mathbb{Q}_1, \mathbb{Q}_2$ are SELTS such that $\mathbb{Q}_1 \approx_{se} \mathbb{Q}_2$ then $\mathbb{Q}_1 \models \neg(\Box\Diamond\eta = tick) \vee F$ iff $\mathbb{Q}_2 \models \neg(\Box\Diamond\eta = tick) \vee F$.

Similar to the results for strong state-event equivalence, abstractions based upon weak state-event equivalence preserve truth values of SESI formulas under parallel composition [13, 14].

4.3. PVS Theories for State-Event Equivalences

A TTM can always be expanded to a (possibly infinite state) SELTS so we do not need to give another definition of equivalence for TTMs. We use the same definition as for SELTSs. Here we only briefly outline how the equivalences are formulated in PVS. A detailed description is provided in [32] and complete files are available online[1].

The theory `statetrans` provides a parameterized definition of of *state invariant transitive closure*. We use this definition in the theory `sesim`, which gives one direction in the definitions of strong and weak state-event bisimulations. The theory `sebisim` imports the theory `sesim` twice to give both directions in the definition of the bisimulation. The predicate `wsebisim?` identifies weak state-event bisimulations and is used to create the `Wsebisim` predicate subtype of weak state-event bisimulations. Similar definitions are provided for strong state-event equivalence.

In a similar fashion to the TTM templates of section 3.2 the PVS template for state-event bisimulation provides a straightforward environment for specifying state-event bisimulation in PVS. The template provides the option for the user to import reachability or other invariants to help in verifying the bisimulation relation. Then we define the event type `A` which is the common event type for the two structures.

[1] `http://sqrl.mcmaster.ca/~lawford/papers/FI05`

In our setting of weak equivalence, `tau` in PVS (representing the τ event) is always included in this set of events. The state type `state` (also called the state output) is the common state type for the two structures. We also need to instantiate the functions `pS1`, `pS2`, `pA1`, `pA2`. Predicates `Is_tau?`, `dd1` and `dd2` identify the unobservable events and the transition relations of the two structures. By importing the theory `sebisim` with concrete parameters from our specification, we get all the definitions for state-event bisimulation. The user supplies a relation `RR` and its type is specified as `Wsebisim` (`Sebisim`) if it is a weak (strong) state-event bisimulation between the two structures. The user will be required to prove the Type Correctness Condition automatically generated by PVS to confirm that `RR` is a weak (strong) state-event bisimulation. Finally, the lemma `weakequi` (`strongequi`) must be proved to confirm that the two initial states are related by `RR`, so that we can say the two structures are weakly (strongly) state-event equivalent.

5. Verification of a Real-time Controller

The Delayed Reactor Trip (DRT) system was first described in [15]. It is a typical example from the process control industry. Below we describe the system and its TTM models of its specification and implementation and refer the reader to [13, 15] for the details of how these models were obtained and validated. What is new in this work is the PVS models of these TTM and the verification of their equivalence using PVS.[2]

When the reactor pressure and power exceed acceptable safety limits in a specified way, we want the DRT control system to shut down the reactor. Otherwise, we want the control system to be reset to its initial monitoring state. The desired action for the Delayed Reactor Trip system has the following

Figure 5. Block Diagrams for the DRT

informal description: if the power exceeds the power threshold PT and the pressure exceeds the delayed set point DSP, then wait for 3 seconds. If after 3 seconds the power is still greater than PT, then open the relay for 2 seconds. The old implementation of the DRT using timers, comparators and logic gates is show in Figure 6.

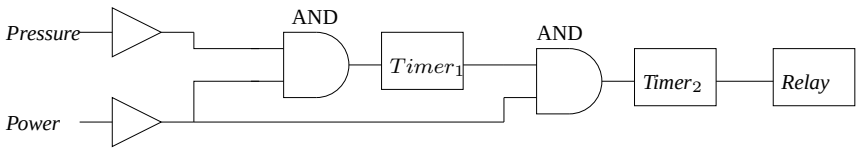


Figure 6. Analog Implementation of the Delayed Reactor Trip System

[2]Complete files for this example are available for download from `http://sqrl.mcmaster.ca/~lawford/papers/FI05`.

The hardware implementation is almost a direct translation of the above informal specification: When the reactor power and pressure exceed PT and DSP respectively, the comparators cause Timer1 to start. Timer1 times out after 3 seconds, sending a signal to one input of the second AND gate. The other input of the second AND gate is reserved for the output of the power comparator. The output of the second AND gate causes Timer2 to start if the power exceeds its threshold and Timer1 has timed out. Once Timer2 starts, it runs for 2 seconds while signaling the relay to remain open.

The new DRT system is to be implemented on a microprocessor system with a cycle time of 100ms. The system samples the inputs and passes through a block of control code every 0.1 seconds.

5.1. Formalizing the DRT Specification

By modeling the specification as a TTM (Figure 7), we can clarify the ambiguities in the informal specification and ensure that the input/output actions are completely determined. In order to verify the correctness of the microprocessor system, the DRT specification is put in a form that closely resembles the microprocessor system. A *tick* of the global TTM clock is assumed to be 100 ms, the scan period of the microprocessor. We assume proper filtering of the input signals and a sufficiently high sample rate. Thus the enablement conditions of a transition must be satisfied for at least one clock *tick* before the transition can occur. The transitions $(\mu, \alpha, \rho_1, \rho_2, \gamma)$ have lower and upper bounds of 1, exemplifying this filtering assumption.

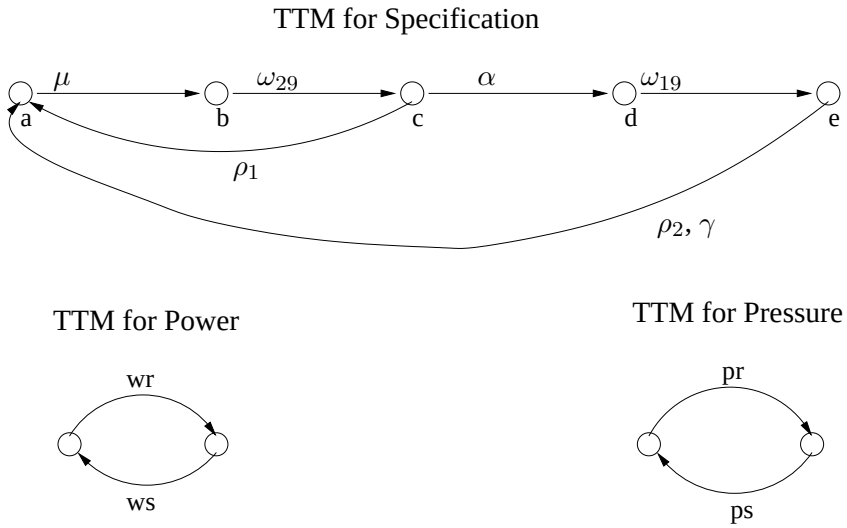
In the TTM, if the power and pressure exceed their corresponding thresholds, then the transition μ is enabled. After μ occurs, the system waits in activity *b* for 29 `ticks` (2.9 seconds) before proceeding to activity *c*. In activity *c*, the power level is checked again. If the power is still too high then the system opens the relay via transition α, otherwise the system resets via transition ρ_1 to go back to activity *a* and monitor power and pressure again. After transition α the system waits in activity *d* for 19 `ticks` (1.9 seconds) and then proceeds to *e*. At *e*, as an added safety feature [13], the system checks the power level again. If the power still exceeds the threshold, the system returns to activity *a* with the relay still open via transition γ, otherwise the system resets to *a* via ρ_2 while closing the relay. We model the pressure and the power as two separate simple TTMs (Figure 7).

With the help of the theories and template we defined in section 3, formalization of the TTM specification in PVS is very straightforward. We just follow the TTM representation of the specification in Figure 7 and input all the information into the template which we discussed in section 3.2.

We define the internal state as a record type: `s_internal_state: TYPE = [# Relay:bool, Power:bool, Pressure:bool #]`. To validate the PVS formalization, we formulated an invariant describing properties of the specification at all the reachable states. By proving the invariant, we confirm that our ideas about the TTM's behavior are correct. Later, by including the invariant in the definition of weak equivalence, we narrow down the state space needed by PVS to verify weak equivalence [32].

5.2. Formalizing the DRT Implementation

For the microprocessor DRT implementation each time the microprocessor passes through the block of code (originally represented by the pseudocode in [15]), it performs one of the group of operations identified in the TTM model by a transition name. The TTM for the implementation is obtained by replacing the "TTM for Specification" in Figure 7 with that shown in Figure 8, resulting in



$\mathcal{V} := \{x, Relay, Power, Pressure\}$

$\Theta := (x = a \wedge Relay = CLOSED \wedge Power = LO \wedge Pressure = LO)$

$\mathcal{T} := \{ \quad \mu := (Power = HI \wedge Pressure = HI, [\,], 1, 1)$

$\omega_{29} := (True, [\,], 29, 29)$

$\alpha := (Power = HI, [Relay : OPEN], 1, 1)$

$\omega_{19} := (True, [\,], 19, 19)$

$\rho_1 := (Power = LO, [\,], 1, 1)$

$\rho_2 := (Power = LO, [Relay : CLOSED], 1, 1)$

$\gamma := (Power = HI, [\,], 1, 1)$

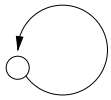
$wr := (Power = HI, [Power : LO], 1, \infty)$

$ws := (Power = LO, [Power : HI], 1, \infty)$

$pr := (Pressure = HI, [Pressure : LO], 1, \infty)$

$ps := (Pressure = LO, [Pressure : HI], 1, \infty)\ \}$

Figure 7. TTM for the DRT specification - *SPEC||POWER||PRESSURE*



Selfloop($\mu_1, \mu_2, \alpha, \beta, \gamma, \rho_1, \rho_2$)

$\mathcal{V}$:= $\{x, Relay, Power, Pressure, c_1, c_2\}$

Θ:=($x = a \wedge Relay = CLOSED \wedge Power = LO \wedge Pressure = LO$
$c_1 = 0, c_2 = 0$)

$\mathcal{T}$:= {

μ_1 := $(e_{\mu_1}, [c_1 : c_1 + 1], 1, 1)$

μ_2 := $(c_1 = 0 \wedge 1 \leq c_2 \leq 19, [c_2 : c_2 + 1, Relay : OPEN], 1, 1)$

α := $(Power = HI \wedge c_1 \geq 30, [c_1 : 0, c_2 : c_2 + 1, Relay : OPEN], 1, 1)$

ρ_1 := $(Power = LO \wedge c_1 \geq 30, [c_1 : 0], 1, 1)$

ρ_2 := $(Power = LO \wedge c_1 = 0 \wedge c_2 \geq 20, [c_2 : 0, Relay : CLOSED], 1, 1)$

γ := $(Power = HI \wedge c_1 = 0 \wedge c_2 \geq 20, [c_2 : 0], 1, 1)$ }

Where

e_{μ_1}:= $(Power = HI \wedge Pressure = HI \wedge c_1 = c_2 = 0)$
$\vee (1 \leq c_1 \leq 29)$

Figure 8. TTM for DRT implementation - $PROG$

PROG||POWER||PRESSURE. Since it directly models a cyclic executive, all the transitions for the microprocessor DRT implementation are modeled as selfloops.

As the microprocessor scans through the code each cycle (100 ms), it picks out one of the labeled sections of code. The section picked is the one whose TTM model transition enablement condition is satisfied. The microprocessor then loops back to the beginning and re-evaluates all the enablement conditions in the next cycle. So each transition, except for those which simulate the power or pressure, has a lower and upper time bound of 1.

Formalization and validation of the implementation in PVS is similar to that of the specification. This time the internal state is given in the TTM template as: `internal_state: TYPE=[# Relay:bool, Power:bool, Pressure:bool, c1:nat, c2:nat #]`. We use the two variables c_1 and c_2 to count the time requirements of ω_{29} and ω_{19} respectively in the specification. As can be seen in Figure 8, we do not have ω_{19} and ω_{29} in the set of actions. Note that the actions μ_1, α, μ_2, ρ_1, ρ_2 and γ all have enablement conditions involving c_1 and/or c_2.

As with the DRT specification, the verification of an invariant for the DRT implementation gives us confidence that the TTM is doing what we want and narrows down the state space needed by PVS to verify weak equivalence.

5.3. Weak Equivalence Verification

With the help of the theories and template described in section 4, we can define weak equivalence between the specification and implementation in PVS. The functions $ps1$ and $ps2$ map their respective state records to the system inputs and output: Power, Pressure and Relay. In the event output functions $pa1$ and $pa2$, $tick$ is mapped to itself to preserve timing information and all other actions are mapped to

unobservable τ events.

We need to prove the Type-correctness condition (TCC) automatically produced by this definition in PVS. By proving the TCC which requires the relation `RR` to be a type of `Wsebisim`, we conclude that the relation `RR` is a weak state-event bisimulation between these two TTMs. By proving the lemma `weakequi` which implies that the two initial states are related by `RR`, we conclude these two TTMs are weakly equivalent.

The current equivalence proof takes slightly less than 48 minutes of CPU time on a dual 2.4 GHz Xeon machine with 4 GB of RAM running RedHat Linux 9.0. Constructing the actual proof took considerably longer and required effort to decompose the proof into several lemmas to deal with memory limitations. This represents an initial brute force effort to complete the proof. With the further development of custom bisimulation proof strategies it should be possible to reduce the time and effort required for similar proofs to a more reasonable level. The integration of the new Integrated Canonical Solver (ICS) [6] decision procedures in PVS and the planned capability to export from SAL (the Symbolic Analysis Library) model-checking environment into PVS [7] may provide a means for more efficient analysis of large TTM verification. The TTMs could be first debugged in SAL and then exported to PVS for equivalence verification where the ICS decision procedures can be used.

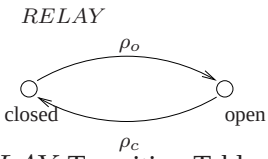
5.4. Model Checking the DRT

In [12, 15] the DRT verification problem was deemed to be solved, in effect, as soon as $prog$ was verified to be weakly state-event equivalent to $spec$. While the equivalence verification process proved to be useful (an error in the original pseudocode was found and fixed in [12]), the problem with such equivalence verification techniques is that while the implementation has been verified, its correct operation still depends upon the abstract specification model correctly capturing the desired system properties. An equivalent implementation is only as good as its specification. How can one verify that the original specification was correct? Is there any guarantee that the equivalence used in the verification process preserves the relevant system properties?

For the DRT we will attempt to state some desired system properties as SESI temporal logic formulas. By verifying the temporal logic specification formulas on the DRT specification employing *SPEC* using model-checking, the satisfaction preserving properties of weak state-event equivalence will guarantee that the property holds in any equivalent implementation. To validate the results, each temporal logic formula that is model-checked on the specification will also be model-checked on the equivalent implementation. Verification of the detailed implementations provides some empirical confirmation of the correctness of Theorem 4.2, and also illustrates the computational benefits of using reduced models for verification purposes.

5.4.1. Refining the Reactor Model

Before model-checking our DRT design we complete our model of the reactor system interacting with our controller model (*SPEC* or *PROG*) by adding the subsystem in Fig. 9 to model the reactor's shutdown relay state. We assume that $RELAY$'s activity variable x_{RELAY} represents the current state of the reactor's relay. Any change to the value of the variable $Relay$ by $SPEC$ (or $PROG$) causes an "instantaneous" change in X_{RELAY} (i.e. before the next clock $tick$, provided $Relay$'s value remains at the new value) so that after ρ_o or ρ_c occurs $X_{RELAY} = Relay$.



$RELAY$ Transition Table

$$\begin{aligned}
\Theta &:= x_{RELAY} = closed \wedge Relay = \text{CLOSED} \\
\rho_o &:= (Relay = \text{OPEN}, [\,], 0, 0) \\
\rho_c &:= (Relay = \text{CLOSED}, [\,], 0, 0)
\end{aligned}$$

Figure 9. $RELAY$ - TTM model of the shutdown relay.

Although $RELAY$ provides the possibility of non-Zeno behavior, an infinite number of successive non-$tick$ transitions, this would require non-Zeno behavior of the input variable $Relay$. In both $SPEC$ and $PROG$, all TTM transitions have lower time bounds ≥ 1 and so each can only perform a finite number of transitions between successive clock $tick$s. Thus the composite system is guaranteed to have an infinite number of $tick$s in all computations and hence $control \| plant \models \Box\Diamond(\eta = tick)$ for *CONTROL* $\in \{SPEC, PROG\}$. Therefore we may drop the $\neg\Box\Diamond(\eta = tick)$ disjunction that occurs in Theorem 4.2 since it is false for all computations of *CONTROL*$\|$*PLANT* where *PLANT* $=$ *POWER*$\|$*PRESSURE*$\|$*RELAY*.

5.4.2. Model-Checking Details

In the following model-checking results we will say that a real-time temporal logic formula F has been model-checked or verified for a given timed system when, in fact, we have verified an untimed temporal logic formula F' on the untimed system that incorporates timer variables and additional TTM transitions to "observe" the timed property. The construction of F' and the TTM transitions to be added to the system before it is translated into SAL input can be difficult. Often the untimed model-check will fail to capture precisely the desired real-time behavior but may verify something close enough to the original real-time behavior to suit the designer's purposes. Below we assume that the untimed model-checks are "close enough" when stating that a timed property has been verified by the untimed model-check. In the absence of a powerful model-checking tool for RTTL, the "untimed" model-checks will have to suffice to illustrate our model reduction theory.

The TTMs of the plant and controller systems were translated from PVS into SAL as described in section 3.3. All of the model checking results below are for the a beta version of SAL 2.4 running on a dual 2.4 GHz Xeon machine with 4 GB of RAM running Linux kernel version 2.6.6. The model-checking results are shown in Table 1 where `exec.time` is the time (in seconds) from invoking the checker to termination of the process, including compiling the symbolic transition relation, whereas `verif.time` only includes the time to verify the particular formula after the symbolic representation of transition system has been constructed.

5.4.3. Verification of System Response

This subsection demonstrates that specification models and formulas do not always embody the properties one initially thinks they capture. The first property we would like to check for our specification module, and hence the implementation module, is correct response to stimulae from the plant. The informal DRT system requirements may be restated in a form more suggestive of a Temporal Logic translation as:

> Henceforth, if $Power$ and $Pressure$ simultaneously exceed their threshold values for at least 2 $tick$s and 30 $tick$s later $Power$ exceeds its threshold for another 2 $tick$s, then within 30 to 32 $tick$s open the reactor relay for at least 20 $tick$s.

In the rephrased informal specification we have added "at least 2 $tick$s" requirements to ensure that the DRT has time to react to the changes to its input.

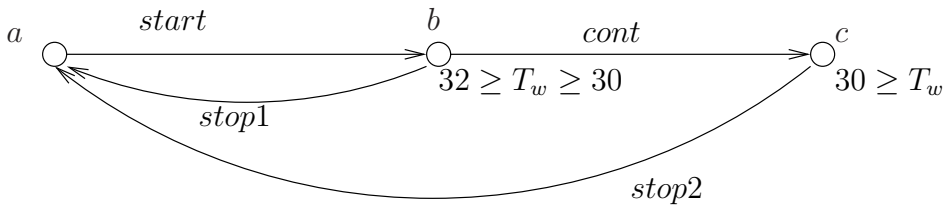
We call our temporal logic translation of this formula the System Response formula, F_{Res}:

$$\Box[\Box_{<2}(Power \geq \text{PT} \wedge Pressure \geq \text{DSP}) \wedge \Diamond_{30}\Box_{<2}Power \geq \text{PT} \rightarrow \Diamond_{[30,32]}\Box_{<20}x_{RELAY} = open]$$

The first $\Box$ operator with the square braces around the rest of the formula says that the property contained within holds in the initial state of the computation and at all later points (all suffixes) of the computation. For a formula F, $\Diamond_{[30,32]}F$ is shorthand notation for $true\,\mathcal{U}_{[30,32]}F$ which translates directly as "eventually after at least 30 but no more than 32 $tick$s, F is true". $\Diamond_{30}F$ and $\Box_{<2}F$ are used to denote $\Diamond_{[30,30]}F$ and $\neg\Diamond_{[0,1]}\neg F$. We can paraphrase $\Box_{<2}F$ as "From now until 2 $tick$s have occurred, F holds".

The SAL model checker does not explicitly support the simple real-time temporal logic described in section 2.3. Thus in order to verify the real-time aspects of F_{Res} we will add the timer variable T_r to the system to time how long $RELAY = open$. We assume that initially $T_r = 0$. The operation functions of ρ_o and ρ_c become $[cd(T_r, 20)]$ and $[stop(T_r)]$ respectively. Here $cd(T_r, 20)$ in the operation function of ρ_o has the effect of initializing T_r to a value of 20 whenever x_{RELAY} changes from $closed$ to $open$. T_r will then count down with each $tick$ until it reaches a value of 0 or is halted at its current value via the $stop(T_r)$ operation. Thus if $T_r = 0$ and $x_{RELAY} = open$, the reactor relay has been open for 20 $tick$s. The addition of the T_r operations to $RELAY$ will allow the untimed system to "observe" the $\Box_{<20}x_{RELAY} = open$ part of F_{Res}. The rest of the formula will be dealt with in the untimed system by an additional "property observer" TTM RES (see Figure 10) that will run in parallel with the rest of the system.

When $Power$ and $Pressure$ simultaneously exceed their threshold values, the ψ_{start} transition of RES starts the timer T_w counting down from 32. If $Power$ or $Pressure$ drop below their threshold values before two $tick$s of the the clock have occurred (i.e. before $T_w = 30$) then ψ_{stop1} occurs, stopping timer T_w. If T_w counts down to 30 then $\Box_{<2}Power \geq PT \wedge Pressure \geq DSP$ is true. Transition ψ_{cont} occurs to "observe" this fact. We then wait to check the power when $0 \leq T_w \leq 2$ (30 to 32 $tick$s after $Power$ and $Pressure$ first exceeded their threshold values). If during that time $Power < PT$, then the $\Diamond_{30}\Box_{<2}x_{RELAY}Power \geq PT$ conjunct in the antecedent of F_{Res} is violated so RES resets via ψ_{stop2}, stopping T_w. On the other hand, if RES is in activity c and $T_w = 0$, then $\Diamond_{30}\Box_{<2}Power \geq PT$ is $true$ and previously $\Box_{<2}Power \geq PT \wedge Pressure \geq DSP$ was true since ψ_{cont} occurred to bring us to c in the first place. Thus we will approximate the antecedent of F_{Res} by $T_w = 0 \wedge x_{RES} = c$. Combining



RES Transition Table

$$
\begin{aligned}
\Theta &:= x_{RES} = a \wedge Power = \text{LO} \wedge Pressure = \text{LO} \wedge T_w = 0 \\
\psi_{start} &:= (Power \geq PT \wedge Pressure \geq DSP, [cd(T_w, 32)], 0, 0) \\
\psi_{stop1} &:= (Power < PT \vee Pressure < DSP, [stop(T_w)], 0, 0) \\
\psi_{stop2} &:= (Power < PT \wedge 0 \leq T_w \leq 2, [stop(T_w)], 0, 0) \\
\psi_{cont} &:= (T_w = 30, [\,], 0, 0)
\end{aligned}
$$

Figure 10. RES – TTM Observer for F_{Res} used in creating untimed formula F'_{Res}.

the above observations we have the untimed formula F'_{Res} that we will model-check with SAL:

$$\Box[(T_w = 0 \wedge x_{RES} = c) \rightarrow \Diamond(x_{RELAY} = open \wedge T_r = 0)]$$

Now that we have the formula F'_{Res} without timed operators we translate our system with the additional counter variables and property observer TTM into SAL input and model-check F'_{Res} in place of property F_{Res}. The results of attempting to verifying F'_{Res} with appears in Table 1. We conclude *SPEC*$\|$*PLANT* $\not\models F_{Res}$ and *PROG*$\|$*PLANT* $\not\models F_{Res}$. The computational results are summarized in Table 1. The counterexample computation generated by SAL reveals why our system specification model, implementation model, and indeed the original hardware implementation, all fail to satisfy this property.

While Timer 1 is running (*SPEC* is in activity b or $PROG$ has a non-zero value of c_1), the system is effectively ignoring its inputs. Consider the possible input timing diagram in Figure 11. $Power$ and

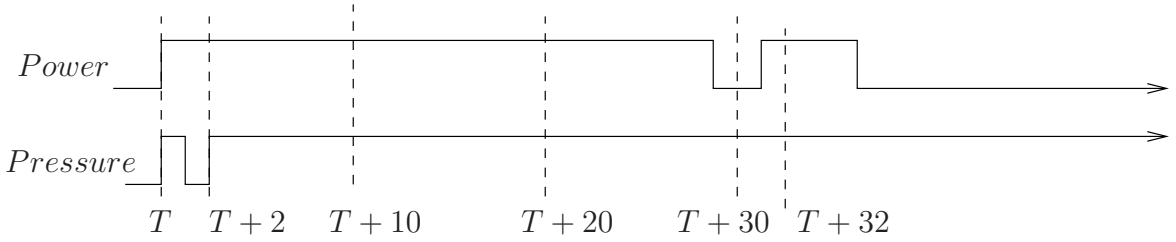


Figure 11. Input sequence generating a counter example to F_{Res}

$Pressure$ simultaneously exceeding their threshold values at time T will cause Timer 1 to start but at time $T + 30$, $Power$ = LO so the $Relay = open$ "signal" is not sent and the system goes back

to monitoring its inputs. However, while Timer 1 was running, at $T + 2$ $Power$ and $Pressure$ also exceeded their threshold values and 30 $tick$s later at time $T + 32$ $Power$ is exceeding its threshold. Because Timer 1 was already running at $T + 2$ in response to the conditions at time T, it is unable to respond to the conditions at $T + 10$. The system therefore has no way of knowing that it should check the value of $Power$ at time $T + 32$ and consequently open the relay.

While it is possible to design a relatively simple software implementation that does satisfy F_{Res} through the use of registers as bit arrays, for illustrative purposes we will assume that we are trying to design a software system that provides similar input/output behavior to the original system. In this case F_{Res} is an inappropriate temporal logic specification. Changing the antecedent of F_{Res} to require that the DRT controller be in its initial state (i.e. neither timer is running) when $Power$ and $Pressure$ exceed their threshold values, we can alter F_{Res} to obtain a formula capturing the behavior of the original system. We call this new property the Initialized System Response formula, F_{IRes}:

$$\Box[\Theta_{CONTROL} \wedge \Box_{<2}(Power \geq \text{PT} \wedge Pressure \geq \text{DSP}) \wedge \Diamond_{30}\Box_{<2}Power \geq \text{PT} \rightarrow \Diamond_{[30,32]}\Box_{<20}X_{RELAY} = open]$$

Here $\Theta_{CONTROL} := \Theta_{SPEC}$ or $\Theta_{CONTROL} := \Theta_{PROG}$ depending on whether we are model-checking control *SPEC* or control *PROG*.

The untimed formula F'_{Res} used in place of F_{Res} can be used as the untimed formula F'_{IRes} to model-check in place of F_{IRes} provided we modify the property observer TTM RES. We add the $\Theta_{control}$ conjunct to the enablement condition of ψ_{start} to obtain the new property observer TTM $IRES$. Thus the new enablement condition for ψ_{start} is $\Theta_{CONTROL} \wedge Power \geq PT \wedge Pressure \geq DSP$.

The results of model-checking F'_{IRes} with its observer system are also contained in Table 1. They show that both the specification and implementation satisfy F'_{IRes}.

The above pair of model checking results have helped us to gain a deeper understanding of the behavior of our system and, by the agreement of results for the use of *SPEC* and *PROG* as the control, have illustrated Theorem 4.2. We will have more to say about the results regarding the space (number of states) and time requirements in section 5.4.5.

5.4.4. Verification of System Recovery

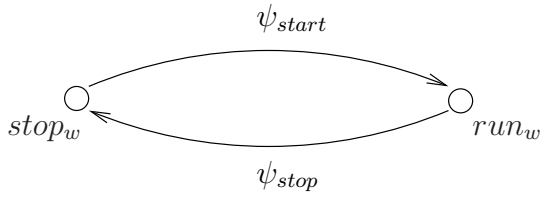
In the original hardware implementation a signal to open the reactor relay is only sent during the 2 seconds that Timer 2 is running. As an added safety feature in our microprocessor design, *SPEC* was set up to continue sending the $X_{Relay} = open$ signal until $Power$ was no longer exceeding its threshold. Since the DRT is but one of many reactor control systems operating in the actual reactor, a reasonable requirement might be that the closed-loop system "recover" in a timely fashion after the $Relay =$ OPEN signal has been sent for at least 20 $tick$s (2 seconds) and $Power$ returns to normal operating levels. An informal statement of this property might be:

> Henceforth if $x_{RELAY} = open$ for the next 20 $tick$s and after the 20th $tick$ $Power < PT$ for at least 2 $tick$s, then before the 22nd $tick$ $X_{RELAY} = closed$.

We translate this statement into the System Recovery formula F_{Rec}:

$$\Box[(\Box_{<20}X_{RELAY} = open \wedge \Diamond_{20}\Box_{<2}Power = LO) \rightarrow (\Diamond_{<22}X_{RELAY} = closed)]$$

As we did for F_{Rec}, we can use the addition of the timer T_r to $RELAY$ to check the subproperty $\Box_{<20} X_{RELAY} = open$. Again the remainder of the formula will be handled by a property observer TTM. Figure 12 contains REC, the TTM property observer for F_{Rec}.



REC Transition Table

$$\begin{aligned} \Theta &:= x_{REC} = stop \wedge T_w = 0 \wedge x_{RELAY} = closed \wedge Power = \text{LO} \\ \psi_{start} &:= (x_{RELAY} = open \wedge T_r = 0 \wedge Power < PT, [cd(T_w, 2)], 0, 0) \\ \psi_{stop} &:= (x_{RELAY} = closed \vee Power \geq PT, [stop(T_w)], 0, 0) \end{aligned}$$

Figure 12. REC – TTM Observer for F_{Rec} used in creating untimed property F'_{Rec}.

The transition ψ_{start} occurs once the reactor relay has been open for 20 $tick$s ($x_{RELAY} = open \wedge T_r = 0$) and $Power$ is LO ($Power < PT$). It starts timer T_w counting down from an initial value of 2. If $Power$ becomes HI or the reactor relay closes, transition ψ_{stop} takes place, immediately stopping the timer T_w and returning REC to activity $stop$. Thus if REC is in activity run and $T_w = 0$ then the reactor relay has been open for 20 $tick$s, and subsequently $Power$ has been LO for more than 2 clock $tick$s. This is a violation of F_{Rec}. Therefore we can reduce model-checking the timed property F_{Rec} to model-checking the untimed safety property F'_{Rec}:

$$\Box\neg(T_w = 0 \wedge x_{REC} = run)$$

Thus F'_{Rec} says that it is never the case that $T_w = 0$ when TTM REC is in activity run.

While it seems plausible that our current *SPEC* and *PROG* will force the closed loop system to satisfy F_{Rec}, model-checking proves the contrary (see Table 1). The counterexamples generated by SAL show that the γ transitions of *SPEC* and *PROG* are at the root of the closed-loop systems' failures to meet the recovery specification.

Consider the TTM *SPEC* in Figure 7 Activity e is where the value of $Power$ is reevaluated after $Relay =$ OPEN has been true for the required 20 $ticks$ in activity d. If $Power \geq PT$ then $SPEC$ returns to activity a via transition γ, leaving $Relay =$ OPEN. The system can now only "recover" by returning to activity e when shortly after $Power$ returns to an acceptable level and then executing a ρ_2 transition that sets $Relay =$ CLOSED.

Removal of the γ transition will ensure that $SPEC$ remains at activity e until $Power < PT$. If $Power$ is less than PT while $SPEC$ is in e, then before two clock $tick$s ρ_2 occurs, setting $Relay =$ CLOSED, and thereby ensuring satisfaction of F_{Rec}. With the removal of γ, the only way that $SPEC$ can enter a when $Relay =$ OPEN is via ρ_2. We will also delete the γ from $PROG$. Call the revised systems formed by the elimination of their γ transitions $SPEC_r$ and $PROG_r$, respectively. While the

new systems are smaller and perhaps agree more closely with the designer's intuition of how the system should behave, changing the systems brings into question their equivalence and the satisfaction of the Initialized Response formula F_{IRes}, while creating the possibility that the closed-loop system will now satisfy F_{Rec}.

From Table 1 we see that $SPEC_r \| PLANT \models F_{Rec}$ and $PROG_r \| PLANT \models F_{Rec}$. Further model-checks also confirm that $SPEC_r \| PLANT \models F_{IRes}$ and $PROG_r \| PLANT \models F_{IRes}$. This mutual satisfaction of F_{Rec} and F_{IRes} by *SPEC* and *PROG* was not merely accidental. It was forced by Theorem 4.2 because $SPEC_r \approx_{se} PROG_r$. Given the simple structure of the systems and the one-to-one correspondence between γ transitions in *SPEC* and *PROG*, the PVS proof of $SPEC \approx_{se} PROG$ can be modified to provide a proof of $SPEC_r \approx_{se} PROG_r$, though this has not yet been attempted in PVS.

5.4.5. Model-Checking Concurrent Controllers

So far we have typically seen a factor of 1.5-2 times improvement in both the total model-checking execution time and states and a 2-3 times improvement in verification time, by using the abstract *SPEC* model instead of the full *PROG* model. If this were always the case it would be hard to justify the additional complexity of the equivalence verification in PVS or the possibility of using an $O(n^3)$ algorithm for weak state-event equivalence model reduction computation. More significant gains from our model reduction technique can be made when there are multiple controllers running in parallel. Each controller module is identical. Therefore, because of the compositional consistency of weak state-event equivalence for TTM modules, the model reduction computation or proof need only be performed once for a single controller module. The reduction can be used for each controller module added to the system providing a multiplicative effect in the reduction of the state size without any additional computational or manual effort. To illustrate the preceding concept, this section extends the basic DRT closed-loop system to the case when we have a redundant controller.

Two copies of our revised DRT controllers are run in parallel with the plant. The enablement conditions of the plant's $RELAY$ transitions are changed to accommodate the additional controllers and the plant module's interface is modified accordingly. We attempt to verify F_{IRes} and F_{Rec} for compositions of the reduced and unreduced revised DRT models. The results begin to demonstrate that the real benefits of compositional model reduction are realized when multiple reduced models are composed. The TTMs $SPEC_r$ and $PROG_r$ can have their transitions and internal and output variables subscripted by integers $i = 1, 2$ to avoid transition label and variable name conflicts. In interfacing the plant with the two controllers we assume that the plant will only change the state of the reactor relay x_{RELAY} when both controllers are in agreement. To accomplish this we modify the $RELAY$ TTM of Figure 9. The results of model-checking are shown in Table 1.

We see that for the system $control_1 \| control_2 \| PLANT_2$ the verification of property F_{IRes} for the reduced $control_i := SPEC_{r_i}$ case required roughly an order of magnitude less time and space to obtain the same result as model-checking the detailed implementation $control_i := PROG_{r_i}$ case.

The results of the model-check for the somewhat simpler property F_{Rec} show a definite improvement in the time and space required to decide the property using the reduced models. The answer is somewhat unexpected. While operating in the single control environment both $SPEC_r$ and $PROG_r$ result in closed loop systems that satisfy F_{Rec} but when run concurrently with another control, the closed-loop system fails to satisfy F_{Rec}. The counterexamples generated by SAL show that controllers can get out of synchronization from their e states. If $Power < PT$ while $Pressure \geq DSP$ then the following

state-event sequence can occur in $SPEC_{r_1} \| SPEC_{r_2} \| PLANT_2$:

$$(\text{LO}, \text{HI}, e, e) \overset{tick}{\rightarrow} (\text{LO}, \text{HI}, e, e) \overset{\rho_{2_1}}{\rightarrow} (\text{LO}, \text{HI}, a, e) \overset{ws}{\rightarrow} (\text{HI}, \text{HI}, a, e) \overset{tick}{\rightarrow} \ldots$$

The 4-tuples represent the value of the variables $(Power, Pressure, x_{SPEC_{r_1}}, x_{SPEC_{r_2}})$. We see that once $Power = \text{LO}$ for one tick, module $SPEC_{r_1}$ reacts, but before $SPEC_{r_2}$ can react, ws occurs setting $Power = \text{HI}$ and disabling ρ_{2_2}. The two systems are now out of synchronization and the situation deteriorates from there to a point where the reactor relay, once opened for more than 20 $tick$s will in some cases not close even if $Power < PT$ for up to 19 $tick$s! At first one might think the failure of the 2 controller system to satisfy F_{Rec} is the result of the lower time bounds of 1 on the reactor transitions wr, ws, pr and ps but putting reactor outputs through a low pass filter to increase the lower bounds up to at least 19 would still fail to eliminate all possible counterexamples.

Formula	$control$	result	exec.time	verif.time	states
$\mathbf{F}_{Res}$ - System Response	*SPEC*	fail	508	308	75759
	PROG	fail	915	530	141147
	$SPEC_r$	fail	244	139	50490
	$PROG_r$	fail	415	215	93490
$\mathbf{F}_{Rec}$ - System Recovery	*SPEC*	fail	55	3	931
	PROG	fail	100	10	1656
	$SPEC_r$	pass	44	2	920
	$PROG_r$	pass	80	6	1632
	$SPEC_{r1} \| \| SPEC_{r2}$	fail	320	56	53846
	$PROG_{r1} \| \| PROG_{r2}$	fail	652	58	105076
$\mathbf{F}_{IRes}$ - Initialized System Response	*SPEC*	pass	100	13	3167
	PROG	pass	300	24	6102
	$SPEC_r$	pass	77	4	1095
	$PROG_r$	pass	150	43	3748
	$SPEC_{r1} \| \| SPEC_{r2}$	pass	347	67	58927
	$PROG_{r1} \| \| PROG_{r2}$	pass	3117	2039	1088245

Table 1. Summary of SAL model-checking results of control$\|\|$*PLANT*

For all of the results in the above table, properties checked were specified in LTL logic and model-checked using SAL's symbolic model-checker.

5.4.6. Model-checking with UPPAAL

UPPAAL is a toolbox for verification of real-time systems. Systems are modeled as networks of timed automata extended with bounded integer variables, structured data types, and channel synchronization

[5]. In contrast to TTMs discrete model of time, UPPAAL uses a continuous time model. The query language of UPPAAL is a subset of CTL (Computation Tree Logic). UPPAAL model checker employs 'on-the-fly' search technique and automatically generates counterexamples, that can be imported into the simulator with graphical visualization [11].

We translated the DRT's TTMs into the UPPAAL timed automata and model-checked the response and recovery properties. As one might have expected, the real-time model-checking tool, UPPAAL proved to be much more efficient than our direct implementation of TTMs in the general purpose SAL model-checker. All the properties were checked in less than 5 seconds (results are in the Table 2), except for the F_{IRes} property, for which the verification took 100 seconds for $SPEC_{r1}||SPEC_{r2}$, and 260 seconds for $PROG_{r1}||PROG_{r2}$ system. The current version of UPPAAL does not offer the information on the size of the state space generated during verification so this information is missing from the table.

However, translation of the TTMs into the timed automata of UPPAAL turned out to be much more time consuming compared to modeling in SAL. The similar syntax for PVS and SAL input files made the task of translating the instantiated PVS templates for the TTMs into SAL input files straightforward.

Formula	$control$	result	exec.time
$\mathbf{F}_{Res}$ - System Response	*SPEC*	fail	0.2
	PROG	fail	1
	$SPEC_r$	fail	0.3
	$PROG_r$	fail	0.3
$\mathbf{F}_{Rec}$ - System Recovery	*SPEC*	fail	0.15
	PROG	fail	0.4
	$SPEC_r$	pass	0.5
	$PROG_r$	pass	1
	$SPEC_{r1}\|\|SPEC_{r2}$	fail	2.3
	$PROG_{r1}\|\|PROG_{r2}$	fail	0.1
$\mathbf{F}_{IRes}$ - Initialized System Response	*SPEC*	pass	2.5
	PROG	pass	6
	$SPEC_r$	pass	1.5
	$PROG_r$	pass	4
	$SPEC_{r1}\|\|SPEC_{r2}$	pass	100
	$PROG_{r1}\|\|PROG_{r2}$	pass	260

Table 2. Summary of UPPAAL model-checking results of control||*PLANT*

6. Conclusion

The paper gives the definitions of strong and weak state-event equivalences for SELTSs and TTMs in PVS. It also provides the beginnings of a unified modeling environment for SELTSs and TTMs in PVS and SAL which allows the user to specify and verify TTMs more easily. Further, it illustrates the use of the TTM modeling environment and describes how the TTM and equivalence theories have been used to formalize and verify the correctness of an industrial real-time controller.

The model-checking results applied to the same illustrate the correctness of Theorem 4.2. Weakly state-event equivalent systems did satisfy the same formulas on all their computations in which time advances. The benefits of compositionally consistent weak model reduction have been partially demonstrated by the multiple controller model-checking results.

The superior counter example generation features of the SAL model-checker were particularly useful in debugging the system. The counter examples from the failed model-checks of the DRT system illuminated system behavior that otherwise may not have been considered in the system design. The model-checking in turn benefited from the compositionally consistent equivalence verification technique as it provided a means of compositionally consistent model reduction. In the case of the DRT design, the combination of equivalence verification and model-checking were mutually beneficial, leading to a better design than would have been achieved by the application of either method in isolation.

6.1. Limitations and Future Research

Currently in our modeling environment, composition of TTMs must be done manually by the user before entry into the TTM template. By formalizing TTM composition in PVS and SAL we should be able to use these tools to compose the TTMs and prove properties of composite TTMs.

The equivalence proof for the DRT was done interactively. It required significant user interaction. In the future, we plan on developing prover strategies to largely automate the proof procedure. Closer integration with PVS's new decision procedures and alternate algebraic formulations of the equivalence should reduce the effort required to produce equivalence proofs. We should note that the theories are designed so that alternative equivalence relations can be easily applied. Further, the theorem proving capabilities of PVS can allow us to verify infinite state systems and the equivalence of whole classes of systems (e.g., for parameterized time bounds or even operation functions). We believe that tighter integration of the theorem prover and model checker may offer the best solution. Plans for a SAL to PVS export capability outlined in [7] hold out significant hope in this regard.

The significantly faster performance of UPPAAL on a subset of the temporal logic verification indicates that integration of the theorem proving capabilities of PVS with a specialized real-time model-checker would allow the verification of significantly larger problems, at a cost of a more difficult translation between the prover and the model-checker.

Acknowledgments

The authors wish to thank the ACSD04 program committee and anonymous reviewers for their constructive comments, Ryszard Janicki for his encouragement and help in reviewing Hong Zhang's thesis, and Jeff Zucker for his expert co-supervision and comments on Hong Zhang's thesis. The Fundamenta Informaticae reviewers provided valuable feedback through their constructive criticisms.

References

[1] Alur, R., Henzinger, T. A., Mang, F. Y. C., Qadeer, S., Rajamani, S. K., Tasiran, S.: MOCHA: Modularity in Model Checking, *Computer Aided Verification*, 1427, 1998.

[2] Archer, M.: TAME: Using PVS Strategies for Special-Purpose Theorem Proving, *Annals of Mathematics and Artificial Intelligence*, **29**(1-4), 2000, 201–232.

[3] Archer, M., Heitmeyer, C., Riccobene, E.: Proving Invariants of I/O Automata with TAME, *Automated Software Engg.*, **9**(3), 2002, 201–232, ISSN 0928-8910.

[4] Arnold, A.: *Finite Transition Systems*, Prentice Hall, 1994.

[5] Behrmann, G., David, A., Larsen, K. G.: A Tutorial on UPPAAL, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004* (M. Bernardo, F. Corradini, Eds.), number 3185 in LNCS, Springer–Verlag, September 2004.

[6] Filliâtre, J.-C., Owre, S., Rueß, H., Shankar, N.: ICS: Integrated Canonizer and Solver., *CAV* (G. Berry, H. Comon, A. Finkel, Eds.), 2102, Springer, 2001, ISBN 3-540-42345-1.

[7] Formal Methods Program: *Formal Methods Roadmap: PVS, ICS, and SAL*, Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003.

[8] Graf, S., Loiseaux, C.: Property Preserving Abstraction under Parallel Composition, in: *TAPSOFT'93* (M.-C. Gaudel, J.-P. Jouannaud, Eds.), number 668 in LNCS, Springer-Verlag, 1993, 644–657.

[9] Heitmeyer, C., Archer, M.: *Mechanical Verification of Timed Automata: A Case Study*, Technical Report 5546-98-8180, Naval Research Laboratory, April 1998.

[10] Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A Toolset for Specifying and Analyzing Software Requirements, *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, 1427, Springer, 1998, ISSN 0302-9743.

[11] Larsen, K. G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell, *International Journal on Software Tools for Technology Transfer*, **1**(1-2), 1997, 134–152.

[12] Lawford, M.: *Transformational Equivalence of Timed Transition Models*, Master Thesis, Dept. of El. Eng., Univ. of Toronto, Canada, January 1992.

[13] Lawford, M.: *Model Reduction of Discrete Real-Time Systems*, Ph.D. Thesis, Dept. of Elec. & Comp. Eng., Univ. of Toronto, Canada, January 1997.

[14] Lawford, M., Ostroff, J., Wonham, W.: Model reduction of modules for state-event temporal logics, in: *Formal Description Techniques IX: Theory, application and tools, Proceedings of FORTE/PSTV'96* (R. Gotzhein, J. Bredereke, Eds.), Chapman & Hall, 1996, 263–278.

[15] Lawford, M., Wonham, W.: Equivalence Preserving Transformations of Timed Transition Models, *IEEE Transactions Automatic Control*, **40**, July 1995, 1167–1179.

[16] Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification, *Proc. of 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.

[17] Lynch, N., Tuttle, M.: An Introduction to Input/Output automata, *CWI-Quarterly*, **2**(3), September 1989, 219–246.

[18] Lynch, N., Vaandrager, F.: Forward and backward simulations II.: timing-based systems, *Inf. Comput.*, **128**(1), 1996, 1–25, ISSN 0890-5401.

[19] Manna, Z., Bjorner, N., Browne, A., Chang, E. Y., Colon, M., de Alfaro, L., Devarajan, H., Kapur, A., Lee, J., Sipma, H., Uribe, T. E.: STeP: The Stanford Temporal Prover, *TAPSOFT*, 1995.

[20] Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York, 1992.

[21] Milner, R.: *Communication and Concurrency*, Prentice Hall, New York, 1989.

[22] de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2, *Computer Aided Verification: 16th International Conference, CAV 2004* (R. Alur, D. A. Peled, Eds.), 3114, Springer-Verlag Heidelberg, Boston, MA, USA, July 2004.

[23] Ostroff, J.: *Temporal Logic for Real-Time Systems*, RSP, Research Studies Press / Wiley, 1989, Taunton, UK.

[24] Ostroff, J.: Deciding properties of timed transition models, *IEEE Transactions Parallel and Distributed Systems*, **1**(2), April 1990, 170–183.

[25] Ostroff, J.: A visual toolset for the design of real-time discrete-event systems, *IEEE Transactions on Control Systems Technology*, **5**(3), 1997, 320–337.

[26] Ostroff, J., Wonham, W.: A Framework for Real-Time Discrete Event Control, *IEEE Transactions on Automatic Control*, **35**(4), April 1990, 386–397.

[27] Ostroff, J. S.: Composition and refinement of discrete real-time systems, *ACM Transactions on Software Engineering and Methodology*, **8**(1), 1999, 1–48.

[28] Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, *IEEE Transactions on Software Engineering*, **21**(2), February 1995, 107–125.

[29] Park, D.: Concurrency and automata on infinte sequences, *5th GI Conference on Theoretical Computer Science*, 104, Springer-Verlag, 1981.

[30] Sistla, A., Clarke, E.: The complexity of propositional linear temporal logic, *J. ACM*, **32**, 1985, 733–749.

[31] Wang, F.: Formal Verification of Timed Systems: A Survey and Perspective, *Proceedings of the IEEE*, **92**(8), August 2004, 1283–1305.

[32] Zhang, H.: *Formal Verification of Timed Transition Models*, Technical Report 17, Software Quality Research Lab, McMaster University, Hamilton, ON, Canada 2003.

A. Instantiated `ttm_decls.pvs` for M in Fig. 1

```
ttm_decls: THEORY
BEGIN
ttm_lib: LIBRARY = "../ttm_lib"
IMPORTING time_thy, actions

activity: TYPE = {a,b,c,d,e}
internal_state: TYPE = [# u:int, v:int #]                                   % * User *
ac: VAR nt_action
W: VAR internal_state

lower_bound(ac):time = CASES ac OF
                         alpha: zero,                                       % * User *
                         beta: two,                                         % * User *
                         gamma: two                                         % * User *
                       ENDCASES

upper_bound(ac):time = CASES ac OF
                         alpha: one,                                        % * User *
                         beta: infinity,                                    % * User *
                         gamma: two                                         % * User *
                       ENDCASES
```

```
enabled_state (ac, w):bool =  CASES ac OF
                        alpha: (w‘u>=0),                                        % * User *
                        beta:   True,                                           % * User *
                        gamma: (w‘v>=0)                                         % * User *
                      ENDCASES;

graph (ac, sa:activity):bool = CASES ac OF
                        alpha: (sa = a),                                        % * User *
                        beta:  (sa = b),                                        % * User *
                        gamma: (sa=a or sa=b)                                   % * User *
                      ENDCASES

IMPORTING ttm[activity,internal_state, nt_action,
      lower_bound, upper_bound,enabled_state, graph]
s: VAR states

enabled (ac, s):bool =
  IF (not (tick?(ac))) THEN enabled_general(ac,s) & enabled_time(ac,s)
  ELSE enabled_tick(s) ENDIF

trans (ac, s):states = IF tick?(ac) THEN s WITH [action_time:= update_clocks(s)]
  ELSE  s_tmp WITH [action_time:=reset_clocks(ac,s_tmp)]
   WHERE s_tmp = CASES ac OF
    alpha: s WITH [ activity:=b, basic:=s‘basic WITH[u:=s‘basic‘u+s‘basic‘v]],          % * User *
    beta:  s WITH [activity:=d, basic:=s‘basic WITH [u:=s‘basic‘u+1, v:=s‘basic‘v-1]],  % * User *
    gamma: s WITH [activity:=IF (s‘activity=a) THEN c                           % * User *
             ELSIF (s‘activity=b) THEN e ELSE s‘activity ENDIF]                 % * User *
   ENDCASES
  ENDIF

start (s):bool = (s=(# activity:=a,basic:= (# u:=0, v:=1 #),                    % * User *
                    action_time:=(LAMBDA (ac:nt_action): zero) #) )             % * User *
 IMPORTING ttm_lib@machine[states,action,enabled,trans,start]
END ttm_decls
```

B. SAL translation of `ttm_decls` for M in Fig. 1

```
ttm_decls: CONTEXT =
BEGIN
  activity: TYPE = {a, b, c, d, e};
  uvtype: TYPE = [0..2];
  action: TYPE = DATATYPE
         tick,
         alpha,
         beta,
         gamma
  END;
  internal_state: TYPE = [# u: uvtype, v: uvtype  #];
  nt_action: TYPE = {ac: action | ac /= tick};
  t1: CONTEXT = time_thy;
  cs: CONTEXT = states{activity, nt_action, internal_state, time_thy!time;};
  lower_bound (ac:nt_action): t1!time =
               IF ac = alpha THEN  t1!zero
               ELSIF ac = beta THEN t1!two
               ELSIF ac = gamma THEN t1!two
       ELSE t1!zero
               ENDIF;

  upper_bound (ac:nt_action): t1!time =
       IF ac = alpha THEN  t1!two
               ELSIF ac = beta THEN t1!infinity
               ELSIF ac = gamma THEN t1!two
               ELSE t1!zero
               ENDIF;


% preconditions

  enabled_state (si:cs!states1): bool =
       IF si.nta = alpha THEN  si.is.u >= 0
               ELSIF si.nta = beta THEN TRUE
               ELSIF si.nta = gamma THEN si.is.v >= 0
               ELSE FALSE
            ENDIF;

  graph(si:cs!states2): bool =
       IF si.nta = alpha  THEN  si.x = a
               ELSIF si.nta = beta THEN si.x = b
               ELSIF si.nta = gamma THEN  si.x = a OR
                           si.x = b
               ELSE FALSE
```

```
       ENDIF;


  t: CONTEXT = ttm{activity, internal_state, nt_action; lower_bound,
upper_bound, enabled_state, graph};
  enabled(ac:action,s:cs!states): bool =
                IF(not (tick?(ac))) THEN t!enabled_general(ac,s)
                 AND t!enabled_time(ac,s)
                ELSE t!enabled_tick(s)
                ENDIF;
  examplettm: MODULE =
  BEGIN
  GLOBAL s: cs!states
  INITIALIZATION
  s = (# activ := a,
                    basic:= (# u := 0, v := 1 #),
                    action_time:=(LAMBDA (a:nt_action): t1!zero)#)


  TRANSITION
  [
      enabled(tick, s) -->  s' = s WITH .action_time := t!update_clocks(s)
      []
      enabled (alpha, s) --> s' = ((s WITH .activ := b)  WITH
       .basic.u := s.basic.u + s.basic.v)  WITH
       .action_time:= t!reset_clocks(alpha,
       (s WITH .activ := b) WITH .basic.u := s.basic.u + s.basic.v)
      []
      enabled(beta, s) -->  s' = (((s WITH .activ := d)
WITH .basic.u := s.basic.u + 1)
        WITH .basic.v := s.basic.v - 1)
        WITH .action_time:= t!reset_clocks(beta,
        ((s WITH .activ := d) WITH .basic.u := s.basic.u + 1)
        WITH .basic.v := s.basic.v - 1)
      []
      enabled(gamma, s) --> s' = IF s.activ = a THEN
(s WITH .activ := c) WITH .action_time:= t!reset_clocks(gamma,
         s WITH .activ := c)
        ELSE
        (s WITH .activ := e) WITH .action_time:= t!reset_clocks(gamma,
         s WITH .activ := e)
        ENDIF
  ]

  END;
END
```