Formal Verification of Timed Transition Models

Formal Verification of Timed Transition Models

By
Hong Zhang, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE(2002) McMaster University
(Computer) Hamilton, Ontario

TITLE: Formal Verification of Timed Transition Models

AUTHOR: Hong Zhang, B.Eng.(East China University of Chemical Technology, PRC

SUPERVISOR: Dr. Mark Lawford and Dr. Jeffery Zucker

NUMBER OF PAGES: xiii, 142

Abstract

Labeled Transition Systems (LTSs) are used to express specifications and implementations of software. State-Event Labeled Transition Systems (SELTS's) extend LTS's by adding a state output map and an event map. A Timed Transition Model (TTM) is a timed extension of SELTS. The extension involves lower and upper time bound constraints and transitions, that relate to the number of occurrences of the special transition *tick*. A TTM can be described as a SELTS with a timer attached to each different transition, so that we can specify the time requirements of the model.

This thesis gives the definitions of invariants, strong equivalence and weak equivalence for SELTSs and TTMs in PVS. It also provides a unified modeling environment for SELTSs and TTMs in PVS which allows the user to specify them easily. Further, the thesis illustrates the use of the modeling environment in PVS to prove invariants, weak equivalence and strong equivalence of SELTS's and TTM's by providing one example in each category. Finally, we use our modeling environment to formalise and verify the correctness of an industrial real-time controller.

Our method has the advantage that it simplifies the procedure for translating SELTSs and TTMs into PVS. A disadvantage is that it still needs a number of interactions between the user and PVS, although some of these could be considered as routine work.

Acknowledgement

First of all I would like to thank my supervisors: Dr. Mark Stephen Lawford and Dr. Jeffery Zucker for their insights, guidance, prompt feedback and stimulating discussions. I have learned a lot from them in both academic and non-academic areas.

Many thanks to Dr. Ryszard Janicki and Dr. Michael Soltys for reviewing my thesis and their valuable suggestions and comments.

Next, I must thank Dr. Natarajan Shankar, Dr. Myla Archer and Dr. Ramesh Bharadwaj for their kind help through email.

Also, I would like to thank my office mates for their helpful discussions and friendship.

Special thanks to my wife Ping, my parents and my son Jim, for their love, encouragement and support.

Financial support for my study and research was generously given by the Ministry of Training through the Ontario Graduate Scholarship, and by McMaster University through scholarships.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

State-Event Labeled Transition Systems (SELTS)[Law97] provide a visual and concise way of describing the design of specification and the implementation of software. The Timed Transition Model (TTM) [Law97] is a concise way of describing state-event transition structures representing real-time systems by adding time requirements. One can expand the TTM specification into the corresponding SELTS specification manually and then check its properties. The objective of this thesis is to present a method for formally specifying SELTS and further verifying its properties in PVS. In addition, the thesis also presents a method to formally specify TTMs in PVS, and further, verify the properties of the TTM in PVS without expanding the TTM specification manually into the corresponding SELTS specification. A unified modeling environment for specifying SELTSs and TTMs in PVS is provided.

The thesis gives the definitions of invariants, strong equivalence and weak equivalence for SELTSs and TTMs in PVS. PVS is used to verify that two SELTSs or TTMs are strongly or weakly equivalent. For example, to prove two TTMs are weakly equivalent in PVS. we first specify the two TTMs in PVS using the modeling environment. Then we define a relation between the states of these two TTMs and prove that the relation is a weak state-event bisimulation. Finally we prove that the initial states of these two TTMs are related by this relation. We then conclude that these two TTMs are weakly equivalent.

1.1 Motivation

According to a study by the U.S. Department of Commerce's National Institute of Standards and Technology(NIST) (Yahoo!News-Jun 28,2002), software bugs cost the U.S. economy about $59.5 billion a year. The impact of software errors is enormous, because virtually every business in the United States now depends on software for the development, production, distribution, and after-sales support of products and services.

In the meantime[Law97], the increasing use of real-time computer control systems in safety-critical systems has led to a need for methods to ensure the correct operation of real-time systems. Increasingly, system designers are being asked to design and build safety critical real-time computer control systems such as nuclear reactor shutdown systems[LFM00]. If system designers deal with these problems without the aid of mathematical tools, errors are inevitable. The development of formal methods to ensure the correct operation of software reliant control systems represents one of the most important and pressing problems in engineering today. However, applying formal methods to practical systems raises several challenges:

- The transfer from the specification method used by the developers to formal descriptions in the theorem can be very difficult for the developers, especially for real time systems.
- An automatic theorem prover demand a higher level of sophistication from the user.

The thesis investigate model checking and theorem proving methods in software development for real-time control systems; especially how to formalize SELTSs and TTMs in PVS and verify invariants, strong equivalence and weak equivalence of these systems.

1.2 Thesis Scope

The thesis gives the definitions of invariant, strong equivalence and weak equivalence for SELTSs and TTMs, according to ideas from[Law97]. It also provides a unified

modeling environment for SELTSs and TTMs in PVS which allows the user to specify SELTSs and TTMs easily in PVS. Further, the thesis illustrates the use of the modeling environment in PVS to prove invariants, weak equivalence, strong equivalence of SELTSs and TTMs by providing one example in each category. And a non-trivial real-time software verification problem from industry is illustrated in Chapter 6.

1.3 Related Work

1.3.1 Real-time Software Verification in a Theorem Prover

The PVS-RT method developed by Lawford and Wu[Wu01, LW00] has been used to verify simple timing properties. The main advantage of the PVS-RT method is that it delivers a guarantee of domain coverage. It checks all possible input sequences, and in the case when the specification and requirement are not equivalent, it provides some insight into the reasons for any discrepancies. Moreover, when a programmmer suspects discrepancy, he can attempt a "refutation" theorem to confirm that the implementation does not satisfy the specification. When properly applied, this method for the verification of timing blocks provides an increased level of confidence in the verification process and aids in detecting subtle timing errors.

The PVS-RT method is a straightforward extension of the existing successful (untimed) methods of [LFM00]. Thus it can integrate with the existing method for verification of functionality of the input/output specifications.

The PVS-RT method has been used to verify real-time software successfully. It can avoid the state space explosion through the use of inductive proofs, and has proved to be useful in expressing some common situations in the process control field. The main obstacle in applying this technology is the excessive amount of user intervention required. Model-checking provides an alternative candidate for automatic verification of timing properties.

1.3.2 Real-time Systems Models and Specification

Model-checking has been used successfully in hardware design. But there are few successful applications in software design, especially in real time software design.

This is due to the state explosion problem, and also to the lack of a natural way of expressing common real-time properties.

In the next sections, we discuss four efforts in model-checking real-time systems.

Verus and RTCTL

In the Verus system[EGP00], a language called Verus is used to describe the real-time systems. The description is then compiled into a labeled state-transition graph that formally models the behavior of the real-time system. The Verus language provides special support for expressing timing aspects such as deadline, priorities and delays, so it is very suitable for software such as operating systems.

RTCTL (Real Time CTL)[EGP00] is an extended logic of CTL(Computation Tree Logic). It introduces bounds in the CTL temporal operators. With this formalism, we can express properties like "every request from a client should be met with a response from the server within five time units" as $AG(request \rightarrow AF^5 response)$.

The main limitation is that it can only deal with finite state models. But it can provide valuable performance information about the system.

Mocha

The input language that MOCHA [AAG+00] uses for model description is REACTIVEMODULES. This is similar to a programming language. The structure of a REACTIVEMODULES description resembles that of a conventional programming language: the statements of the language correspond to *atoms*, and the procedures correspond to *reactive modules*. A complete description consists of one or more modules. The state of the system is described by a set of *state variables:* each system state corresponds to an assignment of values to the variables. The behavior of the system consists of *an initial round*, which initializes the variables to their initial values, followed by a sequence of *update rounds*, which assign new values to the variables, thus describing the evolution of the system's states.

Mocha uses *Alternating Temporal Logic*(ATL) as the specification language. ATL can be seen as a sublanguage of CTL. Linear-time Temporal Logic assumes implicit universal quantification over all paths that are generated by system moves; and Braching-time Temporal Logic allows explicit existential and universal quantifi-

cation over all paths; whereas ATL offers selective quantification over those paths that are outcomes of games, such as the game in which the system and the environment alternate moves. The logics of LTL and CTL have their natural interpretation over the computations of closed systems, where a closed system is one whose behavior is completely determined by its state. However, the compositional modeling and design of reactive systems requires each component to be viewed as an open system, where an open system is one that interacts with its environment and whose behavior depends on its state as well as the behavior of the environment.

UPPAAL

UPPAAL[LPY97] uses networks of timed automata to model the system. The model is constructed by the description language, which is a non-deterministic guarded command language with real-valued clock variables and simple data types. It describes the system behavior as a network of automata extended with clock and data variables. The clocks are real-valued, and the guards are boolean combinations of integer bounds on clocks and clock-differences. Actions may be performed on clocks.

The UPPAAL logic can check reachability and invariance properties of boolean combination of automata locations, and clocks and integers constraints.

UPPAAL is suitable for systems that can be modeled as collections of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. And UPPAAL logic can check for simple invariant and reachability properties.

TTM/RTTL

TTMs("Timed Transition Model") due to Ostroff[Ost95] incorporates lower and upper time bound constraints on transitions other than "tick". We explain this model in the next chapter.

RTTL ("Real Time Temporal Logic") is the specification language for TTMs. RTTL formulas are constructed from state-formulas together with special temporal logic operators such as $\Box(henceforth)$ and $\Diamond(eventually)$. It is easy to express some common time requirements without using operators in convoluted ways.

Above all, model checking avoids the construction of complicated proofs and pro-

vides a counterexample trace when some specification is not satisfied. But for verifying software, abstraction by humans is still unavoidable. It is used to reduce the state space, and to cope with infinite state spaces due to the time variable.

1.3.3 Our efforts to implement TTM in Salsa

Salsa

- Introduction

  Salsa[BS00] is an invariant checker for specifications in SAL(the SCR Abstract Language). It conducts induction verification which is based on a combination of BDD algorithms and a constraint solver for integer linear arithmetic.

- Advantage and Limitations

  It is automatic and it returns a meanful state or a state pair as a counterexample. It is claimed that Salsa can handle large(even infinite state) specifications that current day model checkers cannot do due to the use of induction and the symbolic encoding of expressions involving integers as linear constraints. But due to the incompleteness of induction, users must validate the counterexamples.

  Salsa does not have record types included in its specification language. So it is difficult to prevent interleaving of the variables in the internal states of TTMs.

1.4 Organization of the Thesis

Chapter 2 gives a brief description of Timed Transition Models(TTMs). A typical TTM example called "the simple TTM" is formalized in PVS in Chapter 3.

The verification of invariant of "the simple TTM" in PVS is discussed in Chapter 4. The PVS theories and the proofs are given in Appendix A.

Chapter 5 gives the definition of strong and weak equivalence and how it was specified in PVS. Examples of verification of strong and weak equivalence of SELTSs and TTMs are provided. The PVS theories and proofs of strong equivalence of SELTSs are given in Appendix B. The PVS theories and proofs of strong equivalence of TTMs

are given in Appendix C. The PVS theories and proofs of weak equivalence of SELTSs are given in Appendix D. The PVS theories and proofs of weak equivalence of TTMs are given in Appendix E.

Chapter 6 describes the formalization and verification of an industrial real-time controller in PVS using the modeling environment defined in Chapter 3 and 5.

Chapter 7 summarizes the benefits and limitation of the current methods and proposes some furture work.

Chapter 2

Timed Transition Model

2.1 State-Event Labeled Transition Systems

State-Event Labeled Transition Systems (SELTSs) (introduced in[Law97]) extend Labeled Transition Systems(LTSs) by adding a state output map. We further add an event map used in the definition of equivalence of two SELTSs. SELTS provides a convenient way of illustrating the state and event dynamics of TTM. Our definition of SELTS is similar to the definition in[Law97]:

Definition 2.1.1 *A State-Event Labeled Transition System (SELTS) is a 6-tuple* $\mathbb{Q} := \langle Q, \Sigma, R_\Sigma, q_0, ps, pa \rangle$ *where* Q *is an at most countable set of states,* Σ *is a finite set of elementary actions or events,* $R_\Sigma = \{\xrightarrow{\alpha} : \alpha \in \Sigma\}$ *is a set of binary relations on* Q*,* $q_0 \in Q$ *is the initial state and* $ps : Q \to Q_{common}$ *is the state output map, a function mapping each state into a new set of states.* $pa : \Sigma \to \Sigma_{common}$ *is the event map, a function mapping each event into a new set of events.*

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. Most TTMs can be expanded to a corresponding SELTSs so that we can analyze it.

2.2 Definition of TTM

This is a modified version[Law97] of Ostroff's Timed Transition Model (TTM). It provides a concise way of describing state-event transition structures representing

real-time systems. Most of the following content are taken from [Law97].

A *timed transition model*(TTM) M is a triple given by

$$M := \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$$

where $\mathcal{V}$ is a set of variables, Θ is an initial condition (a boolean valued expression in the variables), 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 usually denote by x. For $v \in \mathcal{V}$ the range space of v is denoted by $Range(v)$.

  We define $\mathcal{Q}$, the set of *state assignments* of M, to be the product of the ranges of the variables in $\mathcal{V}$. That is

  $$\mathcal{Q} := \times_{v_i \in \mathcal{V}} Range(v_i)$$

  For a state assignment $q \in \mathcal{Q}$ and a variable $v \in \mathcal{V}$, we denote the value of v in state assignment q by $q(v)$ where $q(v) \in Range(v)$.

- Θ is *the initial condition*, a boolean valued expression in the variables of $\mathcal{V}$ that is used to identify a unique initial state of the system.

- $\mathcal{T}$ is *the transition set.*

  A transition is a labeled 4-tuple

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

  where α is used as the transition's label, and

  - e_α is the transition's enablement condition (a boolean valued expression in the variables of $\mathcal{V}$).
  - h_α is the operation function.
  - $l_\alpha \in Range(t) = N$ and $u_\alpha \in \mathrm{N} \cup \{\infty\}$ 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 $h_\alpha : \mathcal{Q} \rightarrow \mathcal{P}(\mathcal{Q})$ maps the current state assignment to the set of new state assignments that are possible next states when the transition occurs.

$\mathcal{T}$ always contains the special transition tick:

$$tick := (true, [t : t + 1], -, -)$$

which represents the passage of time on the global clock. *tick* is the only transition that affects the time variable t and also has no lower or upper time bound. All other transition time bounds are given relative to numbers of occurrences of *tick*.

2.3 TTM Semantics

A trajectory of a TTM is any infinite sequence of TTM state assignments and transitions of the form $q_0 \stackrel{\alpha_0}{\rightarrow} q_1 \stackrel{\alpha_1}{\rightarrow} q_2 \stackrel{\alpha_2}{\rightarrow} \ldots$ The interpretation is that q_i goes to q_{i+1} via the transition α_i. A state trajectory $\sigma := q_0 \stackrel{\alpha_0}{\rightarrow} q_1 \stackrel{\alpha_1}{\rightarrow} q_2 \stackrel{\alpha_2}{\rightarrow} \ldots$ 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.

2. *Succession*: For all i, q_{i+1} is obtained from q_i by applying the operation function of α_i, and α_i is enabled in state assignment q_i.

3. *Ticking*: The clock must tick infinitely often. That is, there are an infinite number of transitions α_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 ticks. This would imply that the TTM is performing an infinite number of actions in a finite time.

4. *Time Bounds*: To determine if the trajectory σ satisfies the time bound requirements of the TTM M, we associate with each non-tick transition α, a counter variable c_α with Range(c_α) = N. Each α transition's counter is initially set to zero and is reset to zero after an α transition or a transition that enters a new

state assignment where α is disabled. The counter is only incremented by the occurrence of a *tick* transition when α is enabled (e_α = true). Any non-tick transition α can legally occur only when its counter is in the region specified by the transition's time bounds (ie. $l_\alpha \leq c_\alpha \leq u_\alpha$). The upper time bounds on transitions are hard time bounds by which the transitions are guaranteed to occur. Thus if α's counter reaches its upper time bound, then it is forced to occur before the next tick of the clock unless it is preempted by another non-tick transition that disables α (and hence resets α's counter). Hence for a *tick* transition to legally occur, every enabled transition α must have a counter value less than its upper time bound ($c_\alpha < u_\alpha$).

The following is a formal description of the above:

For the TTM $M = \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, we will denote the set of transition counters by $C := \{c_\alpha | \alpha \in \mathcal{T} - \{tick\}\}$. We then obtain the TTM's underlying state set $\overline{\mathcal{Q}} := \mathcal{Q} \times N^C$, the set of *extended state assignments*. From the trajectory σ we derive the full trajectory $\overline{\sigma} := \overline{q}_0 \overset{\alpha_0}{\rightarrow} \overline{q}_1 \overset{\alpha_1}{\rightarrow} \overline{q}_2 \overset{\alpha_2}{\rightarrow} \ldots$, where each $\overline{q}_i \in \overline{\mathcal{Q}}$ is obtained from σ as follows:

For all $v \in \mathcal{V}, \overline{q}_i(v) = q_i(v)$

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

$$\overline{q}_{i+1}(c_\alpha) = \begin{cases} \overline{q}_i(c_\alpha) + 1, & \text{if } q_i(e_\alpha) = true \text{ and } \alpha_i = tick \\ 0, & \text{if } q_{i+1}(e_\alpha) = false \text{ or } \alpha_i = \alpha \\ \overline{q}_i(c_\alpha), & \text{otherwise} \end{cases}$$

The trajectory σ satisfies the time bounds of M iff the following two conditions hold in $\overline{\sigma}$ for all i = 0, 1, ...:

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

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

Notes:

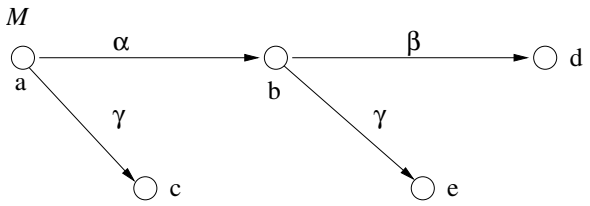
1. A condition equivalent to (a) is that for all $c_\alpha \in C, \overline{q}_i(c_\alpha) \leq u_\alpha$.

2. Any loop of transitions in a TTM (a sequence of transitions starting and ending in the same activity) must have at least one transition with a non-zero upper time bound. Otherwise, once the first transition of the loop is enabled, our transition rules could possibly force an infinite number of non-*tick* transitions to occur without being interleaved by an infinite number of *ticks*.

2.4 A small example of a TTM

1. **TTM description**

   As a small example, consider the TTM $M := \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ shown in Figure 2.1. This example is a very typical example of TTM introduced in [Law97]. We will use this example to test our specification in PVS. The following description of this example is taken from [Law97], interesting readers are referred to [Law97] for more details.



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

Figure 2.1: A small example of TTM

As we can see, the complete enablement conditions for an action α should include the e_α and $x = a$ because α can only happen when the TTM is in activity a. Similiarly, we know the complete enablement conditions for γ should be $v \geq 0$ and $((x = a)$ or $(x = b))$ because γ can happen only in activity a and b according to the graph.

2. **Legal trajectories in this example**

The legal trajectories of the TTM in Figure 2.1 are shown in Figure 2.2.

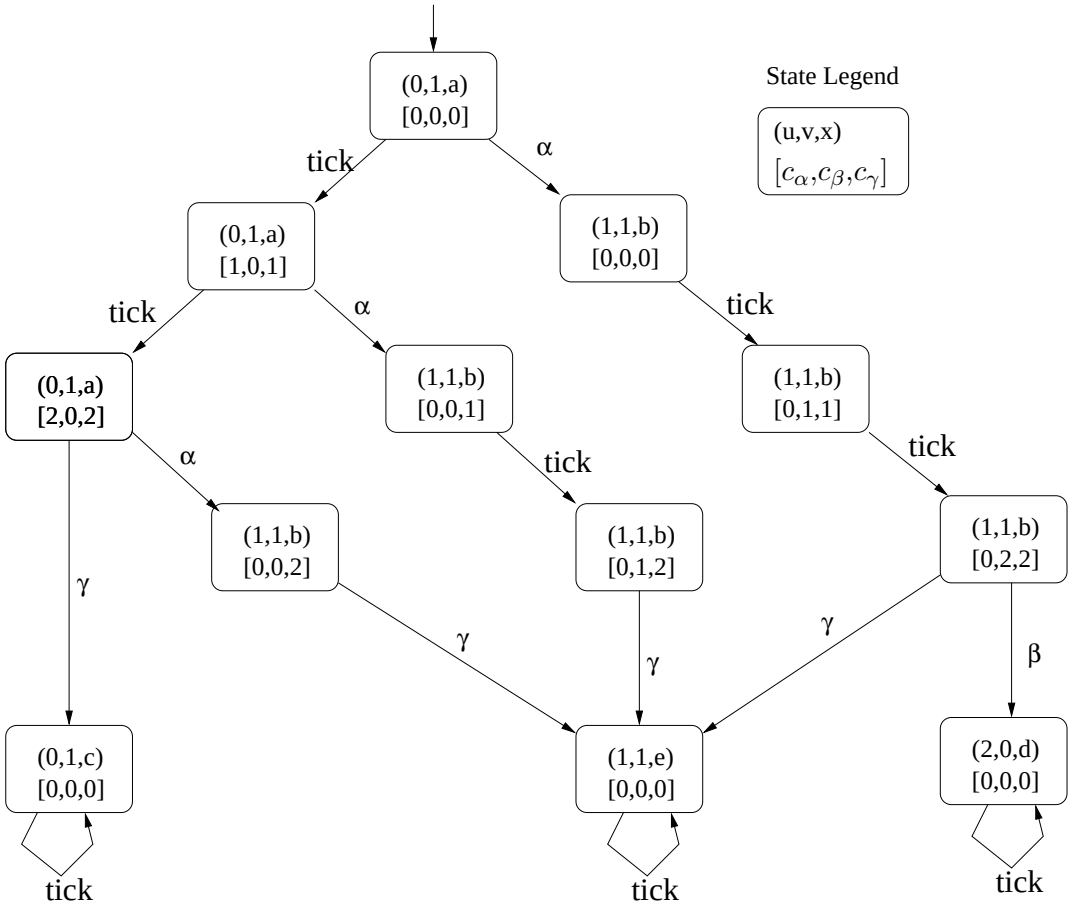


Figure 2.2: The legal trajectories of TTM M of Figure 2.1

2.5 Difference and similarities of TTM, LV Timed Automata and Timed Automata

We compare these three timed automata with respect to how time is represented and how it is used as time constraints, in each automaton. All three are labeled transition systems extended with time.

1. How time is introduced

   All the three automata assume non-time transitions are occuring instantaneously. TTMs[Ost98] use natural numbers to represent time, while Timed Automata[EGP00] and LV Timed Automata[HA98] represent time as non-negative reals. Timed Automata use non-negative rationals as time constants.

   (a) TTM

   Time is associated with operations (transitions) and there is a special transition "tick" which represents the passage of time on the global clock; this is the only transition that affects the time variable t. Other transitions are controlled by the time constraint associated with them and other state variables conditions. So it is convenient to express some common specifications as TTMs.

   (b) LV Timed Automata in TAME

   To be precise, the automata used in the TAME example are a hybrid of LV timed automata and MMT automata (I/O automata together with upper and lower bounds on time). Each state of the automaton has an associated current time value. The actions of the automaton include a special *time-passage action* v. All actions (except for the time-passage action) do not change the time component of the state.

   An MMT automaton is an I/O automaton together with upper and lower bounds on time. The actions are partitioned into *external* and *internal* actions; the external actions are further partitioned into *input* and *output* actions. It is required that the automaton be *input-enabled*, by which is meant that input actions are enabled for every state.

   The LV timed automaton in TAME is obtained by augmenting the MMT automaton with a *now* component, plus *first(C)* and *last(C)* components for each class of the task partitions. The first and last components represent, respectively, the earliest and latest time in the future that an action in class C is allowed to occur. The *now*, *first* and *last* components all take on values that represent *absolute* times, not incremental times. The time-passage action v is also added.

(c) Timed Automata[EGP00]

A timed automaton is a finite automaton augmented with a finite set of real-valued *clocks*. Time can elapse when the automaton is in a state or location. The clocks may be reset to zero explicitly by transitions. Time constraints can be associated with states or with transitions. When the time constraint is associated with a state, it is called *invariant* and require that time can elapse in a location only as long as its invariant stays true. When the time constraint is associated with a switch (or transition), the switch can be taken only if the current values of the clocks satisfy this constraint.

2. Advantages and Limitations

- TTM

  Time is directly related to the actions and state variables(enable conditions). It can directly express specifications involving time constraints and state constraints. But "tick" has to updates all the clocks (i.e. time variables), and the number of time variables increase with the system size.

- LV Timed Automata

  It is not easily expressed as a graph. The automaton is normally described by listing the state variables, initial states and actions in terms of their preconditions and effects.

  The guard can include both the time constraint and other non-time state variables. But the time constraint limits to lower and upper time bounds of some actions.

  Because of the nature of the representation, it is limited to specifying which kind of actions have to be done in some time frame. But it is not always convenient to put time constraints in states or in arbitrary actions. Also the time constraints are restricted to lower and upper bounds. It is, however, very useful for some applications.

- Timed Automata[EGP00]

  These are widely used. The clocks and locations are separate. The clocks have to be reset explicitly in the state transition system. We can consider

the system to be equipped with a finite number of stop-watches which can be started and checked independently of one another, but all stop-watches refer to the same clock. It is an important advantage to have multiple clocks which can be set independently of one another.

However there are no non-clock variables in the state or as a constraint of the switch (transition). TAs can specify time requirements but it is not easy to associate states with state variables which are not time variables in the guards.

Chapter 3

Formalization of Timed Transition Model in PVS

3.1 PVS

PVS stands for "Prototype Verification System," and as the names suggests, it is a prototype environment for specification and verification. The examples and descriptions in the following sections are largely based upon[LFM00],[Wu01], [SORSC99b], [COR+95].

3.1.1 Introduction

The system consists of a specification language, a parser, a type checker, and an interactive theorem prover. The specification language is based on higher-order logic with a richly expressive type system. Its theorem prover is both interactive and highly mechanized: the user chooses each step that is to be applied and PVS performs it, displays the result, and then waits for the next command. PVS differs from most other interactive theorem provers in the power of its basic steps: these can invoke decision procedures for arithmetic, automatic rewriting, induction, and other relatively large units of deduction; it differs from other highly automated theorem provers in being directly controlled by the user.

The PVS theorem prover consists of a powerful collection of inference procedures that are applied interactively under user guidance within a sequent calculus frame-

work. One can use the primitive inferences to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover.

3.1.2 The PVS sequent calculus

Sequent Calculus was introduced by Gentzen in 1935 as a way to present the syntax of logical systems. We review the key principles of the PVS sequent calculus and the associated PVS commands. The basic structure of the PVS sequent calculus is a sequent, which can be defined as follows:

Definition 3.1.1 *A sequent in the sequent calculus is an ordered pair* (Γ, Δ) *of sets of formulas in higher order logic and is written* $\Gamma \vdash \Delta$. *Here* Γ *is called the antecedent,* Δ *is called the consequent, and* $\vdash$ *denotes syntactic entailment.*

Sequents in PVS are represented as a list of antecedents and consequents separated by a turnstile. Henceforth we will use $\neg P_1$, $P_1 \wedge P_2$ and $Q_1 \vee Q_2$ to denote negation, conjunction and disjunction respectively. There are implicit $\wedge$'s among the antecedents and implicit $\vee$'s among the consequents. Below is a typical sequent in PVS:

```
 [-1] P(x)
 {-2} Q(y)
|----------
 {1}  R(x)
 [2]  S(y)
```

Here P(x) and Q(y) are antecedents and R(x) and S(y) are consequents. The square brackets, e.g., [-1], indicate formulas that are unchanged in a subgoal from the parent goal. Whereas the braces, e.g., -2, indicate formulas which are either new or different from those of the parents. The numbers inside them are used to name the corresponding formulas. A negative number represents the antecedent whereas a positive number denotes the consequent. This sequent can be translated into the following logical expression:

$$P(x) \wedge Q(y) \Rightarrow R(x) \vee S(y) \tag{3.1}$$

This logical expression is also called the *characteristic formula*.

3.1.3 Proofs in PVS sequent calculus

Proofs are done by transforming the sequent until one of the following forms is obtained:

1. FALSE is an antecedent
2. TRUE is a consequent
3. Formula P is both an antecedent and a consequent

When these three cases appear in the sequent, PVS will recognize them as trivially true and the proof will be finished showing "Q.E.D" at the end. The entire objective of PVS is to manipulate sequents to obtain one of the three cases. We will outline some of the basic manipulation rules in the sequent calculus, together with the associated, low-level PVS commands.

1. Eliminate the conjunction ($\wedge$) and the disjunction ($\vee$) in the antecedent and consequent respectively.

$$\left|\begin{array}{l} \\ P_1 \wedge P_2 \\ \hline Q_1 \vee Q_2 \\ \\ \end{array}\right. \Longleftrightarrow \left|\begin{array}{l} P_1 \\ P_2 \\ \hline Q_1 \\ Q_2 \end{array}\right. \tag{3.2}$$

   The PVS command associated with this rule is (*flatten*).

2. Eliminate the conjunction ($\wedge$) and the disjunction ($\vee$) in the antecedent by splitting them into two subgoals.

$$\begin{array}{l} \Gamma \\ \hline Q_1 \wedge Q_2 \\ \Delta \end{array} \tag{3.3}$$

$$\swarrow \qquad\qquad \searrow$$

$$\begin{array}{l} \Gamma \\ \hline Q_1 \\ \Delta \end{array} \qquad\qquad \begin{array}{l} \Gamma \\ \hline Q_2 \\ \Delta \end{array}$$

$$\begin{array}{|l} P_1 \vee P_2 \\ \Gamma \\ \hline \Delta \end{array} \qquad (3.4)$$

$$\swarrow \qquad\qquad \searrow$$

$$\begin{array}{|l} P_1 \\ \Gamma \\ \hline \Delta \end{array} \qquad\qquad \begin{array}{|l} P_2 \\ \Gamma \\ \hline \Delta \end{array}$$

The PVS command associated with this rule is (*split*).

3. Eliminate negation in the sequent.

$$\begin{array}{|l} \Gamma \\ \hline \neg Q \\ \Delta \end{array} \iff \begin{array}{|l} \Gamma \\ Q \\ \hline \Delta \end{array} \qquad (3.5)$$

$$\begin{array}{|l} \Gamma \\ \neg P \\ \hline \Delta \end{array} \iff \begin{array}{|l} \Gamma \\ \hline P \\ \Delta \end{array} \qquad (3.6)$$

Normally, PVS will move the negation to the opposite part automatically when we begin the related theorem proof.

4. Eliminate the implication ($\Rightarrow$) in the consequent.

$$\begin{array}{|l} \\ \Gamma \\ \hline P_1 \Rightarrow P_2 \\ \Delta \end{array} \iff \begin{array}{|l} \Gamma \\ P_1 \\ \hline P_2 \\ \Delta \end{array} \qquad (3.7)$$

The PVS command associated with this rule is (*flatten*).

5. Eliminate the implication ($\Rightarrow$) in the antecedent by splitting it into two subgoals.

$$\begin{array}{|l} P_1 \Rightarrow P_2 \\ \Gamma \\ \hline \Delta \end{array} \tag{3.8}$$

$$\swarrow \qquad\qquad \searrow$$

$$\begin{array}{|l} P_2 \\ \Gamma \\ \hline \Delta \end{array} \qquad\qquad \begin{array}{|l} \Gamma \\ \hline P_1 \\ \Delta \end{array}$$

The PVS command associated with this rule is (*split*).

6. Eliminate the universal quantifier in the consequent and the existential quantifier in the antecedent.

$$\begin{array}{|l} \Gamma \\ \hline \forall x_1, \cdots, x_n : Q \\ \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ \hline Q[c_1/x_1, \cdots, c_n/x_n] \\ \Delta \end{array} \tag{3.9}$$

$$\begin{array}{|l} \Gamma \\ \exists x_1, \cdots, x_n : Q \\ \hline \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ Q[c_1/x_1, \cdots, c_n/x_n] \\ \hline \Delta \end{array} \tag{3.10}$$

The PVS command associated with this rule is (*skolem fnum (const)*), where *fnum* is a PVS formula number identifying the formula to be skolemized and *const* is a list of skolem constants $(c_1, \cdots, c_n)$. Note that a skolemized constant c_i must be the *fresh* constant, i.e. it must not already appear in the sequent.

7. Eliminate universal quantifier in the antecedent and existential quantifier in the consequent.

$$\begin{array}{|l} \Gamma \\ \forall x_1, \cdots, x_n : Q \\ \hline \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ Q[t_1/x_1, \cdots, t_n/x_n] \\ \hline \Delta \end{array} \tag{3.11}$$

$$\frac{\Gamma}{\exists x_1, \cdots, x_n : Q, \Delta} \Rightarrow \frac{\Gamma}{Q[t_1/x_1, \cdots, t_n/x_n], \Delta} \tag{3.12}$$

A universally quantified variable in (3.11) or an existential quantified variable in (3.12) can be instantiated by any term of the same type. The PVS command associated with this rule is (*inst fnum term*), where *fnum* is a formula number identifying the formula to be replaced and *term* is a list of instantiation constants $(t_1, \cdots, t_n)$. Note that the quantified formula is not deleted while using (*inst fnum term*). That is, the quantified formula is hidden in the current sequent but it is still there. We can use the PVS command (*reveal fnum*) to reintroduce the hidden formula or use (*inst-cp fnum term*) to instantiate the quantifier and at the same time leave the original quantifier in the sequent. The detailed descriptions of PVS commands can be found in PVS Prover Guide[SORSC99a].

3.1.4 Unprovable sequents and counter examples [LFM00]

Suppose we wanted to use sequent calculus to check if the following formula is a logical theorem:

$$(Q \Rightarrow P_1 \vee P_2) \wedge P_1 \wedge (P_2 \Rightarrow Q) \Rightarrow Q$$

We translate this logical expression into the sequent:

$$\frac{Q \Rightarrow P_1 \vee P_2,\ P_1,\ P_2 \Rightarrow Q}{Q}$$

Using the fact that $(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$ and applying (3.8),(3.4) and (3.6). we obtain the (unprovable) sequent:

$$\frac{P_1}{Q,\ P_2}$$

which has characteristic formula $P_1 \Rightarrow (Q \vee P_2)$. This formula is false when $P_1 = True$ and $P_2 = Q = False$. One can easily verify that this assignment provides a counter example showing the original formula is not a logical theorem.

3.2 TAME: Timed Automata Modeling Environment

The Timed Automata Modeling Environment [AHR00] 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, I/O automata, and the automata model that underlies SCR specifications. TAME supports invariant checking on these automata models but it does not support automated composition of automatons. The user can 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 sufficient for the 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 natural numbers to represent time values. The special *tick* action needed to update all the clocks is associated with each *non-tick* action. Our actions also need to satisfy the state variable requirements, which is a very common situation in control systems. But TAME can be a good basis for our formalization of TTMs in PVS. We will discuss our representation in the next section.

3.3 PVS Theories for the Timed Transition Model

We introduce several PVS theories and a template which supports 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 very straightforward way. It also provides us with unified and appropriately structured PVS specifications which can be used and understood by software developers who are using TTM as the specification method.

3.3.1 The Theory `time`

In a Timed Transition Model, each *action* has an associated timer with values in the natural numbers. The value of the timer is compared with the *lower_bound* and *upper_bound* to decide the *enabled_time* condition for each action. For the *upper_bound* associated with an *action*, ∞ is allowed to represent the case where no final deadline on an action exists. So the time type in our model is the union type of natural numbers and $\{\infty\}$, shown in Figure 3.1.

```
time: DATATYPE

 BEGIN

  fintime(dur: nat): fintime?
  infinity: inftime?

 END time
```

Figure 3.1: Theory `time` specifies the type time

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 also borrowed the theory `time_thy` from TAME[HA98] which contains the definitions of the standard arithmetic operators and predicates for time values. Interesting reader can get more information from [OS97] and [HA98] about the abstract datatype and theory `time_thy`.

3.3.2 The Theory `states`

The theory `states` provide 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*.

It provides us with all the information we need to define the record type to represent the status of the state.

The theory consists of one statement to define the type of states. It has three fields. The first field is *activity* which specifies the label of the state. The second field is *basic* which represent all the non-time information about the status of the state. 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. We can say that each action is associated with a timer. The value of the timer is decided by the value of *internal_state*, the label of the state and how many times `tick` has happened.

```
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
```

Figure 3.2: Theory `states` specifies the type states

3.3.3 The Theory `ttm`

The theory `ttm` is the basic theory which specifies all the common time operations of TTMs. It is a separate file in the modeling environment. The software developer does not need to change anything in this theory. The file name for it is "ttm.pvs". It provides the standard names and definitions that are common to every TTM. Thus the software developer does not need to figure out how to specify the time operations of the TTM in PVS. All they need to do is import this theory into their specification of TTMs based on the TTM template which we will discuss in section 3.4.2. The complete theory `ttm` appears in Figure 3.3.

The theory has seven parameters to define a Timed Transition Model. They are:

activity : the set of all possible labels of states.

internal_state : the state variables excluding the time variable and the activity.

```
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

            enabled_general(ac:nt_action,s:states):bool =
                        enabled_state(ac,s‘basic) & graph(ac,s‘activity)

% Only the tick can change the timers associated with the action. If enabled_general is not satisfied, It is reset to zero.

            update_clocks(s:states): [nt_action−>time] =
                        (LAMBDA (q:nt_action):
                        IF enabled_general(q,s) THEN
                        s‘action_time(q) + one
                        ELSE zero
                        ENDIF
                        )

% After the normal action(except for Tick), Set the value of timers according to the new enable conditions

            reset_clocks(ac:nt_action,s:states): [nt_action−>time] =
                        (LAMBDA (q:nt_action):
                        IF (enabled_general(q,s) & q/=ac) THEN
                        s‘action_time(q)
                        ELSE zero
                        ENDIF
                        )

            enabled_time(ac:nt_action, s:states): bool =
                         s‘action_time(ac) >= lower_bound(ac) & s‘action_time(ac) <= upper_bound(ac)

            enabled_tick(s:states): bool =
                         FORALL (q:nt_action): enabled_general(q,s) =>(s‘action_time(q) < upper_bound(q))


END ttm
```

Figure 3.3: Theory `ttm` which support common time operations of TTM specifications

nt_action : the set of all possible actions exclude the action `tick`.

lower_bound : a function from *nt_action* to *time* which associate each action with a lower time bound.

upper_bound : a function from *nt_action* to *time* which associate each action with an upper time bound.

enabled_state : the enable condition of state variables.

graph : the enable condition of activities (label of states).

The three parameters: *activity*, *internal_state* and *nt_action*, are simply type parameters. The actual parameters are defined according to the Timed Transition Model. The parameters *lower_bound* and *upper_bound* are instantiated according to the time requirement of each action in the Timed Transition Model. An action can only happen when its timer is between *lower_bound* and *upper_bound*. These are all defined in the TTM template in section 3.4.2. 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 label of the state(*activity*).

In the body of the theory `ttm`, *enabled_general* combines the enablement conditions resulting from *enabled_state* and *graph*. According to the *enabled_general* condition, the *update_clocks* function updates the function from *nt_action* to *time* that represents the action's current timer values. Each timer associated with an action is updated based on the *enabled_general* condition. If this condition is true, then the timer is incremented by one. If it is false, then the timer is reset to zero. The *reset_clocks* function resets the function from *nt_action* to *time* after each action happens. It resets the values of the timers according to the new *enable_general* condition. If the *enabled_general* condition is true and the action is not the one that just happened, then the value of the timer for the action is kept. Otherwise it is reset to zero. The *enabled_time* function specifies the timing requirements of each *nt_action*. It requires the timer for the corresponding action to be between *lower_bound* and *upper_bound* for the action to be enabled. The special action `tick` can only happen when no

non-tick actions that are enabled have reached their *upper_bound* time. That is, their timers must be greater than or equal to the corresponding *lower_bound*, and less than the corresponding *upper_bound*. This is specified in the function *enabled_tick*.

3.4 PVS template for Timed Transition Models

The PVS template for Timed Transition Models (TTMs) provides a straightforward environment for specifying TTMs in PVS. To specify a TTM in PVS, we first need to define the theory `actions`. Then we fill all other information in the template.

3.4.1 The theory `actions`

In the theory `actions`, we define *action* as the type of all the possible actions in TTM. *nt_action* is a subtype of *action* excluding the special action `tick`. The template for the theory `actions` is shown in Figure 3.4.

```
actions: THEORY

BEGIN

% action is actions with tick.

                    action: DATATYPE
                    BEGIN
                     tick: tick?
                     <...>
                    END action

% nt_action is actions without tick, We need this because we do not need to associate a clock with tick

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

END actions
```

Figure 3.4: Theory `actions` of Timed Transition Model

3.4.2 The Template

In the template, we import the fixed theory `time_thy` first. Then we define *activity* (the set of the labels of states). We import the theory `actions` to get all the possible

actions in our Timed Transition Model. The subtype *nt_action*(non-tick actions) in the theory `actions` is the set of all possible actions except `tick`. Among all the actions, `tick` is a fixed action which is common for all TTMs. The definition of *internal_state* includes all of the non-time state variables excluding the activity variable.

The functions *lower_bound* and *upper_bound* specify the time requirement of non-tick actions(*nt_action*) in the TTM. The function *enabled_state* specifies the guard conditions on actions in terms of state variables. The function *graph* specifies the requirements of the labels of the states in the TTM. We can get this information directly from the graph of the TTM.

By importing the theory `ttm`, we have all of the predefined operations on *time* and the enablement conditions of actions. We also have the definition of the type *states* because the theory `states` is imported in the theory `ttm`. 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` is the same for all TTMs. It updates all the timers associated with each action according to the *enabled_general* condition. The effects of non-tick actions are also specified in the *trans* function. We also need to reset the timers (the function from *nt_action* to *time*) according to the new state variables assigned by the *trans* function. Finally, the function *start* specifies the start state of the TTM. After we define all these functions, we import the theory `machine` which allows us to specify the invariant of the TTM. The theory `machine` is included in the TAME [HA98] tool.

3.4.3 Instantiating the template

To illustrate an instantiation of the template, we use the template to specify in PVS the example in Figure 2.1. The complete trajectories are shown in Figure 2.2. As we can see, to input the TTM into the PVS template, you only need to translate each part of the TTM into the corresponding PVS part. You do not have to care about how the TTM is specified in PVS. The complete procedure is a translation procedure. The instantiating of the theory `action` is shown in Figure 3.6. The instantiating of the main template is shown in Figure 3.7.

Table 3.1 illustrates the information which the user needs to fill in the template.

```
ttm_decls: THEORY
BEGIN
        ttm_lib: LIBRARY = "../ttm_lib"
        IMPORTING time_thy
        activity: TYPE = {...}
        IMPORTING actions

        internal_state: TYPE = [# ...  #]

        lower_bound(ac:nt_action):time =
                CASES ac OF
                        <...>
                ENDCASES

        upper_bound(ac:nt_action):time =
                CASES ac OF
                        <...>
                ENDCASES

        enabled_state (ac:nt_action, w:internal_state):bool =
                CASES ac OF
                        <...>
                ENDCASES;

        graph (ac:nt_action, sa:activity):bool =
                CASES ac OF
                        <...>
                ENDCASES

        IMPORTING ttm[activity,internal_state, lower_bound,upper_bound,enabled_state, graph]
        enabled (ac:action, s:states):bool =
                IF (not (tick?(ac))) THEN enabled_general(ac,s) & enabled_time(ac,s)
                ELSE enabled_tick(s)
                ENDIF

        trans (ac:action, s:states):states =
                CASES ac OF
                        tick:  s WITH [action_time:= update_clocks(s)],
                        <...>
                ENDCASES

        start (s:states):bool =
         ( s = (# <...>
                   action_time:=(LAMBDA (a:nt_action): zero)
                #)
         )

         IMPORTING ttm_lib@machine[states,action,enabled,trans,start]

END ttm_decls
```

Figure 3.5: The template of the Timed Transition Model

```
actions: THEORY

BEGIN

% action is actions with tick.

        action: DATATYPE
        BEGIN
         tick: tick?
         alpha: alpha?
         beta: beta?
         gamma: gamma?
        END action

% nt_action is actions without tick, We need this because we do not need to associate a clock with the tick action.

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

END actions
```

Figure 3.6: Instantiating the theory `action` for the example in Figure 2.1

As we can see, all the information is very easily obtained from the specification of the TTM such as in Figure 2.1.

```
ttm_decls: THEORY
BEGIN
        ttm_lib: LIBRARY = "../ttm_lib"
        IMPORTING time_thy
        activity: TYPE = {a,b,c,d,e}
        IMPORTING actions

        internal_state: TYPE = [# u:int, v:int #]

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

        upper_bound(ac:nt_action):time =
                CASES ac OF
                        alpha: one,
                        beta: infinity,
                        gamma: two
                ENDCASES

        enabled_state (ac:nt_action, w:internal_state):bool =
                CASES ac OF
                        alpha: (w`u>=0),
                        beta:   True,
                        gamma: (w`v>=0)
                ENDCASES;

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

        IMPORTING ttm[activity,internal_state, lower_bound,upper_bound,enabled_state, graph]

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

        trans (ac:action, s:states):states =
                CASES ac OF
                        tick:  s WITH [action_time:= update_clocks(s)],
                        alpha: s WITH [activity:=b, basic:=s`basic WITH [u:=s`basic`u+s`basic`v],
                                action_time:=reset_clocks(ac,s WITH [basic:=s`basic
                                WITH[u:=s`basic`u+s`basic`v], activity:=b])],
                        beta:  s WITH [basic:=s`basic WITH [u:=s`basic`u+1, v:=s`basic`v−1], activity:=d,
                                action_time:=reset_clocks(ac,s WITH [basic:=s`basic
                                WITH[u:=s`basic`u+1,v:=s`basic`v−1],activity:=d])],
                        gamma:  If (s`activity=a) THEN s WITH [activity:=c,
                                action_time:=reset_clocks(ac,s WITH [activity:=c]) ]
                                ELSIF (s`activity=b) THEN s WITH [activity:=e,
                                action_time:=reset_clocks(ac,s WITH [activity:=e]) ]
                                ELSE s
                                ENDIF
                ENDCASES

        start (s:states):bool =
          ( s = (# activity:=a,
                    basic:= (# u:=0, v:=1 #),
                    action_time:=(LAMBDA (a:nt_action): zero)
                #)
          )

         IMPORTING ttm_lib@machine[states,action,enabled,trans,start]

END ttm_decls
```

Figure 3.7: Instantiating the template for the example in Figure 2.1

Table 3.1: Information required in the TTM specification in PVS

Name	**User Fills in**	**Comment**
activity	Declaration of all activities	The labels used in the TTM of the graph
internal_state	Declaration of the internal state which represent non-time properties of the state	Normally a record type
lower_bound	The lower bound requirement of time for all non-tick actions	Get this information from the description of actions
upper_bound	The upper bound requirement of time for all non-tick actions	Get this information from the description of actions
enabled_state	The precondition of state for all non-tick actions	Get this information from the description of actions
graph	The precondition of activity value for all non-tick actions	Get this information according to the graph for the TTM
trans	The effects of all actions	For the TTM, the tick will update clocks, and all other actions have to reset clocks according to the new state
start	The initial state of the TTM	The initial activity, initial value of the internal state; all clocks are set to zero

3.5 Summary

By separating the fixed parts of the specification of the TTM into separate theories and providing some common base of the TTM, we are able to present a straightforward way to specify the TTM in PVS driectly. By using this modeling environment, one can translate the TTM into PVS without knowing the details of the specification method in PVS, and still be confident it is specified correctly in PVS. In the following chapters we will further explore the methods to verify the properties of the TTM by PVS.

Chapter 4

Invariant Verification of Timed Transition Models in PVS

4.1 Invariant

For an automaton, a predicate is said to be an *invariant* if it holds for all the reachable states of the automaton. The same situation applies to the State Event Labeled Transition System (SELTS) and Timed Transition Model (TTM), as each of them can be considered a kind of automaton.

4.2 Definition of Invariants in PVS

In our modeling environment, we borrow the theory `machine` from TAME [AHR00] to specify the invariants of the SELTS and TTM.

4.2.1 The Theory `machine`

The theory `machine` contains the definition of the invariant and the induction principle for the invariant in the automata model. The description of the theory `machine` below is taken from [HA98]:

> Figure 4.1 shows the PVS specification of the theory `machine`. This theory, which defines mathematical induction in the context of the timed

```
machine  [ states, actions: TYPE,
         enabled: [actions,states −> bool],
         trans: [actions,states −> states],
         start: [states −> bool] ] : THEORY

 BEGIN
  s,s1: VAR states;
  a: VAR actions;
  n,n1: VAR nat;
  Inv: VAR pred[states];

  reachable_hidden(s,n): RECURSIVE bool =
      IF n = 0 THEN start(s)
      ELSE (EXISTS a, s1 : reachable_hidden(s1,n − 1) &
                           enabled(a,s1) &
                           s = trans(a,s1))
      ENDIF
      MEASURE (LAMBDA s,n: n);

  reachable(s): bool = (EXISTS n : reachable_hidden(s,n));

  base(Inv) : bool = (FORALL s: start(s) => Inv(s));

  inductstep(Inv) : bool =
    (FORALL s, a: reachable(s) & Inv(s) & enabled(a,s) => Inv(trans(a,s)));

  inductthm(Inv): bool =
    base(Inv) & inductstep(Inv) => (FORALL s : reachable(s) => Inv(s));

  machine_induct: THEOREM (FORALL Inv: inductthm(Inv));

  reachable_trans_fact(s,a) : bool =
    (reachable(s) & enabled(a,s) => reachable(trans(a,s)));

  reachable_trans: LEMMA (FORALL s,a : reachable_trans_fact(s,a));

 END machine
```

Figure 4.1: Theory `machine` defining the invariant of the automata

automata model, is the core of our general PVS strategy for performing the standard steps of the invariance proofs of the state invariants.

The theory has the five parameters needed to define a timed automaton: *states*, the automaton's states; *actions*, its input alphabet; *start*, its start states; *enabled*, the guards on state transitions; and *trans*, the automaton's transition function. The two parameters *states* and *actions* are simply type parameters. The actual parameters in an instantiation of the template are the *states* and *actions* types (i.e., the sets of possible values of *states* and *actions*) of some particular timed automaton. The parameter *start* is instantiated by a predicate on states true only for start states, and the parameter *enabled* by a predicate on actions and states that is true only when the action is enabled in the state. The parameter *trans* is instantiated by a function that maps an action and a state to a new state. Together, *enabled* and *trans* define the transition relation of the timed automaton.

The body of the theory describes six predicates used to define the induction principle. The first predicate *Inv* represents an arbitrary predicate (i.e., an invariant) on states. The second predicate *reachable_hidden* is true of a state s and natural number n if s is reachable from a start state in n steps. The MEASURE clause of this definition permits PVS to verify during type checking that the predicate *reachable_hidden* is always well defined, i.e., that its (recursive) definition terminates on all arguments. The predicate *reachable* is true of a state s if *reachable_hidden* is true for s and some natural number n. The next two predicates define the two parts of the induction principle: *base*, which states that the given invariant holds for the base case, and *inductstep*, which states that the invariant is preserved by every enabled action on a reachable state. Finally, the predicate *inductthm* on predicates states that a (invariant) predicate is true if it holds in the base case and is preserved in the induction step.

4.2.2 Invariant Specification and Proof

Our specifications are different from the ones in TAME. As suggested by Dr. Zucker, we first specify all possible values of state variables as an invariant, and then try to prove it as an invariant of the TTM. If the proof fails, one can find errors in the specification through the unprovable sequent in PVS. If it is proved, then we use the result to prove other invariants we want to prove.

Our experience is that this greatly reduces the load of the proof and the proofs of most invariants become routine work. The complete specification and proof of the example in Figure 2.1 are provided in Appendix A.

```
ttm_invariants       : THEORY

 BEGIN

        IMPORTING  ttm_decls
        IMPORTING  ttm_unique_aux
        IMPORTING  ttm_rewrite_aux_1
        IMPORTING  ttm_rewrite_aux_2

Inv_1(s:states) : bool =      (s`activity=a and s`basic`u=0 and s`basic`v=1) or
                              (s`activity=c and s`basic`u=0 and s`basic`v=1) or
                              (s`activity=b and s`basic`u=1 and s`basic`v=1) or
                              (s`activity=e and s`basic`u=1 and s`basic`v=1) or
                              (s`activity=d and s`basic`u=2 and s`basic`v=0)

Inv_2(s:states):  bool =      (s`basic`u<=2);

s: VAR states

lemma_aux1: LEMMA Inv_1(s) => Inv_2(s);

lemma_1: LEMMA (FORALL (s:states): reachable(s) => Inv_1(s));

lemma_2: LEMMA (FORALL (s:states): reachable(s) => Inv_2(s));

END ttm_invariants
```

Figure 4.2: The invariant specification of the example in Figure 2.1

As we can see from Figure 4.2, the first invariant *Inv_1* lists all possible values of the state variables. We employ the notion $(a1, ..., an)$ to represent the values of the state variables in the TTM. For example, $(0, 1, a)$ means $(u = 0, v = 1, x = a)$.

1. Starting from the initial state $(0, 1, a)$, if action γ happens, then the state is changed to $(0, 1, c)$. If action α happens, then the state is changed to $(1, 1, b)$.

2. From activity b, if action β happens, the state is changed to $(2, 0, d)$. If action γ happens, the state is changed to $(1, 1, e)$.

We list all the possible values of state variables in the first invariant *Inv_1*. After we have proved *Inv_1* by *lemma_1*, it will be easy to prove the *lemma_2* which states the state variable u is always less and equal to 2. [1]

For small safety critical specifications, a designer should have a good idea of the system's reachable state space. Writing down and verifying the reachability invariant in PVS should help the designer understand and debug the system under design.

4.3 Summary

By first specifying and proving all possible values of state variables as an invariant, we can reduce the proof effort for other invariants. By trying to prove that a predicate is an invariant, we can find errors in the TTM when this does not work. Proving the first invariant can greatly help us to prove other invariants. The complete procedure can become routine work for SELTSs and TTMs.

[1] To simplify the explanation, we did not take account the `tick` action and time requirements here

Chapter 5

State-Event Bisimulation and Equivalence

In this chapter, we first give the definitions of strong and weak state-event equivalence. Then we introduce the PVS files and templates which help us to specify strong and weak state-event equivalence in PVS. Finally, we give the verification of two examples of strong state-event equivalence, each for SELTSs and TTMs, and two examples of weak state-event equivalence, each for SELTSs and TTMs.

5.1 Strong State-Event Bisimulation and Strong State-Event Equivalence

Strong state-event bisimulation is a generalization of (event) bisimulation, introduced by[Par81] and used by Milner[Mil89]. Strong state-event bisimulation means that two systems are strongly equivalent if they have the same choices of transitions and the same state outputs after executing the same sequence of transitions. State-event bisimulation was introduced in [Law97] as the basis for equivalence of TTMs.

5.1.1 Strong State-Event Bisimulation

This definition adds event output maps in addition to the state output maps of state-event labeled transition systems (SELTS)[Law97]. Formally, *strong state-event*

bisimulation is defined as follows.

Definition 5.1.1 *Let* $\mathbb{Q}_1 = \langle Q_1, \Sigma_1, R_{\Sigma_1}, q_{10}, ps_1, pa_1 \rangle$, $\mathbb{Q}_2 = \langle Q_2, \Sigma_2, R_{\Sigma_2}, q_{20}, ps_2, pa_2 \rangle$ *be two SELTSs where* ps_1 *and* ps_2 *are state output mappings from the state types* Q_1 *and* Q_2 *(resp.) to a new state type* Q *(i.e.*$ps_1 : Q_1 \rightarrow Q$ *and* $ps_2 : Q_2 \rightarrow Q$*), and* pa_1 *and* pa_2 *are event mappings from the event types* Σ_1 *and* Σ_2 *(resp.) to a new event type* Σ *(i.e.* $pa_1 : \Sigma_1 \rightarrow \Sigma$ *and* $pa_2 : \Sigma_2 \rightarrow \Sigma$*). A relation* $S \subseteq Q_1 \times Q_2$ *is a strong state-event bisimulation iff* $(q_1, q_2) \in S$ *implies*

- $\forall \alpha_1 \in \Sigma_1$, *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* $pa_1(\alpha_1) = pa_2(\alpha_2)$).
- $\forall \alpha_2 \in \Sigma_2$, *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* $pa_1(\alpha_1) = pa_2(\alpha_2)$).

5.1.2 Strong Equivalence

Definition 5.1.2 *Two SELTSs* $\mathbb{Q}_1 = \langle Q_1, \Sigma_1, R_{\Sigma_1}, q_{10}, ps1, pa1 \rangle$ *and* $\mathbb{Q}_2 = \langle Q_2, \Sigma_2, R_{\Sigma_2}, q_{20}, ps2, pa2 \rangle$ *are said to be strongly equivalent* ($\mathbb{Q}_1 \backsim_{se} \mathbb{Q}_2$) *if and only if there is a strong state-event bisimulation* S *over* $Q_1 \times Q_2$ *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 [Law97].

5.2 Weak State-Event Bisimulation and Weak State-Event Equivalence

In some cases, strong equivalence is more discriminating than we would like because it "observes" unobservable transitions. We use "τ" to denote unobservable transitions. They are also called "internal" or "silent" transitions, because they do not produce event outputs. We claim it is important if two structures produce the same sequence of outputs, ignoring the unobservable transitions which do not generate new outputs. *Weak State-Event Bisimulation* were introduced in [Law97] as the basis for *Weak State-Event Equivalence* of TTMs.

5.2.1 State Invariant Transitive Closure

We introduce the concept of *state invariant transitive closure*, which forms the basis for the definition of *weak state-event bisimulation*.

Given a SELTS $\mathbb{Q} := \langle Q, \Sigma, R_\Sigma, q_0, ps, pa \rangle$ where Q is a countable set of states, Σ is a finite set of elementary actions or events, R_Σ is a set of binary relations on Q, $q_0 \in Q$ is the initial state, ps is the state output mapping from the state type Q to the common state output Q_{com}, and pa is the event mapping from the event type Σ to the common state output Σ_{com}. The special event τ represents unobservable events in Σ_{com}. If an action $\alpha \in \Sigma$ maps to $\tau \in \Sigma_{com}$ through pa, we consider α as an unobservable transition τ. When a τ transition happens, it does not produce an output event, though it may produce a change in the state output. We have two kinds of unobservable transtions (τ):

- If $q \xrightarrow{\tau} q'$ and $ps(q) = ps(q')$, then there is no change in the state output.
- If $q \xrightarrow{\tau} q'$ and $ps(q) \neq ps(q')$, then there is a change in state output when τ occurs even though no event is observed.

The *state invariant transitive closure* for a given SELTS $\mathbb{Q}$ is defined as the relation $\Rightarrow_{se}$ such that for $q, q' \in Q$, $q \Rightarrow_{se} q'$ iff for some $n \geq 0$, $\exists q_0, q_1, \ldots, q_n \in Q$, such that

- $q = q_0 \xrightarrow{\tau} q_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{n-1} \xrightarrow{\tau} q_n = q'$, and
- $ps(q_i) = ps(q) = ps(q')$ for $i = 0, 1, \ldots, n$.

To illustrate the concept of *state invariant transitive closure*, we give an example in Figure 5.1

Assume the variable u is the state output. The states are partitioned into three cells according to the value of u. So the *state invariant transitive closure* is the reflexive and transitive closure of the τ relation within each cell.

5.2.2 Weak State-Event Bisimulation

We use $q \overset{\beta}{\Rightarrow} q'$, where $\beta \in \Sigma$, to denote $q \Rightarrow_{se} q_1 \xrightarrow{\beta} q_2 \Rightarrow_{se} q'$, where $q, q', q_1, q_2 \in Q$ for a given SELTS $\mathbb{Q}$. It will be used in our definition of *weak state-event bisimulation*. The definition is very similiar to the strong version, but differs in that it adds event output

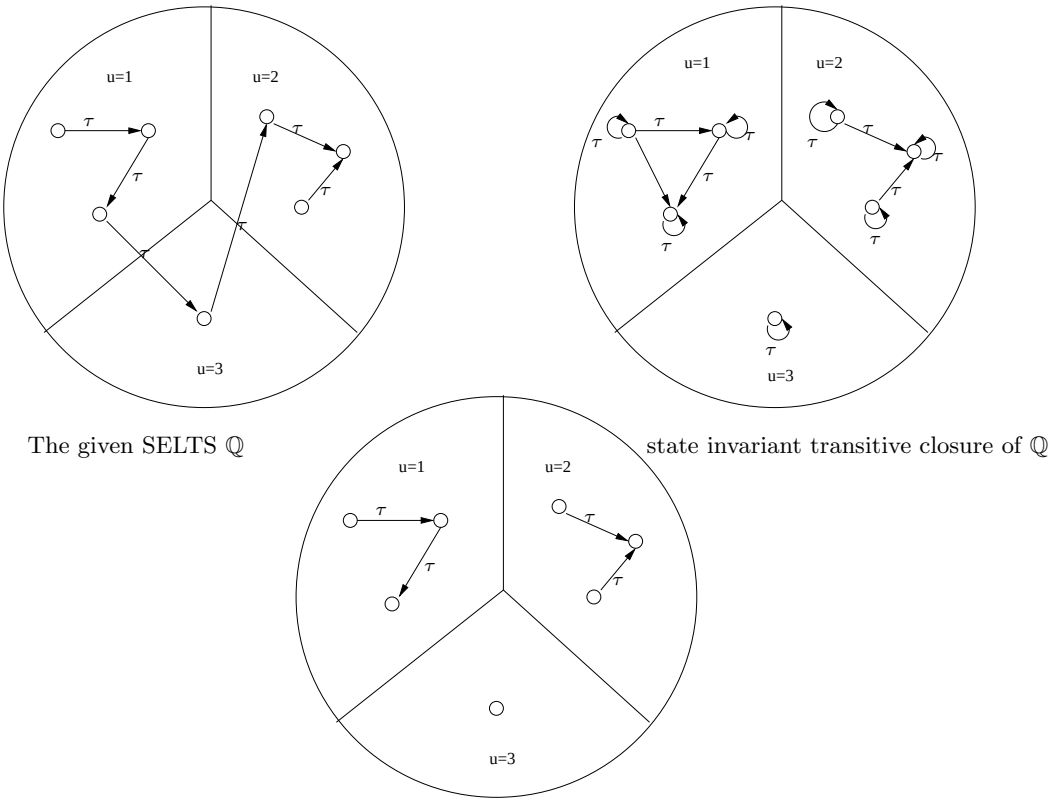


Figure 5.1: Illustrating state invariant transitive closure

maps in addition to the state output maps of state-event labeled transition systems (SELTS) [Law97]. Formally, *weak state-event bisimulation* is defined as follows.

Definition 5.2.1 *Let* $\mathbb{Q}_1 = \langle Q_1, \Sigma_1, R_{\Sigma_1}, q_{10}, ps_1, pa_1\rangle$ *and* $\mathbb{Q}_2 = \langle Q_2, \Sigma_2, R_{\Sigma_2}, q_{20}, ps_2, pa_2\rangle$ *be two SELTSs where* ps_1 *and* ps_2 *are state output mappings from the state types* Q_1 *and* Q_2 *(resp.) to a new state type* Q *(i.e.* $ps_1 : Q_1 \rightarrow Q$ *and* $ps_2 : Q_2 \rightarrow Q$*), and* pa_1 *and* pa_2 *are event mappings from the event types* Σ_1 *and* Σ_2 *(resp.) to a new event type* Σ *(i.e.* $pa_1 : \Sigma_1 \rightarrow \Sigma$ *and* $pa_2 : \Sigma_2 \rightarrow \Sigma$*). A relation* $S \subseteq Q_1 \times Q_2$ *is a weak state-event bisimulation if* $(q_1, q_2) \in S$ *implies*

- $\forall \alpha_1 \in \Sigma_1$, *whenever* $q_1 \stackrel{\alpha_1}{\rightarrow} q_1'$ *then*
  - $(\exists q_2' \in Q_2, \alpha_2 \in \Sigma_2$ *where* $q_2 \stackrel{\alpha_2}{\Rightarrow} q_2'$ *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* $pa_1(\alpha_1) = \tau)$
- $\forall \alpha_2 \in \Sigma_2$, *whenever* $q_2 \stackrel{\alpha_2}{\rightarrow} q_2'$ *then*
  - $(\exists q_1' \in Q_1, \alpha_1 \in \Sigma_1$ *where* $q_1 \stackrel{\alpha_1}{\Rightarrow} q_1'$ *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* $pa_2(\alpha_2) = \tau)$

5.2.3 Weak State-Event Equivalence

Definition 5.2.2 *Two SELTSs* $\mathbb{Q}_1 = \langle Q_1, \Sigma_1, R_{\Sigma_1}, q_{10}, ps_1, pa_1\rangle$ *and* $\mathbb{Q}_2 = \langle Q_2, \Sigma_2, R_{\Sigma_2}, q_{20}, ps_2, pa_2\rangle$ *are said to be weakly equivalent* ($\mathbb{Q}_1 \approx_{se} \mathbb{Q}_2$) *if and only if there is a weak state-event bisimulation* S *over* $Q_1 \times Q_2$ *such that* $(q_{10}, q_{20}) \in S$.

5.2.4 Observation Congruence

Observation Congruence as first defined by Milner[Mil89] is extended to TTMs in [Law92]. Informally, observation congruence is observation equivalence with the additional requirement that any τ actions from the initial state of one system must be

matched by a similar initial τ action in the other system. We can say that strong equivalence implies weak equivalence and observation congruence implies weak equivalence. Observation congruence provides us an assurance that we can exchange a part of a TTM with an equivalent part and still expect the resulting TTM to produce behavior equivalent to the original system. Please see [Law92] for details about synchronization of TTMs and observation congruence.

5.3 PVS Theories for Bisimulation and Equivalence

Because a TTM can always be expanded to a (possibly infinite state) SELTS, we do not need to give another definition of bisimulation and equivalence for TTMs. We use the same definition as for SELTSs. In the following sections, we will further formulate these definitions in PVS.

5.3.1 Theory `statetrans`

The theory `statetrans`(Figure 5.2) has two simple type parameters. The first parameter *OUT* is the state output type. The second parameter S is the state type. The body of the theory gives two formulations of the concept of "transitive closure". The first (due to Paul Y Gloess at E.N.S.E.R.B, France [1]) is the standard inductive definition of transitive reflexive closure. The second is our *state invariant transitive closure* which also requires the same state output. We use the second formulation in our theory `sesim`, which gives one direction in the definition of strong and weak state-event simulation.

5.3.2 The Theory `sesim` and `sebisim`

The theory `sesim` (Figure 5.3) gives one direction in the definition of state-event simulation. In the body of the theory, the function *wsesim*? uses the functions and relations in PVS for one direction in the definition of weak state-event simulation

[1] `http://dept-info.labri.u-bordeaux.fr/~gloess/pvs/`

```
state_trans [       OUT:TYPE+,
                    S:TYPE+
             ]
           : THEORY

BEGIN

%  transitive closure
  x,y,z:VAR S
  R:VAR PRED[[S,S]]
  P:VAR [S−>OUT]

   tc(R)(x,z): INDUCTIVE bool
    =    x=z OR R(x, z)
      OR (EXISTS (y: S): tc(R)(x, y) AND tc(R)(y, z)) ;

%  state_invariant_transitive_closure

   sitc(R: [S, S −> bool], P:[S−>OUT]): PRED[[S,S]] =
         tc(lambda x,y: R(x,y) & P(x)=P(y))

END state_trans
```

Figure 5.2: Theory `statetrans`

(5.2.2). The function *sesim*? gives one direction in the definition of strong state-event simulation (5.1.1).

The theory `sesim` has 13 parameters: S is the common state type which allows the two state types in the two structures to have mappings to some common variables, $S1$ is the first state type, and $S2$ is the second state type. Similiarly, A is the common event type which allows the two event types to map to common events, $A1$ is the first event type, and $A2$ is the second event type. The following are the other 7 function parameters:

- $pS1$ is the function which maps $S1$ to the common state output S.
- $pS2$ is the function which maps $S2$ to the common state output S.
- $pA1$ is the function which maps $A1$ to the common event output A.
- $pA2$ is the function which maps $A2$ to the common event output A.
- $d1$ is a predicate on $S1$, $A1$ and $S1$, which is true when the event $a_1 \in A1$ can transfer the first state $s_1 \in S1$ to the second $s_1' \in S1$.

- $d2$ is a predicate on $S2$, $A2$ and $S2$, which is true when the event $a_2 \in A2$ can transfer the first state $s_2 \in S2$ to the second $s_2' \in S2$.

- $Is\_tau?$ is a predicate on event set A, which is true when an event in A is a silent event τ. Typically, we use $pA1$ or $pA2$ to map the corresponding event in $A1$ or $A2$ to the common event A and then decide whether it is a τ event.

In the body of the theory `sesim`, we first import the theory `state_trans` which gives us the definition of *state_invariant_transitive_closure*. R is a predicate on the two state sets $S1$ and $S2$ which is true when the two state variables are related. It decides how the two states are related so that they can be considered as equivalent. $r1$ is a predicate on $S1 \times S1$. In our specification, it is true when the first state can reach the second state via an event which is a τ. $r2$ is a predicate on $S2 \times S2$. In our specification, it is true when the first state can reach the second state via an event which is a τ. *wsesim?* is the function which gives one direction in the definition of weak state-event simulation. It requires that any move from state $s1$ to a new state $s1a$ via transition $a1$ must be matched by a finite sequence of moves from $s2$, which produces the same observation ($pS1(s1a) = pS2(s2c)$) and leads to a state $s2c$ that is weakly similar to $s1a$. *Wsesim* is the type of those R which satisfy *wsesim?*. Similarly, *sesim?* is the function which gives one direction in the definition of strong simulation. It requires that any move from state $s1$ to a new state $s1a$ via transition $a1$ must be matched by a move from $s2$, which produces the same observation ($pS1(s1a) = pS2(s2a)$) and leads to a state $s2a$ that is strongly similar to $s1a$. *Sesim* is the type of those R which satisfy *sesim?*.

The theory `sebisim` (5.4) has the same parameters as the theory `sesim`. The predicates $start1$ and $start2$ are used to specify the initial states of the two structures. We import the theory `sesim` twice to give both directions in the definition of (weak and strong) simulation. *wsebisim?* is the function which gives weak state-event bisimulation. We use the function *converse* to exchange the domain and range of R so as to get both directions in the definition of weak state-event simulation. The function *converse* is included in the prelude of PVS. *Wsebisim* is the type of those R which consititute weak state-event bisimulation. The predicate *weakequivalence* adds the condition that the two initial states belong to R; similarly, *sebisim?* is the function which gives strong state-event bisimulation. *Sebisim* is the type of those R

```
sesim   [ S, S1, S2, A, A1, A2: TYPE+,
            pS1: [S1 -> S],
            pS2: [S2 -> S],
            pA1: [A1 -> A],
            pA2: [A2 -> A],
            d1: PRED[[S1,A1,S1]],
            d2: PRED[[S2,A2,S2]],
            Is_tau?:  PRED[A]
             ]
                                          : THEORY

  BEGIN

  IMPORTING state_trans

  R:VAR PRED[[S1,S2]];

  s1,s1a: VAR S1;
  s2,s2a,s2b,s2c,s2d:VAR S2;
  a1:VAR A1;
  a2:VAR A2;

  r1(s1,s1a):bool = (EXISTS (a1): d1(s1,a1,s1a) & Is_tau?(pA1(a1)));
  r2(s2,s2a):bool = (EXISTS (a2): d2(s2,a2,s2a) & Is_tau?(pA2(a2)));

            wsesim?(R):bool =
            FORALL s1,a1,s2: (FORALL s1a:(d1(s1,a1,s1a) & R(s1,s2) ) =>
                        (EXISTS s2a,a2,s2b,s2c:
                        (sitc(r2,pS2)(s2,s2a) & d2(s2a,a2,s2b) & sitc(r2,pS2)(s2b,s2c) &
                                    pS1(s1a)=pS2(s2c) & pA1(a1)=pA2(a2) & R(s1a,s2c))) OR
                        (EXISTS s2c: pS1(s1a)=pS2(s2c) and Is_tau?(pA1(a1)) and sitc(r2,pS2)(s2,s2c) and R(s1a,s2c)));

  Wsesim:TYPE = (wsesim?)

            sesim?(R):bool =
            FORALL s1,a1,s2: FORALL s1a: d1(s1,a1,s1a) & R(s1,s2) =>
                        EXISTS s2a,a2: (d2(s2,a2,s2a) & pS1(s1a)=pS2(s2a)
                                    &pA1(a1)=pA2(a2)
                                    & R(s1a,s2a));

  Sesim:TYPE = (sesim?)

END sesim
```

Figure 5.3: Theory `sesim` defining state-event simulation

which consititute strong state-event bisimulation. The predicate *strongequivalence* adds the condition that the two initial states belong to R.

```
sebisim    [ S, S1, S2, A, A1, A2: TYPE+,
                    pS1: [S1 -> S],
                    pS2: [S2 -> S],
                    pA1: [A1 -> A],
                    pA2: [A2 -> A],
                    d1: PRED[[S1,A1,S1]],
                    d2: PRED[[S2,A2,S2]],
                    Is_tau?:  PRED[A]
           ]
                                          : THEORY
BEGIN

 R:VAR PRED[[S1,S2]];

 start1: VAR PRED[S1];
 start2: VAR PRED[S2];

 IMPORTING sesim[S,S1,S2,A,A1,A2,pS1,pS2,pA1,pA2,d1,d2,Is_tau?]
 IMPORTING sesim[S,S2,S1,A,A2,A1,pS2,pS1,pA2,pA1,d2,d1,Is_tau?]

 wsebisim?(R):bool = wsesim?(R) & wsesim?(converse(R))

 Wsebisim:TYPE = (wsebisim?)

 Rwse:VAR Wsebisim
 s1:VAR S1
 s2:VAR S2

 weakequivalence(Rwse)(start1,start2)(s1,s2):bool =
          ((start1(s1)&start2(s2))=> Rwse(s1,s2))

 sebisim?(R):bool = sesim?(R) & sesim?(converse(R))

 Sebisim:TYPE = (sebisim?)

 Rse: VAR Sebisim

 strongequivalence(Rse)(start1,start2)(s1,s2):bool =
          ((start1(s1)&start2(s2))=> Rse(s1,s2))

END sebisim
```

Figure 5.4: Theory `sebisim`

5.4 PVS Template for Bisimulation and Equivalence

5.4.1 The Template

The PVS template for state-event bisimulation (Figure 5.5) provides a straightforward environment for specifying state-event bisimulation in PVS.

5.4.2 Instantiating the Template

In the template, we first import the theories `ttm1_invariants` and `ttm2_invariants` for TTMs; or `selts1_invariants` and `selts2_invariants` for SELTSs. Importing these two theories gives us the complete specification of the two structures $\mathbb{Q}_1$ and $\mathbb{Q}_2$. Then we define the event type A which is the common event type for the two structures. 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 $ps1, ps2, pa1, pa2$. The predicate $Is\_tau?$ is a fixed function. We do not need to instantiate it in the case of weak simulation. For strong bisimulation, we set this function to always return $false$. The predicates $dd1, dd2$ are provided by the specification of the two structures. They give us the possible transitions and their effects in the corresponding structure. By importing the theory `sebisim` with concrete parameters from our specification, we get all the definitions for state-event bisimulation. The relation RR is the crucial one for state-event bisimulation. If its type is specified as *Wsebisim*, then it represents a weak state-event bisimulation between the two structures. Similarly, if its type is specified as *Sebisim*, then it represents a strong state-event between the two structures. The user will be required to prove the TCC created by his declaration to confirm that it is a weak (or strong) state-event bisimulation. Finally, the lemma *weakequi* (or *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 (or strongly) state-event equivalent.

To illustrate an instantiation of the template, we use the template to specify four examples in each category. All are described in the following sections. The complete

```
sebisimul: THEORY

BEGIN

        IMPORTING ttm1_invariants
        IMPORTING ttm2_invariants

% Beginning of definition of intermediate states and actions etc.
% tau is used for weak bisimulation definition.

        action: DATATYPE
        BEGIN
          <...>
          tau: tau?
        END action

        state: TYPE = <...>

        s1,s10: VAR S_state;
        s2,s20: VAR I_state;
        a1: VAR S_action;
        a2: VAR I_action;

        ps1(s1): state = <...>

        ps2(s2): state = <...>

        pa1(a1): action =
                CASES a1 OF
                        <...>
                ENDCASES

        pa2(a2): action =
                CASES a2 OF
                        <...>
                ENDCASES

% For strong bisimulation, we set this function always return false.
        Is_tau?(a:action): bool = (a=tau)

        dd1(s1,a1,s10): bool =enabled(a1,s1)&(s10 = trans(a1,s1))

        dd2(s2,a2,s20): bool =enabled(a2,s2)&(s20 = trans(a2,s2))

        IMPORTING
        ttm_lib@sebisim[state,S_state,I_state,action,S_action,I_action,ps1,ps2,pa1,pa2,dd1,dd2,Is_tau?]

% For strong bisimulation, The data type is Sebisim.
        RR:Wsebisim = LAMBDA(s1,s2):
                        <...>

% For strong equivalence, The function name is strongequivalence.
        weakequi: LEMMA weakequivalence(RR)(start,start)(s1,s2);

END sebisimul
```

Figure 5.5: Template to specify two transitions systems are equivalent

PVS specification files and proofs are included in Appendices B, C, D and E

Table 5.1 below illustrates the information needed to fill in the template for state-event equivalence.

Table 5.1: Information required in template for weak equivalence in PVS

Name	**User Fills in**	**Comment**
action	Declaration of common actions for the two SELTSs or TTMs	Every action in the two SELTSs or TTMs is mapped to one of these actions
state	Declaration of type state which the two different states can project to	The common state
ps1	The mapping from the first state to the common state	Sometimes the common state is a part of the first state
ps2	The mapping from the second state to the common state	Sometimes the common state is a part of the second state
pa1	The mapping from the first action to the common action	This also decides which action is considered as a τ action
pa2	The mapping from the second action to the common action	This also decides which action is considered as a τ action
RR	The relation between the first and the second state	This serves as the bisimulation relation which defines the weak state-event equivalence of the two structures

5.5 Verifications

5.5.1 Verification of SELTS Strong State-Event Equivalence

The verification of SELTS strong state-event equivalence in PVS is a procedure which goes through the specifications of the two SELTSs and expands their definitions to verify that they are strongly equivalent. Figure 5.6 demonstrates examples of strong state-event bisimulation between SELTSs. We claim that $\mathbb{Q}_1$ is strongly equivalent to $\mathbb{Q}_2$ but not to $\mathbb{Q}_3$. In this example, we assume the output we are concerned with is of type *int*. We represent this variable in $\mathbb{Q}_1$ as u, and in $\mathbb{Q}_2$ and $\mathbb{Q}_3$ as v. As we can see from Figure 5.6, in every state in $\mathbb{Q}_1$, there is always a corresponding state in $\mathbb{Q}_2$ which has the same choice of next transitions and the same output mapping. For example, the state with activity b in $\mathbb{Q}_1$ has two possible actions τ and β which change the outputs to 3 and 4. We have two states with activities f and g in $\mathbb{Q}_2$ which have the same choice of actions and lead to the same output mapping. The same condition also holds for $\mathbb{Q}_2$.

Figure 5.6, on the other hand, shows that $\mathbb{Q}_1$ and $\mathbb{Q}_2$ are not strongly equivalent to $\mathbb{Q}_3$. In order to see this, note that the state with activity m does not correspond to any of the states in $\mathbb{Q}_1$ or $\mathbb{Q}_2$ because none of these states have only the choice of action τ.

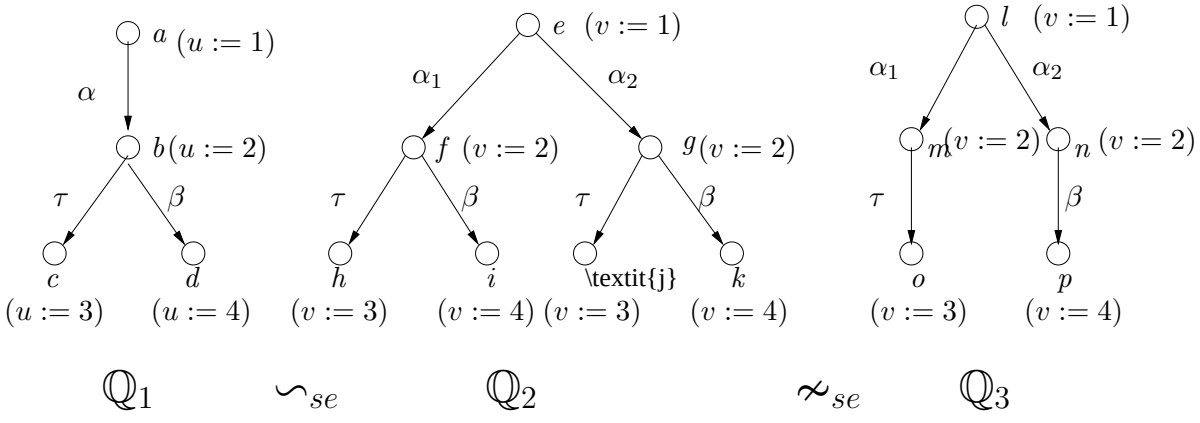


Figure 5.6: Examples of Strong State-Event Equivalence of SELTSs

The PVS specification files and proofs of strong equivalence in Figure 5.6 are included in Appendix B.

5.5.2 Verification of TTM Strong State-Event Equivalence

To prove the strong equivalence of two TTMs, we need to expand them to their corresponding SELTSs. Figure 5.7 demonstrates examples of strong state-event bisimulation of TTMs. We claim that $\mathbb{Q}_1$ is strongly equivalent to $\mathbb{Q}_2$ but not to $\mathbb{Q}_3$. Again we assume the output we are concerned with is of type *int*. We represent this variable in $\mathbb{Q}_1$ as u, in $\mathbb{Q}_2$ as v and in $\mathbb{Q}_3$ as w. As we can see from Figure 5.7, in every state in $\mathbb{Q}_2$, there is always a corresponding state in $\mathbb{Q}_1$ which has the same choice of next transitions and the same output mapping. For example, after one `tick`, the only possible action for $\mathbb{Q}_1$ is α, and the only possible action for $\mathbb{Q}_2$ is β, which has the same output mapping as α.

Figure 5.7, on the other hand, shows that $\mathbb{Q}_1$ and $\mathbb{Q}_2$ are not strongly equivalent to $\mathbb{Q}_3$. In order to see this, note that from the initial state in $\mathbb{Q}_3$, we must have two `ticks` before the action γ can happen. The state after the first `tick` does not correspond to any of the states in $\mathbb{Q}_1$ or $\mathbb{Q}_2$ because none of these states have a predecessor action of `tick` and a successor action of `tick`.

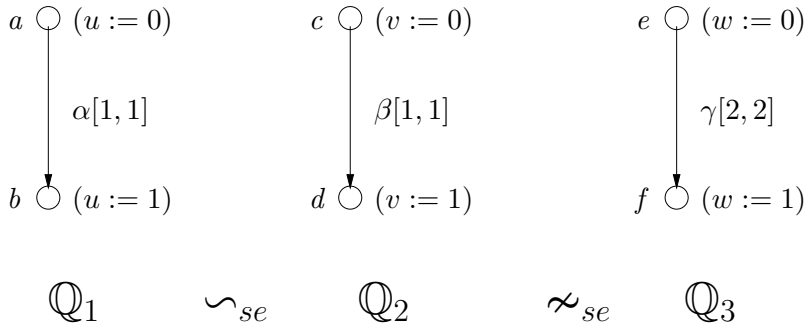


Figure 5.7: Examples of Strong State-Event Equivalence of TTMs

The PVS specification files and proofs of strong state-event equivalence in Figure 5.7 are included in Appendix C.

5.5.3 Verification of SELTS Weak State-Event Equivalence

The verification of SELTSs weak equivalence in PVS is a procedure which goes through the specifications of the two SELTSs and expands their definitions.

Figure 5.8 demonstrates an example of weak state-event equivalence of SELTSs. Again, we assume the output we are concerned with is of type *int*. We represent this variable in $\mathbb{Q}_1$ as u, and in $\mathbb{Q}_2$ as v. The actions α and β are not τ actions and the action τ is a unobservable action. We use the activity to represent the state. The relation S is defined as $\{(a,d),(b,e),(b,f),(c,g)\}$. As we can see from Figure 5.8, in every state in $\mathbb{Q}_1$, there is always a corresponding state in $\mathbb{Q}_2$ which has the same choice of next transitions and the same output mapping. For example, the state with activity b in $\mathbb{Q}_1$ has a possible action β which changes the output to 3. Correspondingly, the state with activity f in $\mathbb{Q}_2$ has the same choice of actions(β) and leads to the same output mapping. In the reverse direction, the situation is more complicated than for strong equivalence. The states with activity e (and action τ) do not have a corresponding state and action in $\mathbb{Q}_1$. But according to our definition of weak state-event bisimulation, there exists a state with activity b whose reflexive relation (b,b) created by the *state invariant transitive closure* of $\mathbb{Q}_1$ can be considered as its corresponding action. So we can say that these two structures are weakly equivalent.

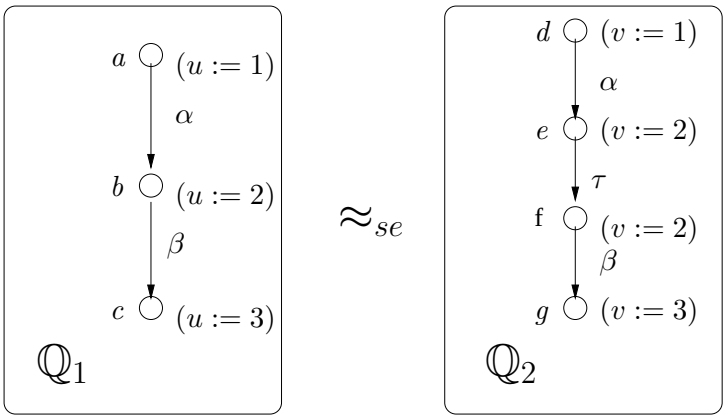


Figure 5.8: Example of Weak State-Event Equivalence of SELTSs

The PVS specification files and proofs of weak equivalence in Figure 5.8 are included in Appendix D.

5.5.4 Verification of TTM Weak State-Event Equivalence

To prove the weak equivalence of two TTMs, PVS will expand them to their corresponding SELTSs. Figure 5.9 demonstrates an example of weak state-event bisimulation of TTMs. We claim that $\mathbb{Q}_1$ is weakly equivalent to $\mathbb{Q}_2$. The special action `tick` is considered to be an unobservable action which means we don't care how many times `tick` happens between other observable actions. (If we were to consider the action `tick` as observable, then these two structures would not be weakly equivalent because the second `tick` in $\mathbb{Q}_2$ cannot be matched by any transition in $\mathbb{Q}_1$.) Again, the output we are concerned with is of type *int*. We represent this variable in $\mathbb{Q}_1$ as u, and in $\mathbb{Q}_2$ as v. As we can see from Figure 5.9, the first tick in $\mathbb{Q}_1$ can be matched by the first `tick` in $\mathbb{Q}_2$, and α can be matched by β or by the second `tick` plus β. Similarly, the first `tick` in $\mathbb{Q}_2$ can be matched by the first `tick` in $\mathbb{Q}_1$, and the second `tick` in $\mathbb{Q}_2$ can be matched by the reflexive relation (a, a) of the state before α. Finally, the action β in $\mathbb{Q}_2$ can be matched by α in $\mathbb{Q}_1$. Therefore we can say these two structures are weakly equivalent.

The PVS specification files and proofs of weak equivalence of Figure 5.9 are included in Appendix E.

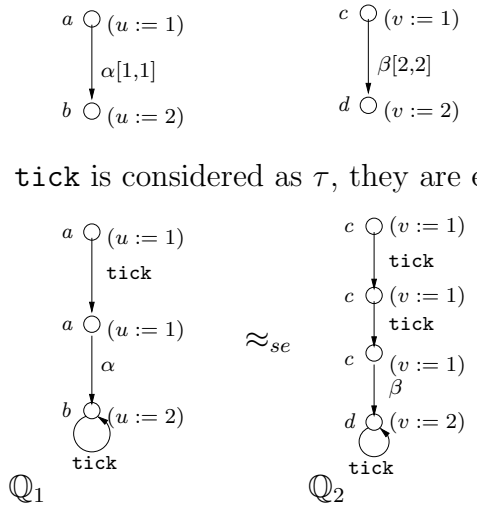


Figure 5.9: Example of Weak State-Event Equivalence of TTMs

5.6 Summary

By providing a theory for state-event equivalence and a template to assist in the specification of state-event equivalence, we can specify weak or strong equivalence of two SELTSs (or TTMs) in PVS in a straightforward manner. The examples we provide here also give some very useful hints on how to prove the state-event equivalence of SELTS (or TTMs). In the next chapter, we prove weak equivalence of TTMs for a non-trivial real-time software verification problem.

Chapter 6

Formalization and Verification of an Industrial Real-time Controller

Industrial reactive systems are often specified using a combination of timing diagrams, block diagrams, pseudo-code and careful English narrative. These methods have the advantage of being easily accessible to a wide community [Ost98]. But such a specification has the disadvantage that even the clearest informal descriptions are prone to omissions and ambiguities, just like uncompiled programs. Even the most experienced programmer cannot guarantee that his latest program will pass compilation without an error. Here we focus on whether real-time software will do what we want it to do.

6.1 The Delayed Reactor Trip System

The Delayed Reactor Trip (DRT) system was first described by Lawford [LW95]. It is a typical example from the process control industry. We borrow some descriptions of this example from [Law97] in the following sections.

When a certain set of events happens, we want to react to it in a timely fashion. In this case, 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.

Figure 6.1: Block Diagram for the Delayed Reactor Trip System

The desired action for the Delayed Reactor Trip system has the following 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 the Figure 6.2.

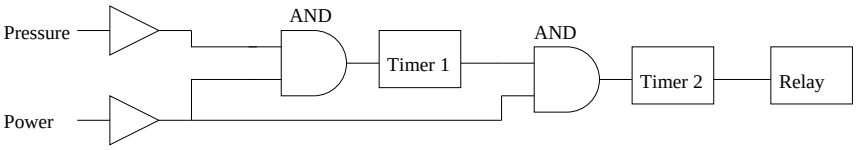


Figure 6.2: Analog Implementation of the Delayed Reactor Trip System

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 signalling 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. It is assumed that the input signals have been properly filtered and that the sampling rate is sufficiently fast to ensure proper control.

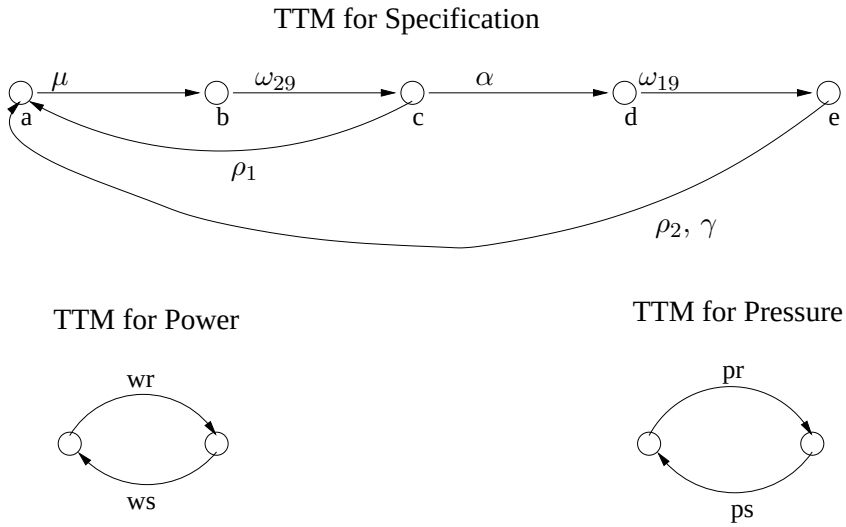
6.2 Modeling the DRT Specification

By modelling the specification as a TTM (Figure 6.3), we can clarify the ambiguities in the informal specification and ensure that the input/output actions are completely determined by the TTM.

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 the corresponding threshold, 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 [Law97], 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 6.3). The lower bound of (most of) the transitions in Figure 6.3 is 1, meaning that the enablement condition must be satisfied for at least one `tick` before the transition can occur. The upper bound of (some of) the transitions is ∞, meaning that if the enablement condition has been satisfied for one `tick`, then the state may transfer to another state at any time.



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

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

$\mathcal{T}$:

$\mu := (Power = HI \wedge Pressure = HI, [\,], 1, 1)$

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

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

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

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

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

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

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

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

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

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

Figure 6.3: SPEC: TTM representation of the DRT specification

6.3 Modelling the Microprocessor DRT Implementation

For the microprocessor DRT implementation (Figure 6.4), each time the microprocessor passes through the block of code represented by the pseudocode, it performs one of the group of operations identified by the transition name. (This is like the case statement in C simulated by 'if' and 'else' statements.) Identical groups of operations on the program variables are identified by identical transition names. All the transitions for the microprocessor DRT implementation are modeled as selfloops (Figure 6.14). The enablement conditions for these transitions are formed by taking the disjunction of the conditions specified by the 'if' statements for each occurence of a given transition name's program operations. As an example, let us consider e_{μ_1}, the enablement condition for μ_1:

1. The first occurrence of μ_1 happens if $(Power = HI \wedge c_1 = 0 \wedge c_2 = 0 \wedge Pressure = HI)$
2. The second occurrence of μ_1 happens if $(Power = HI \wedge (not(c_1 = 0)) \wedge (not(c_1 \geq 30)))$
3. The third occurrence of μ_1 happens if $((not(Power = HI)) \wedge (not(c_1 = 0)) \wedge (not(c_1 \geq 30)))$

Assuming the counter variable to be reset when it is equal to 0, and counter c_1 to be timed out when $c_1 \geq 30$, μ_1's enablement condition is:

$$\begin{aligned} e_{\mu_1} &= (Power = HI \wedge c_1 = 0 \wedge c_2 = 0 \wedge Pressure = HI) \vee \\ &\quad (Power = HI \wedge (not(c_1 = 0)) \wedge (not(c_1 \geq 30))) \vee \\ &\quad ((not(Power = HI)) \wedge (not(c_1 = 0)) \wedge (not(c_1 \geq 30))) \\ &= (Power = HI \wedge Pressure = HI \wedge c_1 = 0 \wedge c_2 = 0) \vee 1 \leq c_1 \leq 29 \end{aligned}$$

Similarly we can get the enablement conditions for the other transitions, as shown in Figure 6.14.

As the microprocessor scans through the code each cycle (100 ms), it picks out one of the labeled blocks of code. The block picked is the one whose enablement conditions are satisfied. The microprocessor then loops back to the beginning and re-evaluates all the enablement conditions in the next cycle. So each transition, except

```
If Power = HI then
    If counter c1 is reset then
        If counter c2 is reset then
            If Pressure = HI then
                increment c1                    Transition : μ1
            Endif
        Else
            If counter c2 timed out then
                reset c2                        Transition : γ
            Else
                increment c2                    Transition : μ2
                open Relay
            Endif
        Endif
    Else
        If counter c1 timed out then
            open Relay                          Transition : α
            reset c1
            increment c2
        Else
            increment c1                        Transition : μ1
        Endif
    Endif
Else
    If counter c1 is reset then
        If counter c2 is reset then
            close Relay                         Transition : β
        Else
            If counter c2 timed out then
                close Relay                     Transition : ρ2
                reset c2
            Else
                increment c2                    Transition : μ2
                open Relay
            Endif
        Endif
    Else If counter c1 timed out then
            reset c1                            Transition : ρ1
        Else
            increment c1                        Transition : μ1
        Endif
    Endif
Endif
```

Figure 6.4: Pseudocode for the microprocessor DRT implementation

for those which simulate the power and pressure, has a lower and upper time bound of 1.

6.4 Formalization of the DRT Specification

With the help of the theories and template we defined in Chapter 3, formalization of the TTM specification in PVS is very straightforward. We just follow the TTM representation of the specification in Figure 6.3 and input all the information into the template which we discussed in Section 3.4. In this way we get the formalization of the TTM in PVS.

We define the internal state as a record type:

s_internal_state: TYPE = [# Relay:bool, Power:bool, Pressure:bool #]

The timing requirements in the specification are directly implemented in the TTM. The definitions of *lower_bound* and *upper_bound* for the actions are given in Figure 6.5. As we can see, it is a direct translation from Figure 6.3. Both the *upper_bound* and *lower_bound* for the action ω_{29} are *twenty_nine*[1]. Similarly for the other actions.

The enablement condition of the state variables such as *Pressure* and *Power* is specified in the function *enabled_state*. It is a direct translation from Figure 6.3. For example, the action μ has a requirement that both *Pressure* and *Power* are HI. The function *enabled_state* is shown in Figure 6.6.

The requirement of the TTM graph is specified in the function *graph*. Again, it is a direct translation from Figure 6.3. For example, from the graph we know that the action α can only happen in activity c, so the enablement condition in the function *graph* for the action α is that the activity is equal to c. The function *graph* is shown in Figure 6.7.

To verify that we have the right formalization, we formulate the invariant of the specification by listing what we suppose the TTM will do at all the reachable states in the invariant. The invariant is shown in Figure 6.8. As we can see from Figure 6.8, we divide the invariant into 5 areas. Each area has its own activity value and possible values for the timers. In the initial state, the activity is set to a, all the timers are

[1]As our datatype *time* is the union of natural numbers and $\{\infty\}$ as we discussed in section 3.3.1. *twenty_nine* is the constant of datatype *time* for the natural number 29.

```
lower_bound (ac:S_nt_action): time =
        CASES ac OF
                mu: one,
                omega29: twenty_nine,
                alpha: one,
                omega19: nineteen,
                rho1: one,
                rho2: one,
                gamma: one,
                wr: one,
                ws: one,
                pr: one,
                ps: one
        ENDCASES

upper_bound (ac:S_nt_action): time =
        CASES ac OF
                mu: one,
                omega29: twenty_nine,
                alpha: one,
                omega19: nineteen,
                rho1: one,
                rho2: one,
                gamma: one,
                wr: infinity,
                ws: infinity,
                pr: infinity,
                ps: infinity
        ENDCASES
```

Figure 6.5: Time bounds of DRT specification

```
enabled_state (ac:S_nt_action, si:s_internal_state): bool =
        CASES ac OF
                mu: si‘Pressure and si‘Power,
                omega29: True,
                alpha: si‘Power,
                omega19: True,
                rho1: Not si‘Power,
                rho2: Not si‘Power,
                gamma: si‘Power,
                wr: si‘Power,
                ws: Not si‘Power,
                pr: si‘Pressure,
                ps: Not si‘Pressure
        ENDCASES;
```

Figure 6.6: Function *enabled_state* of DRT specification

```
graph (ac:S_nt_action, sa:activity):bool =
        CASES ac OF
                mu:  sa=a,
                omega29: sa=b,
                alpha: sa=c,
                omega19: sa=d,
                rho1: sa=c,
                rho2: sa=e,
                gamma: sa=e,
                wr: True,
                ws: True,
                pr: True,
                ps: True
        ENDCASES
```

Figure 6.7: Function *graph* of DRT specification

set to *zero*, *Power* and *Pressure* are set to *LO* and *Relay* is set to $CLOSED$. As we can see from Figure 6.3, starting from the initial state in activity a, the only possible value of the timer for μ is *zero* and *one*. The only possible value of the timers for ω_{29}, α, ω_{19}, ρ_1, ρ_2 and γ are *zero*. This is illustrated in area 1 in Figure 6.8. After action μ, the activity is changed to b. The possible value of the timer for ω_{29} is bigger than or equal to *zero* and less than or equal to $twenty\_nine (0 \leqslant T_{\omega_{29}} \leqslant 29)$. The only possible value of the timers for μ, α, ω_{19}, ρ_1, ρ_2 and γ is *zero*, as illustrated in area 2 in Figure 6.8. Similarly, we go through the whole TTM and get all the reachable states for it.

By proving the invariant, we confirm that our ideas about the TTM's behavior are correct. The invariant is also used when we define the weak equivalence relation between specification and implementation. The complete formalization of the specification TTM of Figure 6.3 is given in Appendix F.1.1. The verification of the full invariant for this TTM is given in Appendix F.2.1. By including the invariant in the definition of weak equivalence, we narrow down the state space needed by PVS to verify weak equivalence.

In the process of proving the invariant, we can find omissions in our specification of the invariant in PVS. By reviewing the unproved sequent and the invariant, we can pinpoint the omissions in our specification of the invariant in PVS. We now give an example to illustrate it.

When we first wrote the invariant according to the areas, we wrote down all the

```
%%%%%%%%%%%%%%%%%%%%%%%%
%                        %
%  Invariant of DRT specification  %
%                        %
%%%%%%%%%%%%%%%%%%%%%%%%

Inv_1(s:S_state): bool =

% Invariant @ activity a, area 1.

(s‘activity=a and (s‘action_time(mu)= zero or s‘action_time(mu)=one)
and s‘action_time(omega29)=zero and s‘action_time(alpha)=zero
and s‘action_time(omega19)=zero and s‘action_time(rho1)=zero
and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

% Invariant @ activity b, area 2

(s‘activity=b and s‘action_time(mu)= zero
and (s‘action_time(omega29)>=zero and s‘action_time(omega29)<=twenty_nine)
and s‘action_time(alpha)=zero and s‘action_time(omega19)=zero
and s‘action_time(rho1)=zero and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

% Invariant @ activity c, area 3

(s‘activity=c and s‘action_time(mu)= zero and s‘action_time(omega29)=zero
and ((s‘action_time(alpha)=zero and s‘action_time(rho1)=zero)
 or (s‘action_time(alpha)=one and s‘action_time(rho1)=zero)
 or (s‘action_time(alpha)=zero and s‘action_time(rho1)=one))
and s‘action_time(omega19)=zero
and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

% Invariant @ activity d, area 4

(s‘activity=d and s‘action_time(mu)= zero and s‘action_time(omega29)=zero
and s‘action_time(alpha)=zero
and (s‘action_time(omega19)>=zero and s‘action_time(omega19)<=nineteen)
and s‘action_time(rho1)=zero and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

% Invariant @ activity e, area 5

(s‘activity=e and s‘action_time(mu)= zero and s‘action_time(omega29)=zero
and s‘action_time(alpha)=zero and  s‘action_time(omega19)= zero and s‘action_time(rho1)=zero
and ((s‘action_time(rho2)=zero or s‘action_time(gamma)=zero)
or  (s‘action_time(rho2)=zero or s‘action_time(gamma)=one)
or  (s‘action_time(rho2)=one or s‘action_time(gamma)=zero)))
```

Figure 6.8: Invariant of DRT specification

possible values of the timers for each action and then "and"-ed them together. It works fine for areas 1,2 and 4. The invariant of area 5 is as shown in Figure 6.9.

```
% Invariant @ activity e, area 5

(s‘activity=e and
s‘action_time(mu)= zero and
s‘action_time(omega29)=zero and
s‘action_time(alpha)=zero and
s‘action_time(omega19)= zero and
s‘action_time(rho1)=zero and

% This part is not correct
((s‘action_time(rho2)=zero or s‘action_time(rho2)=one) and
(s‘action_time(gamma)=zero or s‘action_time(gamma)=one))
)
```

Figure 6.9: Invariant of area 5 which cannot be proved

When we tried to prove the reachable invariant as shown in Figure 6.9, we got the unproved sequent as shown in Figure 6.10. We can see from the sequent that the values of the timers for both γ and ρ_2 are $fintime(1)$. We know from Figure 6.3 that this is impossible. Only one of these two timers can be *one*, because the enablement conditions for these two actions cannot be true at the same time. The enablement condition for action γ is $Power = HI$, while that for action ρ_2 is $Power = LO$.

```
lemma_1.3.2.1.5.2.2 :

[−1]  s!1‘action_time(gamma) = fintime(1)
[−2]  s!1‘action_time(rho2) = fintime(1)
[−3]  e?(s!1‘activity)
[−4]  s!1‘action_time(mu) = fintime(0)
[−5]  s!1‘action_time(omega29) = fintime(0)
[−6]  s!1‘action_time(alpha) = fintime(0)
[−7]  s!1‘action_time(omega19) = fintime(0)
[−8]  s!1‘action_time(rho1) = fintime(0)
[−9]  tick?(a!1)
  |−−−−−−−
[1]   s!1‘basic‘Power
[2]   fintime(2) = fintime(0)
[3]   fintime(2) = fintime(1)
```

Figure 6.10: The unproved sequent in PVS

After we have changed the invariant as shown in Figure 6.11, we can prove it. In this corrected invariant, we exclude the situation that the values of the timers for both γ and ρ_2 are *one*.

```
% Invariant @ activity e, area 5

(s‘activity=e and
s‘action_time(mu)= zero and
s‘action_time(omega29)=zero and
s‘action_time(alpha)=zero and
s‘action_time(omega19)= zero and
s‘action_time(rho1)=zero and

% This part is correct and proved.
((s‘action_time(rho2)=zero or s‘action_time(gamma)=zero)
or  (s‘action_time(rho2)=zero or s‘action_time(gamma)=one)
or  (s‘action_time(rho2)=one or s‘action_time(gamma)=zero)))
```

Figure 6.11: Corrected invariant of area 5

6.5 Formalization of the DRT Implementation

Formalization and verification of the implementation is similiar to that of the specification. Following the TTM representation of the implementation in Figure 6.14, we input all the information into the template. The main difference is that we use counters c_1 and c_2 to record time rather than ω_{19} and ω_{29}. Unlike the case of the specification, the timing requirements are implemented by the counters c_1 and c_2. The timing requirements of ω_{19} and ω_{29} are not specified in the time bounds of the TTM in Figure 6.12. In addition to *Power*, *Pressure* and *Relay* in the definition of s_internal_state in section 6.4, The *internal_state* for the implementation has two more natural number variables c_1 and c_2 which are used to control how many ticks have happened.

internal_state: TYPE=[#Relay:bool,Power:bool,Pressure:bool,c1:nat,c2:nat#]

We use these two variables in the state variables to count the time requirements of ω_{19} and ω_{29} in the specification. As can be seen in Figure 6.12, we do not have ω_{19} and ω_{29} in the set of actions.

```
lower_bound (ac:I_nt_action): time =
    CASES ac OF
        mu1: one,
        alpha: one,
        mu2: one,
        rho1: one,
        rho2: one,
        gamma: one,
        wr: one,
        ws: one,
        pr: one,
        ps: one
    ENDCASES

upper_bound (ac:I_nt_action): time =
    CASES ac OF
        mu1: one,
        alpha: one,
        mu2: one,
        rho1: one,
        rho2: one,
        gamma: one,
        wr: infinity,
        ws: infinity,
        pr: infinity,
        ps: infinity
    ENDCASES
```

Figure 6.12: Time bounds of DRT implementation

The timing of ω_{19} and ω_{29} is implemented in the function *enabled_state* by variables c_1 and c_2 respectively. As can be seen in Figure 6.13, the actions μ_1, α, μ_2, ρ_1, ρ_2 and γ all have enablement conditions involving c_1 and/or c_2.

```
enabled_state (ac:I_nt_action, si:internal_state): bool =
    CASES ac OF
        mu1: (si‘Pressure and si‘Power and si‘c1=0 and si‘c2=0) or (si‘c1>=1 and si‘c1<=29),
        alpha: si‘Power and si‘c1>=30,
        mu2: (si‘c1=0 and si‘c2>=1 and si‘c2<=19),
        rho1: Not si‘Power and si‘c1>=30,
        rho2: Not si‘Power and si‘c1=0 and si‘c2>=20,
        gamma: si‘Power and si‘c1=0 and si‘c2>=20,
        wr: si‘Power,
        ws: Not si‘Power,
        pr: si‘Pressure,
        ps: Not si‘Pressure
    ENDCASES;
```

Figure 6.13: Function *enabled_state* of DRT implementation

The *graph* function is very simple. As we have only one activity a with selflooped transitions, thus we do not need to change the activity with any action. We can just say the graph (activity) condition is TRUE, i.e.,

```
graph(ac:I_nt_action, sa:I_activity): bool = TRUE
```

The complete formalization of the implementation of the TTM of Figure 6.14 is given in Appendix F.1.2. The whole procedure is a direct translation, with the help of the modeling environment developed in Chapter 3.

The verification of the full invariant for the DRT implementation (Figure 6.15) is given in Appendix F.2.2. As with the DRT specification, we divide the invariant into 5 areas. As we can see from Figure 6.22, in area 1, the counters c_1 and c_2 are equal to 0. If both *Power* and *Pressure* are *HI*, μ_1 will be executed. We transfer to area 2 with counter c_1 set to 1. In area 2, the enablement condition for μ_1 is always $True$ ($1 \leqslant c_1 \leqslant 29$). After μ_1 and *tick* has been executed 29 times, the counter c_1 increase to 29. Then μ_1 is executed again to transfer to area 3 with counter c_1 set to 30. In area 3, the value of *Power* determines whether ρ_1 or α should be executed. If *Power* is not HI anymore, we transfer back to area 1 through action ρ_1 and monitor *Power* and *Pressure* again. If *Power* is still HI, we transfer to area 4 through action α

which sets the counter c_2 to 1 and opens the *Relay*. In area 4, the enabled condition for μ_2 is always $True$ ($1 \leqslant c_2 \leqslant 19$). After μ_2 and *tick* have been executed 19 times, the counter c_2 increase to 19. Then μ_2 is executed one last time to transfer to area 5 with counter c_2 set to 20. In area 5, the value of *Power* determines whether ρ_2 or γ should be executed. If *Power* is not HI anymore, we transfer back to area 1 through action ρ_2 which closes the *Relay* and resets c_2 to 0. If *Power* is still HI, we transfer back to area 1 through action γ which leaves the *Relay* open and resets c_2 to 0.

As with the DRT specification, the verification of the invariant for the DRT implementation gives us confidence that the TTM is doing what we want. The process of proving the invariant is helpful in finding omissions in our model in PVS. By including this invariant in the definition of weak equivalence, we narrow down the state space needed by PVS to verify weak equivalence.

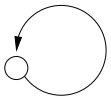
6.6 Verification of Weak Equivalence of Specification and Implementation

With the help of the theories and template defined in Chapter 5, we can define weak equivalence between the specification and implementation in PVS. We need to put all the required information into the template, as discussed in Section 5.4. As can be seen, the common *state* is defined in the same way as the *s_internal_state* in the DRT specification. The *state* in the DRT implementation is projected onto the common *state* by function $ps2$ which excludes the counters c_1 and c_2. In the function $pa1$, action *tick* is mapped to itself to preserve timing information. All other actions α, ρ_1, ρ_2, γ, wr, ws, pr, ps, ω_{29} and ω_{19} are mapped to unobservable τ event. Similiarly, in the function $pa2$, *tick* is mapped to itself. All other actions are mapped to τ. By doing this, we mean we only concern that there are the same number of *ticks* between the same state output change in these two TTMs.

The weak bisimulation relation is defined in Figure 6.17. We need to prove the TCC [2] produced by this definition. 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-

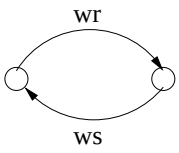
[2]Type-correctness condition that needs to be proved to establish the type-consistency of a PVS specification

TTM for Implementation

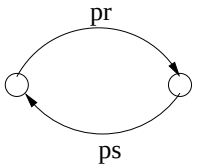


Selfloop($\mu_1, \mu_2, \alpha, \beta, \gamma, \rho_1, \rho_2$)

TTM for Power



TTM for Pressure



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

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

$\mathcal{T}$:

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

$\mu_2 := (c_1 = 0 \wedge 1 \leq c_2 \leq 19, [c_2 : c_2 + 1, Relay : OPEN], 1, 1)$

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

$\rho_1 := (Power = LO \wedge c_1 \geq 30, [c_1 : 0], 1, 1)$

$\rho_2 := (Power = LO \wedge c_1 = 0 \wedge c_2 \geq 20, [c_2 : 0, Relay : CLOSED], 1, 1)$

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

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

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

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

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

Where

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

Figure 6.14: PROG: TTM representation of DRT implementation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Invariant of DRT implementation                %
%                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 Inv_2(s:I_state) : bool =

% Invariant @ c1=0 and c2=0, area 1

(s‘basic‘c1=0 and s‘basic‘c2=0
and (s‘action_time(mu1)= zero or s‘action_time(mu1)=one)
and s‘action_time(mu2)= zero and s‘action_time(alpha)=zero
and s‘action_time(rho1)=zero and s‘action_time(rho2)= zero
and s‘action_time(gamma)=zero ) or

% Invariant @ 1<= c1 <= 29 and c2=0, area 2

((s‘basic‘c1>=1 and s‘basic‘c1<=29)
and s‘basic‘c2=0
and (s‘action_time(mu1)= zero or s‘action_time(mu1)= one)
and s‘action_time(mu2)= zero and s‘action_time(alpha)=zero
and s‘action_time(rho1)=zero and s‘action_time(rho2)= zero
and s‘action_time(gamma)=zero ) or

% Invariant @ c1=30 and c2=0, area 3

(s‘basic‘c1=30 and s‘basic‘c2=0
and s‘action_time(mu1)= zero  and s‘action_time(mu2)= zero
and (    (        s‘action_time(alpha)=zero and s‘action_time(rho1)=zero)
                  or (s‘action_time(alpha)=zero and s‘action_time(rho1)=one)
                  or (s‘action_time(alpha)=one  and  s‘action_time(rho1)=zero))
and s‘action_time(rho2)= zero and s‘action_time(gamma)=zero ) or

% Invariant @ c1=0 and 1<=c2<=19, area 4

(s‘basic‘c1=0
and (s‘basic‘c2>=1 and s‘basic‘c2<=19)
and s‘action_time(mu1)= zero
and (s‘action_time(mu2)= zero or s‘action_time(mu2)= one)
and s‘action_time(alpha)=zero and s‘action_time(rho1)=zero
and s‘action_time(rho2)= zero and s‘action_time(gamma)=zero ) or

% Invariant @ c1=0 and c2=20, area 5

(s‘basic‘c1=0 and s‘basic‘c2=20 and s‘action_time(mu1)= zero
and s‘action_time(mu2)= zero  and s‘action_time(alpha)=zero
and s‘action_time(rho1)=zero
and ((        s‘action_time(rho2)= zero and s‘action_time(gamma)=zero)
              or (s‘action_time(rho2)= zero and s‘action_time(gamma)=one)
              or (s‘action_time(rho2)= one and s‘action_time(gamma)=zero)))
```

Figure 6.15: Invariant of DRT implementation

```
action: DATATYPE
BEGIN
 tick: tick?
  tau: tau?
END action

s1,s10: VAR S_state;
s2,s20: VAR I_state;
a1,a11: VAR S_action;
a2,a22: VAR I_action;

state: TYPE = s_internal_state

ps1(s1): state = s1‘basic

ps2(s2): state = (# Relay:=s2‘basic‘Relay,
                    Power:=s2‘basic‘Power,
                    Pressure:=s2‘basic‘Pressure #)

pa1(a1): action =
        CASES a1 OF
                tick: tick,
                mu: tau,
                omega29: tau,
                alpha: tau,
                omega19: tau,
                rho1: tau,
                rho2: tau,
                gamma: tau,
                wr: tau,
                ws: tau,
                pr: tau,
                ps: tau
        ENDCASES

pa2(a2): action =
        CASES a2 OF
                tick: tick,
                mu1: tau,
                alpha: tau,
                mu2: tau,
                rho1: tau,
                rho2: tau,
                gamma: tau,
                wr: tau,
                ws: tau,
                pr: tau,
                ps: tau
        ENDCASES
```

Figure 6.16: Definition of states and actions relationships for bisimulation

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.

We further expand the state transition diagram of the specification in Figure 6.21 and the implementation in Figure 6.22. As can be seen from these diagrams, we divide the expanded TTM representations of the DRT specification and implementation into five areas just as we did in invariant proving. In each area, only a small number of actions are possible. For example, in area 1 of the specification, only actions μ, pr, ps, wr, ws and *tick* are possible. Correspondly, in area 1 of the implementation, only actions μ_1, pr, ps, wr, ws and *tick* are possible. The action μ in the specification corresponds to the action μ_1 in the implementation. All other actions correspond to their same name counterparts.

We also divide the proof into five areas. These five areas are the same as in the invariant. First, we prove that only the set of possible actions is allowed in each area. Then we prove that each action in this set has a corresponding action in the implementation. The proofs for areas 1,3,5 are very similar, while the proofs for areas 2 and 4 are more complicated. Areas 2 and 4 are further expanded into Figures 6.23 and 6.24 respectively to illustrate the relation between specification and implementation. The correspondence between areas 2 and 4 are illustrated by the dotted lines between the expanded TTMs. For example, in Figure 6.23 for area 4, for the DRT specification, only actions pr, ps, wr, ws, *tick* and ω_{29} are possible, while for the DRT implementation, only actions pr, ps, wr, ws, *tick* and μ_1 are possible. It is obvious that pr, ps, wr, ws and *tick* in the DRT specification correspond to their same name counterparts in the DRT implementation. The action ω_{29} corresponds to the last μ_1 in area 4. All other μ_1's corresponds to self-loop actions created by the state invariant transitive closure that we introduced in section 5.2.1.

As we go further in the proof, PVS use more memory in the machine and garbage collection becomes more frequent. The proof becomes very slow. So we copy some sequents of the proof in areas 2 and 4 and replace some variable names to make a separate lemma. After we prove this lemma, we re-introduce it into the proof and instantiate it with the appropriate variable name to prove the sequent. This allows us to decompose the task of the proof into several small parts which greatly relieves the load to the machine and speeds up the procedure. The lemmas such as sim12_2,

```
RR:Wsebisim = LAMBDA(s1,s2):

        ((Inv_11(s1) & Inv_21(s2) & Inv_1(s1) & Inv_2(s2))
       & s1‘action_time(alpha)=s2‘action_time(alpha)
       & s1‘action_time(rho1)=s2‘action_time(rho1)
       & s1‘action_time(rho2)=s2‘action_time(rho2)
       & s1‘action_time(gamma)=s2‘action_time(gamma)
       & s1‘action_time(wr)=s2‘action_time(wr)
       & s1‘action_time(ws)=s2‘action_time(ws)
       & s1‘action_time(pr)=s2‘action_time(pr)
       & s1‘action_time(ps)=s2‘action_time(ps)
       & ps1(s1) = ps2(s2))
       &

       % area 1
       ((s1‘activity=a and (s2‘basic‘c1=0 &s2‘basic‘c2=0) and
        (s1‘action_time(mu)=s2‘action_time(mu1)))
       or

       % area 2
       (s1‘activity=b and s2‘basic‘c1>=1 and s2‘basic‘c1<=29 and
       ((s1‘action_time(omega29)=fintime(s2‘basic‘c1) and s2‘action_time(mu1)=one) or
       (s1‘action_time(omega29)+ one =fintime(s2‘basic‘c1) and s2‘action_time(mu1)=zero)))
       or

       % area 3
       (s1‘activity=c and s2‘basic‘c1=30 )
       or

       % area 4
       (s1‘activity=d and s2‘basic‘c2>=1 and s2‘basic‘c2<=19 and
       ((s1‘action_time(omega19)=fintime(s2‘basic‘c2) and s2‘action_time(mu2)=one) or
       (s1‘action_time(omega19)+one=fintime(s2‘basic‘c2) and s2‘action_time(mu2)=zero)))
       or

       % area 5
       (s1‘activity=e and s2‘basic‘c2=20))


weakequi: LEMMA weakequivalence(RR)(start,start)(s1,s2);
```

Figure 6.17: The relation which defines weak state-event equivalence

sim12_4, sim21_2121, sim21_221, sim21_4121 and sim21_42 are created to deal with the problem of "memory exhausted". The number in the name of the lemma is the branch number of the proof, so we know when we need to use these lemmas.

The unproved sequents in the proof help us to refine our definition of the bisimulation relation and also find any differences between the DRT specification and implementation. For example, we got 8 similar unproved sequents when we tried to prove the bisimulation in area 4. The first sequent is show in Figure 6.18. It required us to prove that the actions ω_{19} and μ_2 give the same output. As we checked with the DRT specification in Figure 6.3, the action ω_{19} does not change the state of *Relay*. All other actions which can happened after α and before ω_{19} do not change the state of *Relay* either. So after α opens *Relay*, it should stay open before ρ_2 is executed, although this is not enforced by the action ω_{19}. However, according to the DRT implementation in Figure 6.14, after α opens *Relay*, the action μ_2 keeps opening it while increasing the counter c_2. This is added as a safety precaution. If other programs or manual operations close *Relay*, it will be opened again by the action μ_2. This is shown in Figure 6.19.

However, these two TTMs should still be considered as weakly equivalent, since we are comparing them in an isolated environment. To solve this problem, we added another invariant Inv_21 (shown in Figure 6.20) to further narrow the state space. First, we proved this invariant to ensure our idea is correct. Then, we added this invariant to the relation RR and tried to prove it again. We could then prove all the remaining 8 sequents.

The complete proof of weak equivalence has been omitted due to length; it is available on request.

```
sim12_4.1.2.1.1.31 :

{−1}  s1!1‘action_time(omega19) = fintime(s2!1‘basic‘c2)
{−2}  s2!1‘action_time(mu2) = one
{−3}  omega19?(a1!1)
[−4]  enabled_general(omega19, s1!1)
[−5]  enabled_time(omega19, s1!1)
{−6}  d?(s1!1‘activity)
[−7]  s2!1‘basic‘c2 >= 1
[−8]  s2!1‘basic‘c2 <= 19
[−9]  Inv_11(s1!1)
[−10] Inv_21(s2!1)
[−11] Inv_1(s1!1)
[−12] Inv_2(s2!1)
[−13] s1!1‘action_time(alpha) = s2!1‘action_time(alpha)
[−14] s1!1‘action_time(rho1) = s2!1‘action_time(rho1)
[−15] s1!1‘action_time(rho2) = s2!1‘action_time(rho2)
[−16] s1!1‘action_time(gamma) = s2!1‘action_time(gamma)
[−17] s1!1‘action_time(wr) = s2!1‘action_time(wr)
[−18] s1!1‘action_time(ws) = s2!1‘action_time(ws)
[−19] s1!1‘action_time(pr) = s2!1‘action_time(pr)
[−20] s1!1‘action_time(ps) = s2!1‘action_time(ps)
[−21] ps1(s1!1) = ps2(s2!1)
 |−−−−−−−
[1]   (tick?(omega19))
{2}   ps1(trans(omega19, s1!1)) = ps2(trans(mu2, s2!1))
```

Figure 6.18: The first unproved sequent in area 4

In TTM for Specification

$$\omega_{19} := (True, [\,], 19, 19)$$
$$\alpha := (Power = HI, [Relay : OPEN], 1, 1)$$

In TTM for Implementation

$$\mu_2 := (c_1 = 0 \wedge 1 \leq c_2 \leq 19, [c_2 : c_2 + 1, Relay : OPEN], 1, 1)$$
$$\alpha := (Power = HI \wedge c_1 \geq 30, [c_1 : 0, c_2 : c_2 + 1, Relay : OPEN], 1, 1)$$

Figure 6.19: Comparision of action ω_{19} and μ_2

```
Inv_21(s:I_state): bool =  (s‘basic‘c2 >= 1 => s‘basic‘Relay = True)

LEMMA2_2: LEMMA (FORALL (s:I_state): reachable(s) => Inv_21(s));
```

Figure 6.20: The invariant Inv_21

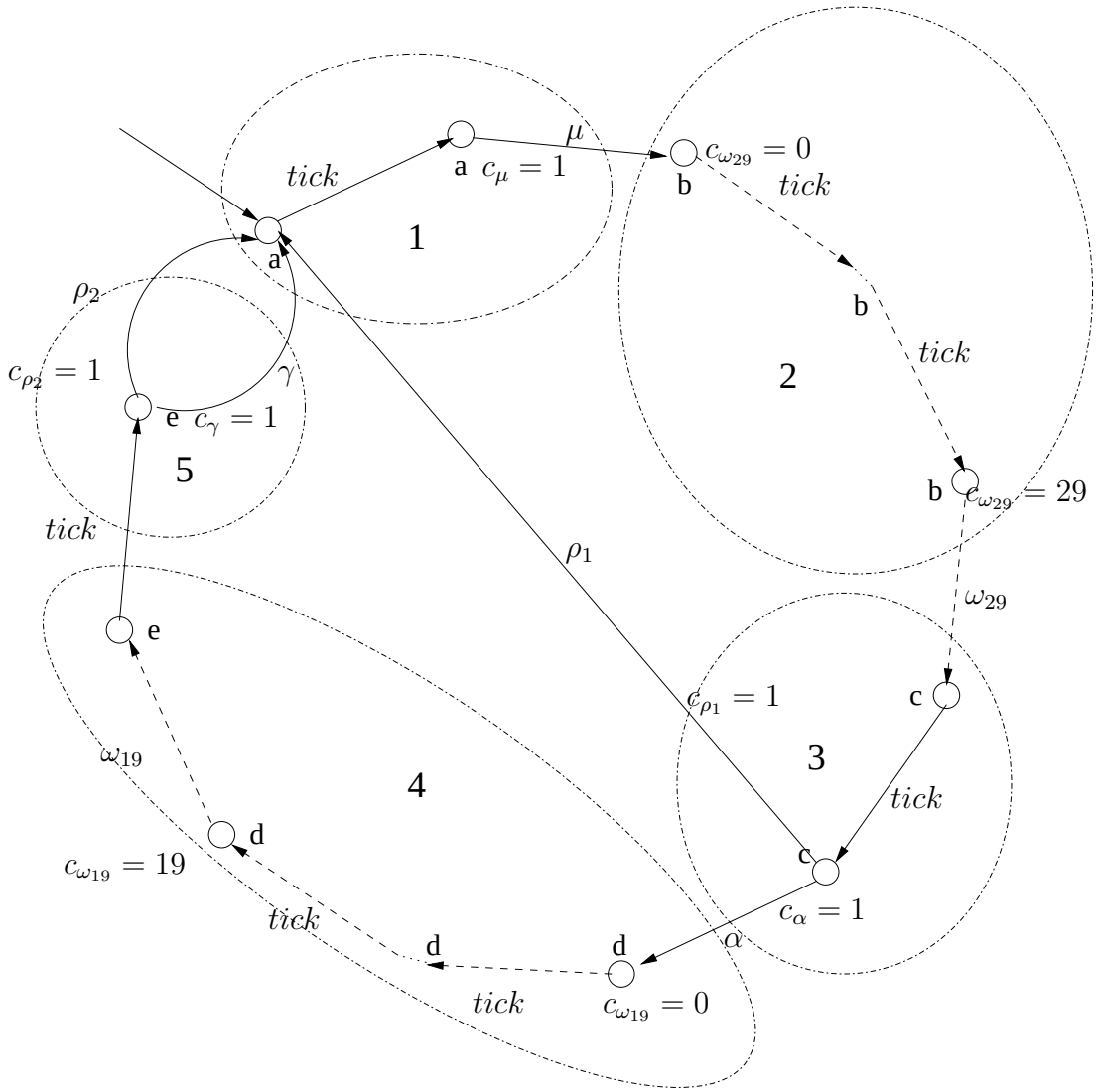


Figure 6.21: Expanded TTM representation of DRT specification

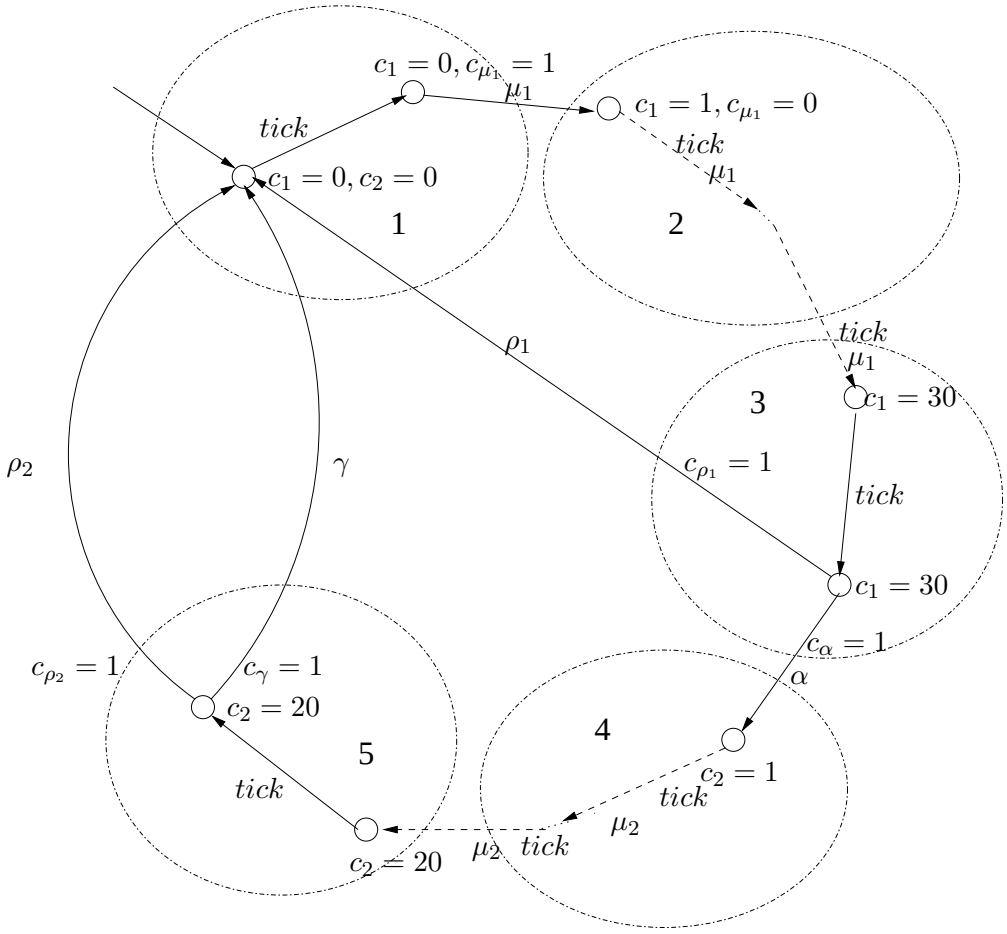


Figure 6.22: Expanded TTM representation of DRT implementation

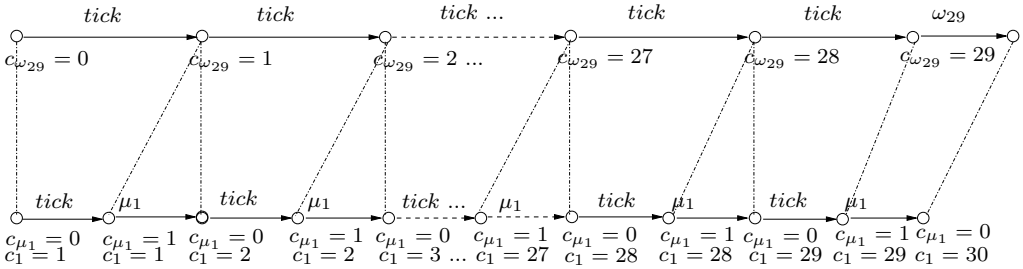


Figure 6.23: Detail of expanded TTM representation of DRT specification

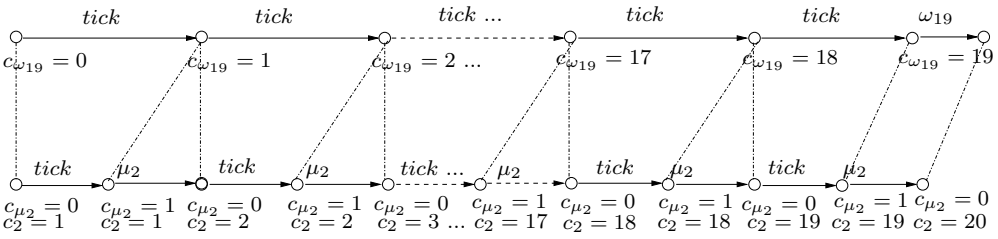


Figure 6.24: Detail of expanded TTM representation of DRT implementation

Chapter 7

Conclusion

In this chapter, we will discuss the results of the thesis, limitations and future work, as well as conclusions.

7.1 Results

The thesis gives the definitions of invariants, strong equivalence and weak equivalence for SELTSs and TTMs in PVS. It also provides a unified modeling environment for SELTSs and TTMs in PVS which allows the user to specify SELTSs and TTMs easily in PVS. Further, it illustrates the use of the modeling environment in PVS to specify and prove invariants, weak equivalence and strong equivalence of SELTSs and TTMs by providing an example in each category. Finally, we use our modeling environment to formalize and verify the correctness of an industrial real-time controller.

7.2 Limitations and Future Research

- In our modeling environment, the proof was done manually. It required quite a lot of interactions between the user and PVS. In the future, we could develop some strategies to simplify the proof procedure. It is hoped that we can simplify the proof by some powerful strategies. Complete automatic proof procedure is still out of reach. We believe that combining the theorem prover and model checker may offers the best solution.

- Compositional verification of real-time software by modularization is an increasingly important area of research. *Observation Congruence* as first defined by Milner[Mil89] and extended to TTMs in[Law92] can be a good basis for compositional verification of real-time software by modularization in PVS.

7.3 Conclusion

This thesis presents a method for formally analyzing control software in PVS. A unified modeling environment for SELTSs and TTMs is presented, and an example in each category, as well as an industrial real-time controller, are provided.

Appendix A

Invariants of TTM

A.1 Formalization of TTM in Figure 2.1

A.1.1 Formalization

```
ttm_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

IMPORTING time_thy

activity_t: TYPE = {a,b,c,d,e}

% action_t is actions with Tick.

action_t: DATATYPE
BEGIN
 Tick: Tick?
 alpha: alpha?
 beta: beta?
 gamma: gamma?
END action_t

% action_t is actions without Tick

% action_t: TYPE = { action:Faction_t | action/=Tick}

% A predicate to tell wether the action is Tick.

        Is_Tick?(a: action_t) : bool = (a = Tick)

% state variables

internal_state_t: TYPE = [# u: int,v: int #]

% Auxiliary functions
lower_bound(ac:action_t):time =
CASES ac OF
Tick: zero,
alpha: zero,
```

```
beta: two,
gamma: two
ENDCASES


upper_bound(ac:action_t):time =
CASES ac OF
Tick: infinity,
alpha: one,
beta: infinity,
gamma: two
ENDCASES

%   preconditions

    enabled_state (ac:action_t, w:internal_state_t):bool =
 CASES ac OF
Tick: True,
alpha: (w`u>=0),
beta: True,
gamma: (w`v>=0)
 ENDCASES;

    graph (ac:action_t, sa:activity_t):bool =
         CASES ac OF
Tick: True,
alpha: (sa = a),
beta: (sa = b),
gamma: (sa=a or sa=b)
 ENDCASES

    IMPORTING ttm[activity_t,internal_state_t,action_t,Is_Tick?,
lower_bound,upper_bound,enabled_state, graph]

%   transitions

    trans (ac:action_t, s:states):states =
         CASES ac OF
Tick:  s WITH [action_time:= update_clocks(s)],
alpha: s WITH [activity:=b, basic:=s`basic WITH [u:=s`basic`u+s`basic`v],
action_time:=reset_clocks(ac,s WITH [basic:=s`basic
WITH[u:=s`basic`u+s`basic`v], activity:=b])],
beta:  s WITH [basic:=s`basic WITH [u:=s`basic`u+1, v:=s`basic`v-1], activity:=d,
action_time:=reset_clocks(ac,s WITH [basic:=s`basic
WITH[u:=s`basic`u+1,v:=s`basic`v-1],activity:=d])],
gamma:  If (s`activity=a) THEN s WITH [activity:=c,
action_time:=reset_clocks(ac,s WITH [activity:=c]) ]
       ELSIF (s`activity=b) THEN s WITH [activity:=e,
action_time:=reset_clocks(ac,s WITH [activity:=e]) ]
ELSE s
        ENDIF
 ENDCASES

% define initial state

start (s:states):bool =
  ( s = (# activity:=a,
   basic:= (# u:=0, v:=1 #),
   action_time:=(LAMBDA (a:action_t): zero)
#)
  )

IMPORTING ttm_lib@machine[states,action_t,enabled,trans,start]


END ttm_decls
```

A.2 Invariants and proof

```
ttm_invariants   : THEORY

   BEGIN

   IMPORTING  ttm_decls
         IMPORTING  ttm_unique_aux
         IMPORTING  ttm_rewrite_aux_1
         IMPORTING  ttm_rewrite_aux_2

  Inv_1(s:states) : bool = (s`activity=a and s`basic`u=0 and s`basic`v=1) or
   (s`activity=c and s`basic`u=0 and s`basic`v=1) or
   (s`activity=b and s`basic`u=1 and s`basic`v=1) or
     (s`activity=e and s`basic`u=1 and s`basic`v=1) or
   (s`activity=d and s`basic`u=2 and s`basic`v=0)

  Inv_2(s:states): bool = (s`basic`u<=2);

  s: VAR states

  lemma_aux1: LEMMA Inv_1(s) => Inv_2(s);

  lemma_1: LEMMA (FORALL (s:states): reachable(s) => Inv_1(s));

  lemma_2: LEMMA (FORALL (s:states): reachable(s) => Inv_2(s));

  END ttm_invariants
```

```
Proof scripts for file ttm_invariants.pvs:


ttm_invariants.lemma_aux1: proved - complete [0](n/a s)

("" (SKOLEM!) (FLATTEN) (EXPAND "Inv_1") (EXPAND "Inv_2") (ASSERT))


ttm_invariants.lemma_1: proved - complete [0](11.22 s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_1")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (GRIND)) ("3" (GRIND))))


ttm_invariants.lemma_2: proved - complete [0](n/a s)

(""
 (LEMMA "lemma_aux1")
 (LEMMA "lemma_1")
 (SKOLEM!)
 (INST -1 "s!1")
 (INST -2 "s!1")
 (FLATTEN)
 (ASSERT))
```

Appendix B

Strong Equivalence of SELTSs

B.1 Formalization of SELTSs in Figure 5.6

B.1.1 Formalization of $\mathbb{Q}_1$

```
selts1decls: THEORY

BEGIN
ttm_lib: LIBRARY = "../ttm_lib"

S_activity_t: TYPE = {a,b,c,d}

% S_action_t

S_action_t: DATATYPE
BEGIN
 alpha: S_alpha?
 beta: S_beta?
 tau: S_tau?
END S_action_t

% state variables

internal_state_t: TYPE = [# u: int #]

 IMPORTING  ttm_lib@states[S_activity_t,S_action_t,internal_state_t]

      S_state_t: TYPE = states [S_activity_t,S_action_t,internal_state_t]

%   preconditions

    enabled_state (ac:S_action_t, s:S_state_t):bool =
 CASES ac OF
alpha: True,
beta: True,
tau: True
 ENDCASES;

    graph (ac:S_action_t, s:S_state_t):bool =
        CASES ac OF
alpha: (s`activity = a),
beta: (s`activity = b),
```

```
tau: (s‘activity = b)
 ENDCASES

    enabled_general(ac:S_action_t,s:S_state_t):bool =
enabled_state(ac,s) & graph(ac,s)

%   transitions
    trans (ac:S_action_t, s1:S_state_t) :S_state_t  =
         CASES ac OF
alpha:  s1 WITH [ activity:=b, basic:= s1‘basic WITH[u:=2]],
beta:   s1 WITH [ activity:=d, basic:= s1‘basic WITH[u:=4]],
tau:    s1 WITH [ activity:=c, basic:= s1‘basic WITH[u:=3]]
 ENDCASES

% define enabled S_action_t

enabled (ac:S_action_t, s:S_state_t):bool =
         enabled_general(ac,s);

% define initial state

start (s:S_state_t):bool =
  ( s = (# activity:=a,
   basic:= (# u:=1 #)
#)
  )

  IMPORTING ttm_lib@machine[S_state_t,S_action_t,enabled,trans,start]

END selts1decls
```

B.1.2 Formalization of $\mathbb{Q}_2$

```
selts2decls: THEORY

BEGIN
ttm_lib: LIBRARY = "../ttm_lib"

% --------------Begin of the Second TTM declaration= Implementation --------------------------------|

% type and function declarations:

I_activity_t: TYPE = {e,f,g,h,i,j,k};

% I_action_t

I_action_t: DATATYPE
BEGIN
 alpha_1: alpha_1?
 alpha_2: alpha_2?
    beta: beta?
     tau: tau?
END I_action_t

% state variable
internal_state_t:TYPE=[# v: int #]

IMPORTING  ttm_lib@states[I_activity_t,I_action_t,internal_state_t]

     I_state_t: TYPE = states [I_activity_t,I_action_t,internal_state_t]

%   access to the state components

       enabled_state (ac:I_action_t, s:I_state_t):bool =
 CASES ac OF
alpha_1: True,
alpha_2: True,
```

```
   beta: True,
    tau: True
  ENDCASES;

     graph (ac:I_action_t, s:I_state_t):bool =
          CASES ac OF
alpha_1: (s‘activity = e),
alpha_2: (s‘activity = e),
   beta: (s‘activity = f) or (s‘activity = g),
    tau: (s‘activity = f) or (s‘activity = g)
  ENDCASES

     enabled_general(ac:I_action_t,s:I_state_t):bool =
enabled_state(ac,s) & graph(ac,s)

%   transitions
     trans (ac:I_action_t, s1:I_state_t ): I_state_t =
          CASES ac OF
alpha_1: s1 WITH [activity:= f, basic:= s1‘basic WITH [ v := 2 ]],
alpha_2: s1 WITH [activity:= g, basic:= s1‘basic WITH [ v := 2 ]],
beta:  IF (s1‘activity=f) THEN (s1 WITH [activity:= i, basic:= s1‘basic WITH [ v:= 4 ]])
 ELSE s1 WITH [activity:= k, basic:= s1‘basic WITH [ v:= 4 ]]
   ENDIF,
tau:   IF (s1‘activity=f) THEN s1 WITH [activity:= h, basic:= s1‘basic WITH [ v:=3]]
ElSE  s1 WITH [activity:= j, basic:= s1‘basic WITH [ v:=3]]
  ENDIF
   ENDCASES

% define enabled S_action_t

enabled (ac:I_action_t, s:I_state_t):bool =
          enabled_general(ac,s);

% define initial state

start (s:I_state_t):bool =
   ( s =  (# activity:=e,
   basic:= (# v:=1 #)
#)
   )

 IMPORTING ttm_lib@machine[I_state_t,I_action_t,enabled,trans,start]


END selts2decls
```

B.2 Invariants and proofs

B.2.1 Invariant of $\mathbb{Q}_1$ and its proof

```
selts1_invariants   : THEORY

   BEGIN

   IMPORTING  selts1decls

  Inv_1(s:S_state_t): bool = (s= (# activity:=a, basic:=(# u:=1 #) #)) or
     (s= (# activity:=b, basic:=(# u:=2 #) #)) or
     (s= (# activity:=c, basic:=(# u:=3 #) #)) or
     (s= (# activity:=d, basic:=(# u:=4 #) #))

  lemma_1: LEMMA (FORALL (s:S_state_t): reachable(s) => Inv_1(s));

  END selts1_invariants
```

```
Proof scripts for file selts1invariants.pvs:


selts1_invariants.lemma_1: proved - complete [0](1.96 s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_1")
 (EXPAND "inductthm")
 (SPLIT -1)
 (("1" (PROPAX))
  ("2" (EXPAND "base") (EXPAND "start") (EXPAND "Inv_1") (PROPAX))
  ("3" (EXPAND "inductstep") (HIDE 2) (SKOSIMP) (EXPAND "trans") (GRIND))))
```

B.2.2 Invariant of $\mathbb{Q}_2$ and its proof

```
selts2_invariants   : THEORY

   BEGIN

  IMPORTING  selts2decls

   Inv_2(s:I_state_t): bool = (s= (# activity:=e, basic:=(# v:=1 #) #)) or
         (s= (# activity:=f, basic:=(# v:=2 #) #)) or
         (s= (# activity:=g, basic:=(# v:=2 #) #)) or
           (s= (# activity:=h, basic:=(# v:=3 #) #)) or
           (s= (# activity:=i, basic:=(# v:=4 #) #)) or
           (s= (# activity:=j, basic:=(# v:=3 #) #)) or
           (s= (# activity:=k, basic:=(# v:=4 #) #))

   lemma_2: LEMMA (FORALL (s:I_state_t): reachable(s) => Inv_2(s));

  END selts2_invariants

Proof scripts for file selts2invariants.pvs:


selts2_invariants.lemma_2: proved - complete [0](9.08 s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_2")
 (EXPAND "inductthm")
 (SPLIT -1)
 (("1" (PROPAX))
  ("2" (HIDE 2) (EXPAND "base") (EXPAND "start") (EXPAND "Inv_2") (PROPAX))
  ("3"
   (HIDE 2)
   (EXPAND "inductstep")
   (SKOSIMP)
   (EXPAND "trans")
   (EXPAND "Inv_2")
   (ASSERT)
   (GRIND))))
```

B.3 Strong Equivalence and proofs

```
seltsstrong: THEORY

BEGIN

 IMPORTING selts1_invariants
 IMPORTING selts2_invariants
```

```
ttm_lib: LIBRARY = "../ttm_lib"

action: DATATYPE
BEGIN
 alpha: alpha?
 beta: beta?
 tau: tau?
END action

     state: TYPE = int

ps1(s1:S_state_t): state =  s1‘basic‘u

ps2(s2:I_state_t): state =  s2‘basic‘v

pa1(a1:S_action_t): action =
CASES a1 OF
alpha: alpha,
beta: beta,
tau: tau
ENDCASES

pa2(a2:I_action_t): action =
CASES a2 OF
alpha_1: alpha,
alpha_2: alpha,
beta:  beta,
tau: tau
ENDCASES

dd1(s1:S_state_t,a1:S_action_t,s10:S_state_t): bool =
enabled(a1,s1)&
s10=trans(a1,s1)

dd2(s2:I_state_t,a2:I_action_t,s20:I_state_t): bool =
enabled(a2,s2)&
s20=trans(a2,s2)

RR(s1:S_state_t,s2:I_state_t):bool =
(
(s1‘activity=a and s2‘activity=e)
 or
(s1‘activity=b and (s2‘activity=f  or s2‘activity=g))
 or
(s1‘activity=c and (s2‘activity=h  or s2‘activity=j))
 or
(s1‘activity=d and (s2‘activity=i  or s2‘activity=k))
)
& ps1(s1)=ps2(s2)

s1: VAR S_state_t;
s2: VAR I_state_t;

IMPORTING ttm_lib@strongbisim[S_state_t,I_state_t,state,S_action_t, I_action_t,action];

simulation1: LEMMA sim1_2(ps1,ps2,pa1,pa2,dd1,dd2)(RR);

simulation2: LEMMA sim2_1(ps1,ps2,pa1,pa2,dd1,dd2)(RR);

bisimul: LEMMA bisimulation(ps1,ps2,pa1,pa2,dd1,dd2)(RR);

strongequi: LEMMA strongequivalence(ps1,ps2,pa1,pa2,dd1,dd2,start,start)(s1,s2)(RR);

END seltsstrong

Proof scripts for file seltsstrong.pvs:
```

```
seltsstrong.simulation1: proved - complete [0](14.70 s)

(""
 (EXPAND "sim1_2")
 (EXPAND "sim")
 (SKOLEM-TYPEPRED)
 (EXPAND "dd1")
 (EXPAND "enabled")
 (GRIND)
 (("1" (INST 1 "alpha_1") (GRIND)) ("2" (INST 4 "beta") (GRIND))
  ("3" (INST 5 "tau") (GRIND)) ("4" (INST 4 "beta") (GRIND))
  ("5" (INST 5 "tau") (GRIND))))


seltsstrong.simulation2: proved - complete [0](14.02 s)

(""
 (EXPAND "sim2_1")
 (EXPAND "sim")
 (SKOLEM-TYPEPRED)
 (GRIND)
 (("1" (INST 1 "alpha") (GRIND)) ("2" (INST 1 "alpha") (GRIND))
  ("3" (INST 5 "beta") (GRIND)) ("4" (INST 6 "tau") (GRIND))
  ("5" (INST 5 "beta") (GRIND)) ("6" (INST 6 "tau") (GRIND))))


seltsstrong.bisimul: proved - complete [0](0.06 s)

(""
 (LEMMA "simulation1")
 (LEMMA "simulation2")
 (EXPAND "bisimulation")
 (BDDSIMP))


seltsstrong.strongequi: proved - complete [0](10.73 s)

(""
 (SKOSIMP)
 (EXPAND "strongequivalence")
 (LEMMA "bisimul")
 (EXPAND "start")
 (EXPAND "RR")
 (GRIND))
```

Appendix C

Strong Equivalence of TTMs

C.1 Formalization of TTMs in Figure 5.7

C.1.1 Formalization of $\mathbb{Q}_1$

```
ttm1_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

% ----Beginning of the frist TTM declaration ----------
% We add a prefix S for specification to the variable names to
% distingush from the prefix I for Implementation.
% type and function declarations:

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

activity: TYPE = {a,b}

IMPORTING action1

% state variables

internal_state: TYPE = [# u: int #]

% Auxiliary functions

lower_bound (ac:S_nt_action):time =
CASES ac OF
alpha: one
ENDCASES

upper_bound(ac:S_nt_action):time =
CASES ac OF
alpha: one
ENDCASES

%   preconditions
```

```
    enabled_state (ac:S_nt_action, si:internal_state):bool =
  CASES ac OF
alpha: True
  ENDCASES;

     graph (ac:S_nt_action, sa:activity):bool =
          CASES ac OF
alpha: (sa = a)
  ENDCASES

IMPORTING ttm[activity,internal_state,S_nt_action,lower_bound,upper_bound,enabled_state,graph]

  S_state: TYPE = states[activity, S_nt_action, internal_state, time]

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

% transitions

     trans (ac:S_action, s:S_state):S_state =
          CASES ac OF
tick: s WITH [action_time:= update_clocks(s)],
alpha: s WITH [ activity:=b, basic:= s`basic WITH[u := 1],
action_time:=reset_clocks(ac, s WITH
[ activity:=b, basic:= s`basic WITH[u := 1]])]
  ENDCASES

% define initial state

start (s:S_state):bool =
   ( s = (# activity:=a,
     basic:= (# u:= 0 #),
     action_time:=(LAMBDA (a:S_action): zero)
#)
   )

IMPORTING ttm_lib@machine[S_state,S_action,enabled,trans,start]

END ttm1_decls
```

C.1.2 Formalization of $\mathbb{Q}_2$

```
ttm2_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

IMPORTING time_thy

% type and function declarations:

activity: TYPE = {c,d};

IMPORTING action2

% state variable

internal_state: TYPE = [#  v: int #]

% Auxiliary functions

lower_bound(ac:I_nt_action):time =
CASES ac OF
beta: one
```

```
ENDCASES

upper_bound(ac:I_nt_action):time =
CASES ac OF
beta: one
ENDCASES

% access to the state components

    enabled_state (ac:I_nt_action, ii:internal_state):bool =
 CASES ac OF
beta: True
 ENDCASES;

    graph (ac:I_nt_action, ia:activity):bool =
        CASES ac OF
beta: (ia = c)
 ENDCASES

IMPORTING ttm[activity,internal_state,I_nt_action,
lower_bound,upper_bound,enabled_state,graph]

    I_state: TYPE = states[activity, I_nt_action, internal_state,time]

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

%   transitions
    trans (ac:I_action, s:I_state):I_state =
        CASES ac OF
tick: s WITH [action_time:= update_clocks(s) ],
beta: s WITH [activity:=d, basic:= s`basic WITH [ v := 1 ],
action_time:= reset_clocks(ac,s WITH [activity:=d, basic:=s`basic WITH [ v:= 1 ]])]
  ENDCASES

% define initial state

start (s:I_state):bool =
  ( s = (# activity:=c,
   basic:= (# v:=0 #),
   action_time:=(LAMBDA (a:I_action): zero)
#)
  )

  IMPORTING ttm_lib@machine[I_state,I_action,enabled,trans,start]


END ttm2_decls
```

C.2 Invariants and proofs

C.2.1 Invariant of $\mathbb{Q}_1$ and its proof

```
ttm1_invariants   : THEORY

   BEGIN

   IMPORTING  ttm1_decls
        IMPORTING  ttm_unique_aux
        IMPORTING  ttm_rewrite_aux_1
        IMPORTING  ttm_rewrite_aux_2
```

```
  Inv_1(s:S_state) : bool = (s‘activity=a and s‘basic‘u=0 and s‘action_time(alpha)=zero) or
      (s‘activity=a and s‘basic‘u=0 and s‘action_time(alpha)=one) or
      (s‘activity=b and s‘basic‘u=1 and s‘action_time(alpha)=zero)

  lemma_1: LEMMA (FORALL (s:S_state): reachable(s) => Inv_1(s));

  END ttm1_invariants
```

Proof scripts for file ttm1invariant.pvs:

```
ttm1_invariants.Inv_1_TCC1: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC2: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.lemma_1: proved - complete [0](n/a s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_1")
 (EXPAND "inductthm")
 (SPLIT -1)
 (("1" (PROPAX))
  ("2"
   (HIDE 2)
   (EXPAND "base")
   (EXPAND "start")
   (EXPAND "Inv_1")
   (ASSERT)
   (GRIND))
  ("3"
   (HIDE 2)
   (EXPAND "inductstep")
   (SKOSIMP)
   (EXPAND "trans")
   (EXPAND "Inv_1")
   (GRIND))))
```

C.2.2 Invariant of $\mathbb{Q}_2$ and its proof

```
ttm2_invariants   : THEORY

   BEGIN

   IMPORTING  ttm2_decls
         IMPORTING  ttm_unique_aux
         IMPORTING  ttm_rewrite_aux_1
         IMPORTING  ttm_rewrite_aux_2

    Inv_2(s:I_state) : bool = (s‘activity=c and s‘basic‘v=0 and s‘action_time(beta)=zero) or
(s‘activity=c and s‘basic‘v=0 and s‘action_time(beta)=one) or
        (s‘activity=d and s‘basic‘v=1 and s‘action_time(beta)=zero)

    lemma2: LEMMA (FORALL (s:I_state): reachable(s) => Inv_2(s));


  END ttm2_invariants
```

Proof scripts for file ttm2invariant.pvs:

```
ttm2_invariants.Inv_2_TCC1: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC2: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.lemma2: proved - complete [0](n/a s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_2")
 (EXPAND "inductthm")
 (SPLIT -1)
 (("1" (PROPAX))
  ("2" (HIDE 2) (EXPAND "base") (EXPAND "Inv_2") (EXPAND "start") (GRIND))
  ("3"
   (HIDE 2)
   (EXPAND "inductstep")
   (SKOSIMP)
   (EXPAND "Inv_2")
   (EXPAND "trans")
   (EXPAND "reset_clocks")
   (GRIND))))
```

C.3 Strong Equivalence and proofs

```
ttm_bisimul: THEORY

BEGIN

IMPORTING ttm1_invariants
IMPORTING ttm2_invariants

% Beginning of definition of intermediate states and actions etc.

action: DATATYPE
BEGIN
 tick: tick?
 gamma: gamma?
END action

     state: TYPE = int

ps1(s1:S_state): state = s1`basic`u

ps2(s2:I_state): state = s2`basic`v

pa1(a1:S_action): action =
CASES a1 OF
tick: tick,
alpha: gamma
ENDCASES

pa2(a2:I_action): action =
CASES a2 OF
tick: tick,
beta: gamma
ENDCASES

dd1(s1:S_state,a1:S_action,s10:S_state): bool =
```

```
enabled(a1,s1)&(s10 = trans(a1,s1))

dd2(s2:I_state,a2:I_action,s20:I_state): bool =
enabled(a2,s2)&(s20 = trans(a2,s2))

RR(s1:S_state,s2:I_state):bool =
(s1`activity=a and s2`activity=c
 or
s1`activity=b and s2`activity=d)
& ps1(s1)=ps2(s2)
& s1`action_time(alpha) = s2`action_time(beta)
& Inv_1(s1) & Inv_2(s2)

s1: VAR S_state;
s2: VAR I_state;

IMPORTING ttm_lib@strongbisim[S_state,I_state,state,S_action,
I_action,action]

simulation1: LEMMA sim1_2(ps1,ps2,pa1,pa2,dd1,dd2)(RR);

simulation2: LEMMA sim2_1(ps1,ps2,pa1,pa2,dd1,dd2)(RR);

bisimul: THEOREM bisimulation(ps1,ps2,pa1,pa2,dd1,dd2)(RR);

strongequi: LEMMA strongequivalence(ps1,ps2,pa1,pa2,dd1,dd2,start,start)(s1,s2)(RR);

END ttm_bisimul

Proof scripts for file ttmbisimul.pvs:


ttm_bisimul.RR_TCC1: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm_bisimul.RR_TCC2: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm_bisimul.simulation1: proved - complete [0](n/a s)

(""
 (EXPAND "sim1_2")
 (EXPAND "sim")
 (SKOSIMP)
 (SKOSIMP)
 (EXPAND "RR")
 (FLATTEN)
 (EXPAND "Inv_1")
 (EXPAND "Inv_2")
 (SPLIT -6)
 (("1"
   (INST 1 "s2!1 WITH [action_time := update_clocks(s2!1) ]" "tick")
   (EXPAND "dd1")
   (GRIND)
   (("1" (TIME_ETC_SIMP) (GRIND)) ("2" (TIME_ETC_SIMP) (GRIND))
    ("3" (TIME_ETC_SIMP) (GRIND)) ("4" (TIME_ETC_SIMP) (GRIND))
    ("5" (TIME_ETC_SIMP) (GRIND)) ("6" (TIME_ETC_SIMP) (GRIND))
    ("7" (TIME_ETC_SIMP) (GRIND)) ("8" (TIME_ETC_SIMP) (GRIND))
    ("9" (TIME_ETC_SIMP) (GRIND)) ("10" (TIME_ETC_SIMP) (GRIND))
    ("11" (TIME_ETC_SIMP) (GRIND)) ("12" (TIME_ETC_SIMP))
    ("13" (TIME_ETC_SIMP) (GRIND)) ("14" (TIME_ETC_SIMP) (GRIND))
    ("15" (TIME_ETC_SIMP) (GRIND)) ("16" (TIME_ETC_SIMP) (GRIND))))
  ("2"
   (INST 1
```

```
    "s2!1 WITH [activity:=d, basic:= s2!1`basic WITH [ v := 1 ],
          action_time:= reset_clocks(beta,s2!1 WITH [activity:=d, basic:=s2!1`basic WITH [ v:= 1 ]])]"
    "beta")
   (GRIND)
   (("1" (TIME_ETC_SIMP)) ("2" (TIME_ETC_SIMP) (GRIND))
    ("3" (TIME_ETC_SIMP) (GRIND)) ("4" (TIME_ETC_SIMP) (GRIND))
    ("5" (TIME_ETC_SIMP) (GRIND)) ("6" (TIME_ETC_SIMP) (GRIND))
    ("7" (TIME_ETC_SIMP) (GRIND)) ("8" (TIME_ETC_SIMP) (GRIND))
    ("9" (TIME_ETC_SIMP) (GRIND)) ("10" (TIME_ETC_SIMP) (GRIND))
    ("11" (TIME_ETC_SIMP) (GRIND)) ("12" (TIME_ETC_SIMP) (GRIND))))
  ("3"
   (INST 1 "s2!1 WITH [action_time:= update_clocks(s2!1)]" "tick")
   (GRIND))))


ttm_bisimul.simulation2: proved - complete [0](n/a s)

(""
 (EXPAND "sim2_1")
 (EXPAND "sim")
 (SKOSIMP)
 (SKOSIMP)
 (EXPAND "converse")
 (EXPAND "RR")
 (FLATTEN)
 (EXPAND "Inv_2")
 (EXPAND "Inv_1")
 (SPLIT -6)
 (("1"
   (INST 1 "s2!1 WITH [action_time:=update_clocks(s2!1)]" "tick")
   (GRIND)
   (("1" (TIME_ETC_SIMP) (ASSERT)) ("2" (TIME_ETC_SIMP) (ASSERT))
    ("3" (TIME_ETC_SIMP) (ASSERT)) ("4" (TIME_ETC_SIMP) (ASSERT))
    ("5" (TIME_ETC_SIMP) (ASSERT)) ("6" (TIME_ETC_SIMP) (ASSERT))
    ("7" (TIME_ETC_SIMP) (ASSERT)) ("8" (TIME_ETC_SIMP) (ASSERT))
    ("9" (TIME_ETC_SIMP) (ASSERT)) ("10" (TIME_ETC_SIMP) (ASSERT))
    ("11" (TIME_ETC_SIMP) (ASSERT)) ("12" (TIME_ETC_SIMP))
    ("13" (TIME_ETC_SIMP) (ASSERT)) ("14" (TIME_ETC_SIMP) (ASSERT))
    ("15" (TIME_ETC_SIMP) (ASSERT)) ("16" (TIME_ETC_SIMP) (ASSERT))))
  ("2"
   (INST 1
    "s2!1 WITH [ activity:=b, basic:= s2!1`basic WITH[u := 1],
              action_time:=reset_clocks(alpha, s2!1 WITH
              [ activity:=b, basic:= s2!1`basic WITH[u := 1]])]"
    "alpha")
   (ASSERT)
   (GRIND)
   (("1" (TIME_ETC_SIMP) (ASSERT)) ("2" (TIME_ETC_SIMP))
    ("3" (TIME_ETC_SIMP) (ASSERT)) ("4" (TIME_ETC_SIMP) (ASSERT))
    ("5" (TIME_ETC_SIMP) (ASSERT)) ("6" (TIME_ETC_SIMP) (ASSERT))
    ("7" (TIME_ETC_SIMP) (ASSERT))))
  ("3"
   (INST 1 "s2!1 WITH [action_time:= update_clocks(s2!1)]" "tick")
   (ASSERT)
   (GRIND))))


ttm_bisimul.bisimul: proved - complete [0](n/a s)

(""
 (EXPAND "bisimulation")
 (LEMMA "simulation1")
 (LEMMA "simulation2")
 (ASSERT))


ttm_bisimul.strongequi: proved - complete [0](n/a s)
```

```
(""
 (SKOSIMP)
 (EXPAND "strongequivalence")
 (LEMMA "bisimul")
 (SPLIT 1)
 (("1" (PROPAX)) ("2" (HIDE -1) (EXPAND "start") (EXPAND "RR") (GRIND))))
```

Appendix D

Weak Equivalence of SELTSs

D.1 Formalization of SELTS in Figure 5.8

D.1.1 Formalization of $\mathbb{Q}_1$

```
ttm1_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"
% Weak equivalence on only the events without considering state output.
% We add a prefix S for specification to the variable names to
% distingush from the prefix I for Implementation.
% type and function declarations:

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

S_activity_t: TYPE = {a}

% S_action_t

S_action_t: DATATYPE
BEGIN
 alpha: alpha?
END S_action_t

% state variables

internal_state_t: TYPE = [# u: int #]

% preconditions

    enabled_state (ac:S_action_t, si:internal_state_t): bool =
CASES ac OF
alpha: True
ENDCASES;

    graph (ac:S_action_t, sa:S_activity_t):bool =
         CASES ac OF
alpha: (sa = a)
```

```
 ENDCASES

IMPORTING states[S_activity_t, S_action_t, internal_state_t]

    S_state_t: TYPE = states[S_activity_t, S_action_t, internal_state_t]

% enabled condition

    enabled (ac: S_action_t, s: S_state_t): bool =
enabled_state(ac, s`basic) & graph(ac, s`activity)

    trans (ac:S_action_t, s:S_state_t):S_state_t =
        CASES ac OF
alpha: s
 ENDCASES

% define initial state

    start (s:S_state_t):bool =
  ( s = (# activity:=a,
   basic:= (# u:=1 #)
#)
  )

IMPORTING ttm_lib@machine[S_state_t,S_action_t,enabled,trans,start]

END ttm1_decls
```

D.1.2 Formalization of $\mathbb{Q}_2$

```
ttm2_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

% type and function declarations:

 I_activity_t: TYPE = {b,c};

% I_action_t

I_action_t: DATATYPE
BEGIN
 alpha: alpha?
  beta: beta?
END I_action_t

% state variable

internal_state_t: TYPE = [# v: int #]

% preconditions

enabled_state (ac:I_action_t, ii:internal_state_t): bool =
 CASES ac OF
alpha: True,
 beta: True
 ENDCASES;

     graph (ac:I_action_t, ia:I_activity_t):bool =
        CASES ac OF
alpha: (ia = b),
```

```
 beta: (ia = c)
 ENDCASES;

 IMPORTING states[I_activity_t, I_action_t, internal_state_t]

    I_state_t: TYPE = states[I_activity_t, I_action_t, internal_state_t]

% enabled condition

    enabled (ac: I_action_t, s: I_state_t): bool =
enabled_state(ac, s`basic) & graph(ac, s`activity)

    trans (ac:I_action_t, s:I_state_t):I_state_t =
        CASES ac OF
alpha: s,
 beta: s
  ENDCASES

% define initial state

start (s:I_state_t):bool =
  ( s = (# activity:=b,
   basic:= (# v:=1 #)
#)
  )

  IMPORTING ttm_lib@machine[I_state_t,I_action_t,enabled,trans,start]


END ttm2_decls
```

D.2 Invariants and their proofs

D.2.1 Invariant of $\mathbb{Q}_1$ and its proof

```
ttm1_invariants   : THEORY

   BEGIN

   IMPORTING  ttm1_decls

   Inv_1(s:S_state_t): bool = (s`activity=a and s`basic= (# u:=1#))

   lemma_1: LEMMA (FORALL (s:S_state_t): reachable(s) => Inv_1(s));

  END ttm1_invariants

Proof scripts for file ttm1_invariants.pvs:


ttm1_invariants.lemma_1: proved - complete

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_1")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (GRIND))
  ("3"
   (EXPAND "inductstep")
   (SKOSIMP)
   (GRIND)
   (("1" (EXPAND "update_clocks") (GRIND) (GRIND) (POSTPONE))
    ("2" (EXPAND "update_clocks") (GRIND) (POSTPONE))
    ("3" (EXPAND "reset_clocks") (GRIND) (POSTPONE))))))
```

D.2.2 Invariant of $\mathbb{Q}_2$ and its proof

```
ttm2_invariants   : THEORY

   BEGIN

   IMPORTING  ttm2_decls

    Inv_2(s:I_state_t) : bool = (s`activity=b and s`basic= (# v:= 1 #))

    lemma2: LEMMA (FORALL (s:I_state_t): reachable(s) => Inv_2(s));


END ttm2_invariants

Proof scripts for file ttm2_invariants.pvs:


ttm2_invariants.lemma2: proved - complete

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_2")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (GRIND))
  ("3"
   (EXPAND "inductstep")
   (SKOSIMP)
   (EXPAND "trans")
   (EXPAND "Inv_2")
   (GRIND))))
```

D.3 Weak Equivalence and proofs

```
ttm_bisimul: THEORY

BEGIN

IMPORTING ttm1_invariants
IMPORTING ttm2_invariants

% Beginning of definition of intermediate states and actions etc.

action: DATATYPE
BEGIN
 alpha: alpha?
  tau: tau?
END action

state: TYPE = int

ps1(s1:S_state_t): state = s1`basic`u

ps2(s2:I_state_t): state = s2`basic`v

pa1(a1:S_action_t): action =
CASES a1 OF
alpha: alpha
ENDCASES

pa2(a2:I_action_t): action =
CASES a2 OF
alpha: alpha,
beta: tau
ENDCASES
```

```
Is_tau?(a:action): bool = (a=tau)

dd1(s1:S_state_t,a1:S_action_t,s10:S_state_t): bool =
enabled(a1,s1)&(s10 = trans(a1,s1))

dd2(s2:I_state_t,a2:I_action_t,s20:I_state_t): bool =
enabled(a2,s2)&(s20 = trans(a2,s2))

r1(s1,s10:S_state_t):bool = (EXISTS (a1: S_action_t): dd1(s1,a1,s10) & Is_tau?(pa1(a1)))

r2(s2,s20:I_state_t):bool = (EXISTS (a2: I_action_t): dd2(s2,a2,s20) & Is_tau?(pa2(a2)))

RR(s1:S_state_t,s2:I_state_t):bool =
(s1`activity=a and s2`activity=b)
& (Inv_1(s1) & Inv_2(s2))

IMPORTING ttm_lib@sebisim[S_state_t,I_state_t,state,S_action_t,
I_action_t,action]

simulation1: LEMMA sim1_2(ps1,ps2,pa1,pa2,dd1,dd2,r1,r2,Is_tau?)(RR);

simulation2: LEMMA sim2_1(ps1,ps2,pa1,pa2,dd1,dd2,r1,r2,Is_tau?)(RR);

bisimul: THEOREM bisimulation(ps1,ps2,pa1,pa2,dd1,dd2,r1,r2,Is_tau?)(RR);

END ttm_bisimul

Proof scripts for file ttm_bisimul.pvs:


ttm_bisimul.simulation1: proved - complete

(""
 (EXPAND "sim1_2")
 (EXPAND "sim")
 (SKOSIMP)
 (SKOSIMP)
 (EXPAND "RR")
 (FLATTEN)
 (EXPAND "Inv_1")
 (INST 1 "(# activity:=b, basic:=(# v:=1 #) #)")
 (FLATTEN)
 (INST 2 "(# activity:=b, basic:=(# v:=1 #) #)" "alpha"
  "(# activity:=b, basic:=(# v:=1 #) #)")
 (GRIND)
 (("1" (EXPAND "tc") (GRIND) (APPLY-EXTENSIONALITY))
  ("2" (EXPAND "tc") (PROPAX))))


ttm_bisimul.simulation2: proved - complete

(""
 (EXPAND "sim2_1")
 (EXPAND "sim")
 (EXPAND "dd2")
 (EXPAND "enabled")
 (EXPAND "RR")
 (EXPAND "graph")
 (EXPAND "converse")
 (SKOSIMP)
 (EXPAND "Inv_2")
 (SKOSIMP)
 (INST 1 "(# activity:=a, basic:= (# u:=1 #) #)")
 (FLATTEN)
 (INST 2 "(# activity:=a, basic:= (# u:=1 #) #)" "alpha"
  "(# activity:=a, basic:= (# u:=1 #) #)")
 (GRIND)
```

```
 (("1" (EXPAND "tc") (FLATTEN) (APPLY-EXTENSIONALITY))
  ("2" (EXPAND "tc") (PROPAX))))


ttm_bisimul.bisimul: proved - complete

(""
 (EXPAND "bisimulation")
 (LEMMA "simulation1")
 (LEMMA "simulation2")
 (ASSERT))
```

Appendix E

Weak Equivalence of TTMs

E.1 Formalization of TTMs in Figure 5.9

E.1.1 Formalization of $\mathbb{Q}_1$

```
ttm1_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"
% Weak equivalence on only the events without considering state output.
% We add a prefix S for specification to the variable names to
% distingush from the prefix I for Implementation.
% type and function declarations:

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

S_activity_t: TYPE = {a}

% S_action_t

S_action_t: DATATYPE
BEGIN
  tick: tick?
 alpha: alpha?
END S_action_t

% A predicate to tell whether the action is tick.

  Is_tick?(a:S_action_t): bool = (a = tick)

% state variables

internal_state_t: TYPE = [# u: int #]

% Auxiliary functions

lower_bound (ac:S_action_t): time =
  CASES ac OF
tick: zero,
```

```
       alpha: one
ENDCASES

upper_bound (ac:S_action_t): time =
CASES ac OF
tick: infinity,
       alpha: one
ENDCASES

% preconditions

    enabled_state (ac:S_action_t, si:internal_state_t): bool =
CASES ac OF
 tick: True,
alpha: True
ENDCASES;

    graph (ac:S_action_t, sa:S_activity_t):bool =
        CASES ac OF
 tick: True,
alpha: (sa = a)
 ENDCASES

IMPORTING ttm[S_activity_t,internal_state_t,S_action_t,Is_tick?,
lower_bound,upper_bound,enabled_state,graph]


IMPORTING states[S_activity_t, S_action_t, internal_state_t,time]

    S_state_t: TYPE = states[S_activity_t, S_action_t, internal_state_t,time]

    trans (ac:S_action_t, s:S_state_t):S_state_t =
        CASES ac OF
 tick: s WITH [ action_time:= update_clocks(s)],
alpha: s WITH [ action_time:= reset_clocks(ac,s)]
 ENDCASES

% define initial state

    start (s:S_state_t):bool =
  ( s = (# activity:=a,
   basic:= (# u:=1 #),
   action_time:=(LAMBDA (a:S_action_t): zero)
#)
  )

IMPORTING ttm_lib@machine[S_state_t,S_action_t,enabled,trans,start]

END ttm1_decls
```

E.1.2 Formalization of $\mathbb{Q}_2$

```
ttm2_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

% type and function declarations:

 I_activity_t: TYPE = {b,c};

% I_action_t
```

```
I_action_t: DATATYPE
BEGIN
  tick: tick?
 alpha: alpha?
  beta: beta?
END I_action_t

% A predicate to tell whether the action is tick.

Is_tick?(a: I_action_t): bool = (a = tick)

% state variable

internal_state_t: TYPE = [# v: int #]

% Auxiliary functions

lower_bound (ac: I_action_t): time =
CASES ac OF
tick: zero,
       alpha: one,
beta: one
ENDCASES

upper_bound(ac: I_action_t): time =
CASES ac OF
tick: infinity,
       alpha: one,
beta: one
ENDCASES

% preconditions

enabled_state (ac:I_action_t, ii:internal_state_t): bool =
 CASES ac OF
 tick: True,
alpha: True,
 beta: True
 ENDCASES;

     graph (ac:I_action_t, ia:I_activity_t):bool =
         CASES ac OF
 tick: True,
alpha: (ia = b),
 beta: (ia = c)
 ENDCASES;

 IMPORTING ttm[I_activity_t,internal_state_t,I_action_t,Is_tick?,
 lower_bound,upper_bound,enabled_state,graph]

 IMPORTING states[I_activity_t, I_action_t, internal_state_t, time]

    I_state_t: TYPE = states[I_activity_t, I_action_t, internal_state_t, time]

    trans (ac:I_action_t, s:I_state_t):I_state_t =
         CASES ac OF
 tick: s WITH [ action_time:= update_clocks(s)],
alpha: s WITH [ action_time:= reset_clocks(ac,s)],
 beta: s WITH [ action_time:= reset_clocks(ac,s)]
  ENDCASES

% define initial state

start (s:I_state_t):bool =
  ( s = (# activity:=b,
   basic:= (# v:=1 #),
   action_time:=(LAMBDA (a:I_action_t): zero)
```

```
#)
  )

  IMPORTING ttm_lib@machine[I_state_t,I_action_t,enabled,trans,start]


END ttm2_decls
```

E.2 Invariants and proofs

E.2.1 Invariant of $\mathbb{Q}_1$ and its proof

```
ttm1_invariants   : THEORY

   BEGIN

   IMPORTING  ttm1_decls

   Inv_1(s:S_state_t): bool = (s`activity=a and s`basic= (# u:=1#) and s`action_time(alpha)= zero) or
      (s`activity=a and s`basic= (# u:=1#) and s`action_time(alpha)= one)

   lemma_1: LEMMA (FORALL (s:S_state_t): reachable(s) => Inv_1(s));

  END ttm1_invariants

Proof scripts for file ttm1_invariants.pvs:


ttm1_invariants.lemma_1: proved - complete

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_1")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX))
  ("2" (EXPAND "base") (EXPAND "start") (EXPAND "Inv_1") (PROPAX))
  ("3"
   (EXPAND "inductstep")
   (HIDE 2)
   (SKOSIMP)
   (EXPAND "trans")
   (EXPAND "Inv_1")
   (GRIND))))
```

E.2.2 Invariant of $\mathbb{Q}_2$ and its proof

```
ttm2_invariants   : THEORY

   BEGIN

  IMPORTING  ttm2_decls

Inv_2(s:I_state_t) : bool  = (s`activity=b and s`basic= (# v:=1#) and s`action_time(alpha)= zero) or
             (s`activity=b and s`basic= (# v:=1#) and s`action_time(alpha)= one )

lemma2: LEMMA (FORALL (s:I_state_t): reachable(s) => Inv_2(s));


END ttm2_invariants

Proof scripts for file ttm2_invariants.pvs:
```

```
ttm2_invariants.lemma2: proved - complete

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_2")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (GRIND))
  ("3"
   (EXPAND "inductstep")
   (SKOSIMP)
   (EXPAND "trans")
   (EXPAND "Inv_2")
   (GRIND))))
```

E.3 Weak Equivalence and proofs

```
ttm_bisimul: THEORY

BEGIN

IMPORTING ttm1_invariants
IMPORTING ttm2_invariants

% Beginning of definition of intermediate states and actions etc.

action: DATATYPE
BEGIN
 alpha: alpha?
  tau: tau?
END action

state: TYPE = int

ps1(s1:S_state_t): state = s1`basic`u

ps2(s2:I_state_t): state = s2`basic`v

pa1(a1:S_action_t): action =
CASES a1 OF
 tick: alpha,
alpha: alpha
ENDCASES

pa2(a2:I_action_t): action =
CASES a2 OF
 tick: alpha,
alpha: alpha,
beta: tau
ENDCASES

Is_tau?(a:action): bool = (a=tau)

dd1(s1:S_state_t,a1:S_action_t,s10:S_state_t): bool =
enabled(a1,s1)&(s10 = trans(a1,s1))

dd2(s2:I_state_t,a2:I_action_t,s20:I_state_t): bool =
enabled(a2,s2)&(s20 = trans(a2,s2))

r1(s1,s10:S_state_t):bool = (EXISTS (a1: S_action_t): dd1(s1,a1,s10) & Is_tau?(pa1(a1)))

r2(s2,s20:I_state_t):bool = (EXISTS (a2: I_action_t): dd2(s2,a2,s20) & Is_tau?(pa2(a2)))

RR(s1:S_state_t,s2:I_state_t):bool =
```

```
(s1`activity=a and s2`activity=b)
& (Inv_1(s1) & Inv_2(s2))
& s1`action_time(alpha)=s2`action_time(alpha)
  & s1`action_time(tick)=s2`action_time(tick)

IMPORTING ttm_lib@sebisim[S_state_t,I_state_t,state,S_action_t,
I_action_t,action]

simulation1: LEMMA sim1_2(ps1,ps2,pa1,pa2,dd1,dd2,r1,r2,Is_tau?)(RR);

simulation2: LEMMA sim2_1(ps1,ps2,pa1,pa2,dd1,dd2,r1,r2,Is_tau?)(RR);

bisimul: THEOREM bisimulation(ps1,ps2,pa1,pa2,dd1,dd2,r1,r2,Is_tau?)(RR);

END ttm_bisimul

Proof scripts for file ttm_bisimul.pvs:


ttm_bisimul.simulation1: proved - complete

(""
 (EXPAND "sim1_2")
 (EXPAND "sim")
 (SKOSIMP)
 (SKOSIMP)
 (EXPAND "RR")
 (EXPAND "Inv_1")
 (EXPAND "Inv_2")
 (FLATTEN)
 (SPLIT -3)
 (("1"
   (INST 1 "s2!1 WITH [ action_time:= update_clocks(s2!1)]")
   (FLATTEN)
   (INST 2 "s2!1" "tick" "s2!1 WITH [ action_time:= update_clocks(s2!1)]")
   (GRIND)
   (("1" (EXPAND "tc") (TIME_ETC_SIMP) (GRIND))
    ("2" (TIME_ETC_SIMP) (GRIND) (EXPAND "tc") (PROPAX))
    ("3" (TIME_ETC_SIMP) (GRIND)) ("4" (TIME_ETC_SIMP) (GRIND))
    ("5" (TIME_ETC_SIMP) (GRIND))
    ("6" (TIME_ETC_SIMP) (GRIND) (EXPAND "tc") (PROPAX))
    ("7" (TIME_ETC_SIMP) (GRIND)) ("8" (TIME_ETC_SIMP) (GRIND))
    ("9" (EXPAND "tc") (PROPAX)) ("10" (EXPAND "tc") (PROPAX))
    ("11" (TIME_ETC_SIMP) (GRIND)) ("12" (TIME_ETC_SIMP) (GRIND))
    ("13" (EXPAND "tc") (PROPAX)) ("14" (EXPAND "tc") (PROPAX))
    ("15" (EXPAND "tc") (EXPAND "tc") (TIME_ETC_SIMP) (GRIND))
    ("16" (TIME_ETC_SIMP) (GRIND)) ("17" (EXPAND "tc") (PROPAX))
    ("18" (EXPAND "tc") (TIME_ETC_SIMP) (GRIND))
    ("19" (TIME_ETC_SIMP) (GRIND)) ("20" (TIME_ETC_SIMP) (GRIND))
    ("21" (TIME_ETC_SIMP) (GRIND)) ("22" (TIME_ETC_SIMP) (GRIND))
    ("23" (TIME_ETC_SIMP) (GRIND)) ("24" (TIME_ETC_SIMP) (GRIND))))
  ("2"
   (INST 1 "s2!1 WITH [action_time:= reset_clocks(alpha,s2!1)]")
   (FLATTEN)
   (INST 2 "s2!1" "alpha"
    "s2!1 WITH [action_time:= reset_clocks(alpha,s2!1)]")
   (GRIND)
   (("1" (TIME_ETC_SIMP) (GRIND)) ("2" (TIME_ETC_SIMP) (GRIND))
    ("3" (TIME_ETC_SIMP) (GRIND)) ("4" (TIME_ETC_SIMP) (GRIND))
    ("5" (TIME_ETC_SIMP) (GRIND)) ("6" (TIME_ETC_SIMP) (GRIND))
    ("7" (TIME_ETC_SIMP) (GRIND)) ("8" (TIME_ETC_SIMP) (GRIND))
    ("9" (TIME_ETC_SIMP) (GRIND)) ("10" (TIME_ETC_SIMP) (GRIND))
    ("11" (TIME_ETC_SIMP) (GRIND)) ("12" (TIME_ETC_SIMP) (GRIND))
    ("13"
     (TIME_ETC_SIMP)
     (GRIND)
     (TIME_ETC_SIMP)
     (GRIND)
```

```
     (EXPAND "tc")
     (PROPAX))
    ("14" (TIME_ETC_SIMP) (GRIND) (EXPAND "tc") (PROPAX))
    ("15" (EXPAND "tc") (PROPAX)) ("16" (EXPAND "tc") (PROPAX))))))


ttm_bisimul.simulation2: proved - complete

(""
 (EXPAND "sim2_1")
 (EXPAND "sim")
 (SKOSIMP)
 (SKOSIMP)
 (EXPAND "RR")
 (EXPAND "converse")
 (FLATTEN)
 (EXPAND "Inv_2")
 (EXPAND "Inv_1")
 (SPLIT -4)
 (("1"
   (INST 1 "s2!1 WITH [action_time:=update_clocks(s2!1)]")
   (FLATTEN)
   (INST 2 "s2!1" "tick" "s2!1 WITH [action_time:=update_clocks(s2!1)]")
   (EXPAND "sitc")
   (EXPAND "tc")
   (GRIND)
   (("1" (EXPAND "tc") (TIME_ETC_SIMP) (GRIND)) ("2" (TIME_ETC_SIMP) (GRIND))
    ("3" (TIME_ETC_SIMP) (GRIND)) ("4" (TIME_ETC_SIMP) (GRIND))
    ("5" (TIME_ETC_SIMP) (GRIND)) ("6" (TIME_ETC_SIMP) (GRIND))
    ("7" (TIME_ETC_SIMP) (GRIND)) ("8" (TIME_ETC_SIMP) (GRIND))
    ("9" (TIME_ETC_SIMP) (GRIND)) ("10" (TIME_ETC_SIMP) (GRIND))
    ("11" (TIME_ETC_SIMP) (GRIND)) ("12" (TIME_ETC_SIMP) (GRIND))
    ("13" (TIME_ETC_SIMP) (GRIND)) ("14" (TIME_ETC_SIMP) (GRIND))
    ("15" (TIME_ETC_SIMP) (GRIND))))
  ("2"
   (INST 1 "s2!1 WITH [action_time:= reset_clocks(alpha,s2!1)]")
   (FLATTEN)
   (INST 2 "s2!1" "alpha"
    "s2!1 WITH [action_time:= reset_clocks(alpha,s2!1)]")
   (GRIND)
   (("1" (TIME_ETC_SIMP) (GRIND)) ("2" (TIME_ETC_SIMP) (GRIND))
    ("3" (TIME_ETC_SIMP) (GRIND)) ("4" (TIME_ETC_SIMP) (GRIND))
    ("5" (TIME_ETC_SIMP) (GRIND) (EXPAND "tc") (PROPAX))
    ("6" (TIME_ETC_SIMP) (GRIND) (EXPAND "tc") (PROPAX))
    ("7" (TIME_ETC_SIMP) (GRIND) (EXPAND "tc") (PROPAX))
    ("8" (EXPAND "tc") (PROPAX))))))


ttm_bisimul.bisimul: proved - complete

(""
 (EXPAND "bisimulation")
 (LEMMA "simulation1")
 (LEMMA "simulation2")
 (ASSERT))
```

Appendix F

Weak Equivalence of an Industrial Real-time Controller

F.1 Formalization of TTMs

F.1.1 Formalization of specification in PVS

```
ttm1_decls: THEORY

BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

% ----Beginning of the frist TTM declaration ----------
% We add a prefix S for specification to the variable names to
% distingush from the prefix I for Implementation.
% type and function declarations:

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

activity: TYPE = {a,b,c,d,e}

IMPORTING action1

%% state variables

s_internal_state: TYPE = [# Relay: bool, Power:bool, Pressure:bool #]

% Auxiliary functions

lower_bound (ac:S_nt_action): time =
  CASES ac OF
        mu: one,
        omega29: twenty_nine,
alpha: one,
omega19: nineteen,
rho1: one,
rho2: one,
```

```
gamma: one,
wr: one,
ws: one,
pr: one,
ps: one
ENDCASES

upper_bound (ac:S_nt_action): time =
CASES ac OF
mu: one,
        omega29: twenty_nine,
alpha: one,
omega19: nineteen,
rho1: one,
rho2: one,
gamma: one,
wr: infinity,
ws: infinity,
pr: infinity,
ps: infinity
ENDCASES

% preconditions

    enabled_state (ac:S_nt_action, si:s_internal_state): bool =
CASES ac OF
  mu: si‘Pressure and si‘Power,
  omega29: True,
alpha: si‘Power,
omega19: True,
rho1: Not si‘Power,
rho2: Not si‘Power,
gamma: si‘Power,
wr: si‘Power,
ws: Not si‘Power,
pr: si‘Pressure,
ps: Not si‘Pressure
ENDCASES;

    graph (ac:S_nt_action, sa:activity):bool =
          CASES ac OF
mu:  sa=a,
  omega29: sa=b,
alpha: sa=c,
omega19: sa=d,
rho1: sa=c,
rho2: sa=e,
gamma: sa=e,
wr: True,
ws: True,
pr: True,
ps: True
  ENDCASES

IMPORTING ttm[activity,s_internal_state,S_nt_action,
lower_bound,upper_bound,enabled_state,graph]


IMPORTING states[activity, S_nt_action, s_internal_state,time]

    S_state: TYPE = states[activity, S_nt_action, s_internal_state,time]

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

    trans (ac:S_action, s:S_state):S_state =
```

```
         CASES ac OF
  tick: s WITH [ action_time:= update_clocks(s)],
mu:   s WITH [ activity:=b, action_time:= reset_clocks(ac, s WITH [activity:=b]) ],
  omega29: s WITH [activity:=c, action_time:= reset_clocks(ac, s WITH [activity:=c]) ],
alpha: s WITH [activity:=d, basic:= s‘basic WITH [Relay:=True ],
action_time:= reset_clocks(ac, s WITH [activity:=d, basic:= s‘basic WITH [Relay:=True]])],
omega19: s WITH [activity:=e, action_time:= reset_clocks(ac, s WITH [ activity:=e ]) ],
rho1: s WITH [activity:=a, action_time:= reset_clocks(ac, s WITH [ activity:=a ]) ],
rho2: s WITH [activity:=a, basic:= s‘basic WITH[Relay:=False],
action_time:= reset_clocks(ac, s WITH [activity:=a, basic:= s‘basic WITH[Relay:=False]])],
gamma: s WITH [activity:=a, action_time:= reset_clocks(ac, s WITH [ activity:=a ]) ],
wr: s WITH [basic:=s‘basic WITH [Power:=False], action_time:= reset_clocks(ac,s WITH [basic:=s‘basic WITH [Power:=False]])],
ws: s WITH [basic:=s‘basic WITH [Power:=True], action_time:= reset_clocks(ac,s WITH [basic:=s‘basic WITH [Power:=True]])],
pr: s WITH [basic:=s‘basic WITH [Pressure:=False], action_time:= reset_clocks(ac,s WITH [basic:=s‘basic WITH [Pressure:=False]])],
ps: s WITH [basic:=s‘basic WITH [Pressure:=True], action_time:= reset_clocks(ac,s WITH [basic:=s‘basic WITH [Pressure:=True]])]
  ENDCASES

% define initial state

    start (s:S_state):bool =
  ( s = (# activity:=a,
   basic:= (#  Relay:=False, Power:=False, Pressure:=False #),
   action_time:=(LAMBDA (a:S_nt_action): zero)
#)
  )

IMPORTING ttm_lib@machine[S_state,S_action,enabled,trans,start]

END ttm1_decls
```

F.1.2 Formalization of implementation in PVS

```
ttm2_decls: THEORY
BEGIN

ttm_lib: LIBRARY = "../ttm_lib"

% type and function declarations:

IMPORTING time_thy
IMPORTING ttm_lib@list_rewrites
IMPORTING ttm_lib@bool_rewrites

I_activity: TYPE = {a}

IMPORTING action2

% state variables

internal_state: TYPE = [# Relay: bool, Power:bool, Pressure:bool, c1:nat, c2:nat #]

% Auxiliary functions

lower_bound (ac:I_nt_action): time =
  CASES ac OF
       mu1: one,
alpha: one,
mu2: one,
rho1: one,
rho2: one,
gamma: one,
wr: one,
ws: one,
pr: one,
ps: one
ENDCASES
```

```
upper_bound (ac:I_nt_action): time =
CASES ac OF
mu1: one,
alpha: one,
mu2: one,
rho1: one,
rho2: one,
gamma: one,
wr: infinity,
ws: infinity,
pr: infinity,
ps: infinity
ENDCASES

% preconditions

    enabled_state (ac:I_nt_action, si:internal_state): bool =
CASES ac OF
  mu1: (si‘Pressure and si‘Power and si‘c1=0 and si‘c2=0) or (si‘c1>=1 and si‘c1<=29),
alpha: si‘Power and si‘c1>=30,
mu2: (si‘c1=0 and si‘c2>=1 and si‘c2<=19),
rho1: Not si‘Power and si‘c1>=30,
rho2: Not si‘Power and si‘c1=0 and si‘c2>=20,
gamma: si‘Power and si‘c1=0 and si‘c2>=20,
wr: si‘Power,
ws: Not si‘Power,
pr: si‘Pressure,
ps: Not si‘Pressure
ENDCASES;

    graph (ac:I_nt_action, sa:I_activity):bool =
          CASES ac OF
mu1: sa=a,
alpha: sa=a,
mu2: sa=a,
rho1: sa=a,
rho2: sa=a,
gamma: sa=a,
wr: True,
ws: True,
pr: True,
ps: True
  ENDCASES

IMPORTING ttm[I_activity,internal_state,I_nt_action,
lower_bound,upper_bound,enabled_state,graph]


IMPORTING states[I_activity, I_nt_action, internal_state,time]

    I_state: TYPE = states[I_activity, I_nt_action, internal_state,time]


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

    trans (ac:I_action, s:I_state):I_state =
          CASES ac OF
  tick: s WITH [ action_time:= update_clocks(s)],
mu1:   s WITH [ basic:= s‘basic WITH [c1:=s‘basic‘c1+1],
action_time:= reset_clocks(mu1, s WITH [basic:=s‘basic WITH [c1:=s‘basic‘c1+1]])],
alpha: s WITH [ basic:= s‘basic WITH [c1:=0,c2:=s‘basic‘c2+1,Relay:=True],
action_time:= reset_clocks(alpha, s WITH [ basic:= s‘basic WITH [c1:=0,c2:=s‘basic‘c2+1,Relay:=True]])],
mu2: s WITH [ basic:=s‘basic WITH [c2:=s‘basic‘c2+1,Relay:=True],
action_time:= reset_clocks(mu2, s WITH [ basic:=s‘basic WITH [c2:=s‘basic‘c2+1,Relay:=True]]) ],
rho1: s WITH [ basic:=s‘basic WITH [ c1:=0 ],
```

```
action_time:= reset_clocks(rho1, s WITH [ basic:= s‘basic WITH [c1:=0]]) ],
rho2: s WITH [ basic:= s‘basic WITH[c2:=0, Relay:=False],
action_time:= reset_clocks(rho2, s WITH [ basic:= s‘basic WITH[c2:=0,Relay:=False]]) ],
gamma: s WITH [ basic:= s‘basic WITH [ c2:=0 ],
action_time:= reset_clocks(gamma, s WITH [ basic:= s‘basic WITH [c2:=0]]) ],
wr: s WITH [basic:=s‘basic WITH [Power:=False], action_time:= reset_clocks(wr,s WITH [basic:=s‘basic WITH [Power:=False]])],
ws: s WITH [basic:=s‘basic WITH [Power:=True], action_time:= reset_clocks(ws,s WITH [basic:=s‘basic WITH [Power:=True]])],
pr: s WITH [basic:=s‘basic WITH [Pressure:=False], action_time:= reset_clocks(pr,s WITH [basic:=s‘basic WITH [Pressure:=False]])],
ps: s WITH [basic:=s‘basic WITH [Pressure:=True], action_time:= reset_clocks(ps,s WITH [basic:=s‘basic WITH [Pressure:=True]])]
  ENDCASES

% define initial state

    start (s:I_state):bool =
  ( s = (# activity:=a,
   basic:= (# Relay:=False,Power:=False,Pressure:=False,c1:=0,c2:=0 #),
   action_time:=(LAMBDA (a:I_nt_action): zero)
#)
  )

IMPORTING ttm_lib@machine[I_state,I_action,enabled,trans,start]

END ttm2_decls
```

F.2 TTM Invariants and their proofs

F.2.1 Invariant of specification and its proof

```
ttm1_invariants   : THEORY

   BEGIN

IMPORTING  ttm1_decls

    Inv_1(s:S_state): bool =

(s‘activity=a and (s‘action_time(mu)= zero or s‘action_time(mu)=one)
  and s‘action_time(omega29)=zero and s‘action_time(alpha)=zero
  and s‘action_time(omega19)=zero and s‘action_time(rho1)=zero
  and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

(s‘activity=b and s‘action_time(mu)= zero
  and (s‘action_time(omega29)>=zero and s‘action_time(omega29)<=twenty_nine)
  and s‘action_time(alpha)=zero and s‘action_time(omega19)=zero
  and s‘action_time(rho1)=zero and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

(s‘activity=c and s‘action_time(mu)= zero and s‘action_time(omega29)=zero
  and ((s‘action_time(alpha)=zero and s‘action_time(rho1)=zero)
 or (s‘action_time(alpha)=one and s‘action_time(rho1)=zero)
 or (s‘action_time(alpha)=zero and s‘action_time(rho1)=one))
  and s‘action_time(omega19)=zero
  and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

(s‘activity=d and s‘action_time(mu)= zero and s‘action_time(omega29)=zero
  and s‘action_time(alpha)=zero
  and (s‘action_time(omega19)>=zero and s‘action_time(omega19)<=nineteen)
  and s‘action_time(rho1)=zero and s‘action_time(rho2)=zero and s‘action_time(gamma)=zero) or

(s‘activity=e and s‘action_time(mu)= zero and s‘action_time(omega29)=zero
  and s‘action_time(alpha)=zero and  s‘action_time(omega19)= zero and s‘action_time(rho1)=zero
                          and       ((s‘action_time(rho2)=zero or s‘action_time(gamma)=zero)
or  (s‘action_time(rho2)=zero or s‘action_time(gamma)=one)
or  (s‘action_time(rho2)=one or s‘action_time(gamma)=zero)))

Inv_11(s:S_state): bool = ((NOT s‘basic‘Power) => s‘action_time(wr) = zero) and
```

```
 (s‘basic‘Power => s‘action_time(ws) = zero) and
 ((NOT s‘basic‘Pressure) => s‘action_time(pr) = zero) and
 ( s‘basic‘Pressure => s‘action_time(ps) = zero)

   lemma_1: LEMMA (FORALL (s:S_state): reachable(s) => Inv_1(s));

lemma1_2: LEMMA (FORALL (s:S_state): reachable(s) => Inv_11(s));

  END ttm1_invariants

Proof scripts for file ttm1invariant.pvs:


ttm1_invariants.Inv_1_TCC1: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC2: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC3: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC4: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC5: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC6: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC7: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC8: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC9: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC10: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.Inv_1_TCC11: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))
```

```
ttm1_invariants.Inv_1_TCC12: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC13: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC14: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC15: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC16: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC17: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC18: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC19: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC20: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC21: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC22: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC23: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC24: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC25: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))
```

```
ttm1_invariants.Inv_1_TCC26: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC27: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC28: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC29: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC30: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC31: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC32: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC33: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC34: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC35: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_1_TCC36: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_11_TCC1: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_11_TCC2: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm1_invariants.Inv_11_TCC3: proved - complete [0](n/a s)
```

```
("" (SUBTYPE-TCC))


ttm1_invariants.Inv_11_TCC4: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm1_invariants.lemma_1: proved - complete [0](146.96 s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_1")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (HIDE 2) (GRIND))
  ("3"
   (HIDE 2)
   (EXPAND "inductstep")
   (SKOLEM-TYPEPRED)
   (FLATTEN)
   (EXPAND "enabled")
   (SPLIT)
   (("1"
     (FLATTEN)
     (EXPAND "enabled_time")
     (FLATTEN)
     (EXPAND "enabled_general")
     (FLATTEN)
     (EXPAND "enabled_state")
     (EXPAND "graph")
     (CASE "a!1=tick")
     (("1" (GRIND))
      ("2"
       (CASE "a!1=mu")
       (("1" (REPLACE -1 *) (GRIND))
        ("2"
         (CASE "a!1=omega29")
         (("1" (GRIND))
          ("2"
           (CASE "a!1=alpha")
           (("1" (GRIND))
            ("2"
             (CASE "a!1=omega19")
             (("1" (GRIND))
              ("2"
               (CASE "a!1=rho1")
               (("1" (GRIND))
                ("2"
                 (CASE "a!1=rho2")
                 (("1" (GRIND))
                  ("2"
                   (CASE "a!1=gamma")
                   (("1" (GRIND))
                    ("2"
                     (CASE "a!1=wr")
                     (("1" (REPLACE -1 10) (GRIND))
                      ("2"
                       (CASE "a!1=ws")
                       (("1" (REPLACE -1 11) (GRIND))
                        ("2"
                         (HIDE -1 -2 -3 -4)
                         (CASE "a!1=pr")
                         (("1" (REPLACE -1 12) (GRIND))
                          ("2"
                           (CASE "a!1=ps")
                           (("1" (REPLACE -1 13) (GRIND))
```

```
                          ("2" (GRIND)))))))))))))))))))))))))))))
    ("2"
     (FLATTEN)
     (EXPAND "enabled_tick")
     (CASE "a!1=tick")
     (("1"
       (REPLACE -1 *)
       (HIDE -2)
       (HIDE -3)
       (EXPAND "Inv_1" -3)
       (SPLIT)
       (("1" (GRIND)) ("2" (FLATTEN) (GRIND))
        ("3"
         (FLATTEN)
         (SPLIT)
         (("1" (GRIND)) ("2" (INST - "alpha") (GRIND))
          ("3" (FLATTEN) (INST - "rho1") (GRIND))))
        ("4" (FLATTEN) (GRIND))
        ("5"
         (FLATTEN)
         (SPLIT)
         (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND))
          ("4" (INST - "gamma") (GRIND)) ("5" (INST - "rho2") (GRIND))
          ("6" (GRIND))))))
      ("2" (GRIND))))))))


ttm1_invariants.lemma1_2: proved - complete [0](111.92 s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_11")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (HIDE 2) (GRIND))
  ("3"
   (HIDE 2)
   (EXPAND "inductstep")
   (SKOLEM-TYPEPRED)
   (FLATTEN)
   (EXPAND "trans")
   (GRIND))))
```

F.2.2 Invariant of implementation and its proof

```
ttm2_invariants   : THEORY

   BEGIN

  IMPORTING  ttm2_decls

  Inv_2(s:I_state) : bool =
(s`basic`c1=0 and s`basic`c2=0
  and (s`action_time(mu1)= zero or s`action_time(mu1)=one)
  and s`action_time(mu2)= zero and s`action_time(alpha)=zero
  and s`action_time(rho1)=zero and s`action_time(rho2)= zero
  and s`action_time(gamma)=zero ) or

((s`basic`c1>=1 and s`basic`c1<=29)
  and s`basic`c2=0
  and (s`action_time(mu1)= zero or s`action_time(mu1)= one)
  and s`action_time(mu2)= zero and s`action_time(alpha)=zero
  and s`action_time(rho1)=zero and s`action_time(rho2)= zero
  and s`action_time(gamma)=zero ) or

(s`basic`c1=30 and s`basic`c2=0 and s`action_time(mu1)= zero
  and s`action_time(mu2)= zero
```

```
  and (    (s‘action_time(alpha)=zero and s‘action_time(rho1)=zero)
or (s‘action_time(alpha)=zero and s‘action_time(rho1)=one)
or (s‘action_time(alpha)=one  and  s‘action_time(rho1)=zero))
  and s‘action_time(rho2)= zero and s‘action_time(gamma)=zero ) or

(s‘basic‘c1=0
  and (s‘basic‘c2>=1 and s‘basic‘c2<=19)
  and s‘action_time(mu1)= zero
  and (s‘action_time(mu2)= zero or s‘action_time(mu2)= one)
  and s‘action_time(alpha)=zero and s‘action_time(rho1)=zero
  and s‘action_time(rho2)= zero and s‘action_time(gamma)=zero ) or

(s‘basic‘c1=0 and s‘basic‘c2=20 and s‘action_time(mu1)= zero
  and s‘action_time(mu2)= zero  and s‘action_time(alpha)=zero
  and s‘action_time(rho1)=zero
  and (    (s‘action_time(rho2)= zero and s‘action_time(gamma)=zero)
or (s‘action_time(rho2)= zero and s‘action_time(gamma)=one)
or (s‘action_time(rho2)= one and s‘action_time(gamma)=zero)))

Inv_21(s:I_state): bool = (s‘basic‘c2 >= 1 => s‘basic‘Relay = True)

LEMMA2_1: LEMMA (FORALL (s:I_state): reachable(s) => Inv_2(s));

LEMMA2_2: LEMMA (FORALL (s:I_state): reachable(s) => Inv_21(s));

END ttm2_invariants

Proof scripts for file ttm2invariant.pvs:


ttm2_invariants.Inv_2_TCC1: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC2: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC3: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC4: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC5: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC6: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC7: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))


ttm2_invariants.Inv_2_TCC8: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))
```

```
ttm2_invariants.Inv_2_TCC9: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC10: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC11: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC12: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC13: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC14: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC15: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC16: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC17: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC18: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC19: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC20: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC21: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC22: proved - complete [0](n/a s)
```

```
("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC23: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC24: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC25: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC26: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC27: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC28: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC29: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC30: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC31: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.Inv_2_TCC32: proved - complete [0](n/a s)

("" (SUBTYPE-TCC))

ttm2_invariants.LEMMA2_1: proved - complete [0](n/a s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_2")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (GRIND))
  ("3"
   (HIDE 2)
   (EXPAND "inductstep")
   (SKOLEM-TYPEPRED)
   (FLATTEN)
   (EXPAND "enabled")
```

```
(SPLIT)
(("1"
  (FLATTEN)
  (EXPAND "enabled_time")
  (FLATTEN)
  (EXPAND "enabled_general")
  (FLATTEN)
  (EXPAND "enabled_state")
  (EXPAND "graph")
  (CASE "a!1=tick")
  (("1" (GRIND))
   ("2"
    (CASE "a!1=mu1")
    (("1" (REPLACE -1 *) (GRIND))
     ("2"
      (CASE "a!1=alpha")
      (("1" (REPLACE -1 *) (GRIND))
       ("2"
        (CASE "a!1=mu2")
        (("1" (GRIND))
         ("2"
          (CASE "a!1=rho1")
          (("1" (REPLACE -1 *) (GRIND))
           ("2"
            (CASE "a!1=rho2")
            (("1" (REPLACE -1 *) (GRIND))
             ("2"
              (CASE "a!1=gamma")
              (("1" (REPLACE -1 *) (GRIND))
               ("2"
                (CASE "a!1=wr")
                (("1" (REPLACE -1 *) (GRIND))
                 ("2"
                  (CASE "a!1=ws")
                  (("1" (REPLACE -1 *) (GRIND))
                   ("2"
                    (CASE "a!1=pr")
                    (("1" (REPLACE -1 *) (GRIND))
                     ("2"
                      (CASE "a!1=ps")
                      (("1" (REPLACE -1 *) (GRIND))
                       ("2" (GRIND)))))))))))))))))))))))
 ("2"
  (FLATTEN)
  (EXPAND "enabled_tick")
  (CASE "a!1=tick")
  (("1"
    (REPLACE -1 *)
    (HIDE -2 -3)
    (EXPAND "Inv_2" -3)
    (SPLIT)
    (("1" (GRIND) (REVEAL -2) (INST - "mu1") (GRIND))
     ("2"
      (FLATTEN)
      (SPLIT)
      (("1" (GRIND)) ("2" (REVEAL -2) (INST - "mu1") (GRIND))))
     ("3"
      (GRIND)
      (("1" (REVEAL -2) (INST - "rho1") (GRIND))
       ("2" (REVEAL -2) (INST - "alpha") (GRIND))))
     ("4" (GRIND) (REVEAL -2) (INST - "mu2") (GRIND))
     ("5"
      (GRIND)
      (("1" (REVEAL -2) (INST - "gamma") (GRIND))
       ("2" (REVEAL -2) (INST - "rho2") (GRIND))))))
   ("2" (ASSERT))))))))
```

```
ttm2_invariants.LEMMA2_2: proved - complete [0](n/a s)

(""
 (LEMMA "machine_induct")
 (INST -1 "Inv_21")
 (EXPAND "inductthm")
 (SPLIT)
 (("1" (PROPAX)) ("2" (HIDE 2) (GRIND))
  ("3" (HIDE 2) (EXPAND "inductstep") (SKOLEM-TYPEPRED) (FLATTEN) (GRIND))))
```

F.3 Weak Equivalence and proof of specification between implementation

```
sebisimul: THEORY

BEGIN

IMPORTING ttm1_invariants
IMPORTING ttm2_invariants

% Beginning of definition of intermediate states and actions etc.

action: DATATYPE
BEGIN
 tick: tick?
  tau: tau?
END action

s1,s10: VAR S_state;
s2,s20: VAR I_state;
a1,a11: VAR S_action;
a2,a22: VAR I_action;

state: TYPE = s_internal_state

ps1(s1): state = s1`basic

ps2(s2): state = (# Relay:=s2`basic`Relay,
    Power:=s2`basic`Power,
    Pressure:=s2`basic`Pressure #)

pa1(a1): action =
CASES a1 OF
tick: tick,
   mu: tau,
          omega29: tau,
   alpha: tau,
   omega19: tau,
rho1: tau,
   rho2: tau,
gamma: tau,
wr: tau,
   ws: tau,
   pr: tau,
   ps: tau
ENDCASES

pa2(a2): action =
CASES a2 OF
tick: tick,
   mu1: tau,
   alpha: tau,
   mu2: tau,
```

```
rho1: tau,
   rho2: tau,
gamma: tau,
wr: tau,
   ws: tau,
   pr: tau,
   ps: tau
ENDCASES

Is_tau?(a:action): bool = (a=tau)

dd1(s1,a1,s10): bool =
enabled(a1,s1)&(s10 = trans(a1,s1))

dd2(s2,a2,s20): bool =
enabled(a2,s2)&(s20 = trans(a2,s2))

IMPORTING ttm_lib@sebisim[state,S_state,I_state,action,S_action,I_action,ps1,ps2,pa1,pa2,dd1,dd2,Is_tau?]

sim12_2: LEMMA    (s1`activity = b
   &  s2`basic`c1 >= 1
&  s2`basic`c1 <= 29
&  ((s1`action_time(omega29) = fintime(s2`basic`c1) AND
         s2`action_time(mu1) = one)
        OR
        (s1`action_time(omega29) + one = fintime(s2`basic`c1) AND
          s2`action_time(mu1) = zero))
&  enabled(a1, s1)
&  Inv_11(s1)
&  Inv_21(s2)
&  Inv_1(s1)
&  Inv_2(s2)
&  s1`action_time(alpha) = s2`action_time(alpha)
  &  s1`action_time(rho1) = s2`action_time(rho1)
  &  s1`action_time(rho2) = s2`action_time(rho2)
  &  s1`action_time(gamma) = s2`action_time(gamma)
&  s1`action_time(wr) = s2`action_time(wr)
&  s1`action_time(ws) = s2`action_time(ws)
&  s1`action_time(pr) = s2`action_time(pr)
&  s1`action_time(ps) = s2`action_time(ps)
&  ps1(s1) = ps2(s2))
=>
   (EXISTS (s2a: I_state),
            (a22: I_action), (s2b: I_state), (s2c: I_state):
        (   sitc(r2, ps2)(s2, s2a) & dd2(s2a, a22, s2b)
          & sitc(r2, ps2)(s2b, s2c) & ps1(trans(a1, s1)) = ps2(s2c)
          & pa1(a1) = pa2(a22) & Inv_11(trans(a1, s1)) & Inv_21(s2c)
          & Inv_1(trans(a1, s1)) & Inv_2(s2c)
          & trans(a1, s1)`action_time(alpha) = s2c`action_time(alpha)
          & trans(a1, s1)`action_time(rho1) = s2c`action_time(rho1)
          & trans(a1, s1)`action_time(rho2) = s2c`action_time(rho2)
          & trans(a1, s1)`action_time(gamma) = s2c`action_time(gamma)
          & trans(a1, s1)`action_time(wr) = s2c`action_time(wr)
          & trans(a1, s1)`action_time(ws) = s2c`action_time(ws)
          & trans(a1, s1)`action_time(pr) = s2c`action_time(pr)
          & trans(a1, s1)`action_time(ps) = s2c`action_time(ps)
          & ps1(trans(a1, s1)) = ps2(s2c)
          & (    (trans(a1, s1)`activity = a AND
                   (s2c`basic`c1 = 0 & s2c`basic`c2 = 0) AND
                    (trans(a1, s1)`action_time(mu) =
                      s2c`action_time(mu1)))
              OR (trans(a1, s1)`activity = b AND
                   s2c`basic`c1 >= 1 AND
                    s2c`basic`c1 <= 29 AND
                     ((trans(a1, s1)`action_time(omega29) =
                        fintime(s2c`basic`c1)
                        AND s2c`action_time(mu1) = one)
                       OR
```

```
                      (trans(a1, s1)`action_time(omega29) + one =
                        fintime(s2c`basic`c1)
                        AND s2c`action_time(mu1) = zero)))
              OR (trans(a1, s1)`activity = c AND s2c`basic`c1 = 30)
              OR (trans(a1, s1)`activity = d AND
                   s2c`basic`c2 >= 1 AND
                    s2c`basic`c2 <= 19 AND
                     ((trans(a1, s1)`action_time(omega19) =
                        fintime(s2c`basic`c2)
                        AND s2c`action_time(mu2) = one)
                       OR
                       (trans(a1, s1)`action_time(omega19) + one =
                         fintime(s2c`basic`c2)
                         AND s2c`action_time(mu2) = zero)))
               OR (trans(a1, s1)`activity = e AND s2c`basic`c2 = 20))))
or   (EXISTS (s2d: I_state):
             ps1(trans(a1, s1)) = ps2(s2d) AND Is_tau?(pa1(a1))
         AND sitc(r2, ps2)(s2, s2d) AND Inv_11(trans(a1, s1))
         AND Inv_21(s2d) AND Inv_1(trans(a1, s1)) AND Inv_2(s2d)
         AND trans(a1, s1)`action_time(alpha) = s2d`action_time(alpha)
         AND trans(a1, s1)`action_time(rho1) = s2d`action_time(rho1)
         AND trans(a1, s1)`action_time(rho2) = s2d`action_time(rho2)
         AND trans(a1, s1)`action_time(gamma) = s2d`action_time(gamma)
         AND trans(a1, s1)`action_time(wr) = s2d`action_time(wr)
         AND trans(a1, s1)`action_time(ws) = s2d`action_time(ws)
         AND trans(a1, s1)`action_time(pr) = s2d`action_time(pr)
         AND trans(a1, s1)`action_time(ps) = s2d`action_time(ps)
         AND ps1(trans(a1, s1)) = ps2(s2d)
         AND (    (trans(a1, s1)`activity = a AND
                    (s2d`basic`c1 = 0 & s2d`basic`c2 = 0) AND
                     (trans(a1, s1)`action_time(mu) =
                       s2d`action_time(mu1)))
               OR (trans(a1, s1)`activity = b AND
                    s2d`basic`c1 >= 1 AND
                     s2d`basic`c1 <= 29 AND
                      ((trans(a1, s1)`action_time(omega29) =
                         fintime(s2d`basic`c1)
                         AND s2d`action_time(mu1) = one)
                        OR
                        (trans(a1, s1)`action_time(omega29) + one =
                          fintime(s2d`basic`c1)
                          AND s2d`action_time(mu1) = zero)))
               OR (trans(a1, s1)`activity = c AND s2d`basic`c1 = 30)
               OR (trans(a1, s1)`activity = d AND
                    s2d`basic`c2 >= 1 AND
                     s2d`basic`c2 <= 19 AND
                      ((trans(a1, s1)`action_time(omega19) =
                         fintime(s2d`basic`c2)
                         AND s2d`action_time(mu2) = one)
                        OR
                        (trans(a1, s1)`action_time(omega19) + one =
                          fintime(s2d`basic`c2)
                          AND s2d`action_time(mu2) = zero)))
               OR (trans(a1, s1)`activity = e AND s2d`basic`c2 = 20)))



sim12_4: LEMMA  ((s1`activity = d AND
         s2`basic`c2 >= 1 AND
          s2`basic`c2 <= 19 AND
           ((s1`action_time(omega19) = fintime(s2`basic`c2) AND
              s2`action_time(mu2) = one)
             OR
             (s1`action_time(omega19) + one = fintime(s2`basic`c2) AND
               s2`action_time(mu2) = zero)))
&  enabled(a1, s1)
&  Inv_11(s1)
&  Inv_21(s2)
```

```
&  Inv_1(s1)
&  Inv_2(s2)
&  s1‘action_time(alpha) = s2‘action_time(alpha)
&  s1‘action_time(rho1) = s2‘action_time(rho1)
&  s1‘action_time(rho2) = s2‘action_time(rho2)
&  s1‘action_time(gamma) = s2‘action_time(gamma)
&  s1‘action_time(wr) = s2‘action_time(wr)
&  s1‘action_time(ws) = s2‘action_time(ws)
&  s1‘action_time(pr) = s2‘action_time(pr)
&  s1‘action_time(ps) = s2‘action_time(ps)
&  ps1(s1) = ps2(s2))
   =>
    (EXISTS (s2a: I_state),
              (a22: I_action), (s2b: I_state), (s2c: I_state):
         (   sitc(r2, ps2)(s2, s2a) & dd2(s2a, a22, s2b)
           & sitc(r2, ps2)(s2b, s2c) & ps1(trans(a1, s1)) = ps2(s2c)
           & pa1(a1) = pa2(a22) & Inv_11(trans(a1, s1)) & Inv_21(s2c)
           & Inv_1(trans(a1, s1)) & Inv_2(s2c)
           & trans(a1, s1)‘action_time(alpha) = s2c‘action_time(alpha)
           & trans(a1, s1)‘action_time(rho1) = s2c‘action_time(rho1)
           & trans(a1, s1)‘action_time(rho2) = s2c‘action_time(rho2)
           & trans(a1, s1)‘action_time(gamma) = s2c‘action_time(gamma)
           & trans(a1, s1)‘action_time(wr) = s2c‘action_time(wr)
           & trans(a1, s1)‘action_time(ws) = s2c‘action_time(ws)
           & trans(a1, s1)‘action_time(pr) = s2c‘action_time(pr)
           & trans(a1, s1)‘action_time(ps) = s2c‘action_time(ps)
           & ps1(trans(a1, s1)) = ps2(s2c)
           & (    (trans(a1, s1)‘activity = a AND
                    (s2c‘basic‘c1 = 0 & s2c‘basic‘c2 = 0) AND
                     (trans(a1, s1)‘action_time(mu) =
                       s2c‘action_time(mu1)))
               OR (trans(a1, s1)‘activity = b AND
                    s2c‘basic‘c1 >= 1 AND
                     s2c‘basic‘c1 <= 29 AND
                      ((trans(a1, s1)‘action_time(omega29) =
                         fintime(s2c‘basic‘c1)
                         AND s2c‘action_time(mu1) = one)
                        OR
                        (trans(a1, s1)‘action_time(omega29) + one =
                          fintime(s2c‘basic‘c1)
                          AND s2c‘action_time(mu1) = zero)))
               OR (trans(a1, s1)‘activity = c AND s2c‘basic‘c1 = 30)
               OR (trans(a1, s1)‘activity = d AND
                    s2c‘basic‘c2 >= 1 AND
                     s2c‘basic‘c2 <= 19 AND
                      ((trans(a1, s1)‘action_time(omega19) =
                         fintime(s2c‘basic‘c2)
                         AND s2c‘action_time(mu2) = one)
                        OR
                        (trans(a1, s1)‘action_time(omega19) + one =
                          fintime(s2c‘basic‘c2)
                          AND s2c‘action_time(mu2) = zero)))
               OR (trans(a1, s1)‘activity = e AND s2c‘basic‘c2 = 20))))
or   (EXISTS (s2d: I_state):
             ps1(trans(a1, s1)) = ps2(s2d) AND Is_tau?(pa1(a1))
          AND sitc(r2, ps2)(s2, s2d) AND Inv_11(trans(a1, s1))
          AND Inv_21(s2d) AND Inv_1(trans(a1, s1)) AND Inv_2(s2d)
          AND trans(a1, s1)‘action_time(alpha) = s2d‘action_time(alpha)
          AND trans(a1, s1)‘action_time(rho1) = s2d‘action_time(rho1)
          AND trans(a1, s1)‘action_time(rho2) = s2d‘action_time(rho2)
          AND trans(a1, s1)‘action_time(gamma) = s2d‘action_time(gamma)
          AND trans(a1, s1)‘action_time(wr) = s2d‘action_time(wr)
          AND trans(a1, s1)‘action_time(ws) = s2d‘action_time(ws)
         AND trans(a1, s1)‘action_time(pr) = s2d‘action_time(pr)
          AND trans(a1, s1)‘action_time(ps) = s2d‘action_time(ps)
          AND ps1(trans(a1, s1)) = ps2(s2d)
          AND (    (trans(a1, s1)‘activity = a AND
                     (s2d‘basic‘c1 = 0 & s2d‘basic‘c2 = 0) AND
```

```
                    (trans(a1, s1)‘action_time(mu) =
                      s2d‘action_time(mu1)))
              OR (trans(a1, s1)‘activity = b AND
                   s2d‘basic‘c1 >= 1 AND
                    s2d‘basic‘c1 <= 29 AND
                     ((trans(a1, s1)‘action_time(omega29) =
                        fintime(s2d‘basic‘c1)
                        AND s2d‘action_time(mu1) = one)
                       OR
                       (trans(a1, s1)‘action_time(omega29) + one =
                         fintime(s2d‘basic‘c1)
                         AND s2d‘action_time(mu1) = zero)))
              OR (trans(a1, s1)‘activity = c AND s2d‘basic‘c1 = 30)
              OR (trans(a1, s1)‘activity = d AND
                   s2d‘basic‘c2 >= 1 AND
                    s2d‘basic‘c2 <= 19 AND
                     ((trans(a1, s1)‘action_time(omega19) =
                        fintime(s2d‘basic‘c2)
                        AND s2d‘action_time(mu2) = one)
                       OR
                       (trans(a1, s1)‘action_time(omega19) + one =
                         fintime(s2d‘basic‘c2)
                         AND s2d‘action_time(mu2) = zero)))
              OR (trans(a1, s1)‘activity = e AND s2d‘basic‘c2 = 20)))


sim21_2121: LEMMA
  ((a22 = mu1)
&  enabled_general(a22, s2)
&  enabled_time(a22, s2)
&  s1‘activity = b
&  s2‘basic‘c1 >= 1
&  s2‘basic‘c1 <= 29
&  ((s1‘action_time(omega29) = fintime(s2‘basic‘c1) AND
         s2‘action_time(mu1) = one)
        OR
          (s1‘action_time(omega29) + one = fintime(s2‘basic‘c1) AND
          s2‘action_time(mu1) = zero))
&  Inv_11(s1)
&  Inv_21(s2)
&  Inv_1(s1)
&  Inv_2(s2)
&  s1‘action_time(alpha) = s2‘action_time(alpha)
&  s1‘action_time(rho1) = s2‘action_time(rho1)
&  s1‘action_time(rho2) = s2‘action_time(rho2)
&  s1‘action_time(gamma) = s2‘action_time(gamma)
&  s1‘action_time(wr) = s2‘action_time(wr)
&  s1‘action_time(ws) = s2‘action_time(ws)
&  s1‘action_time(pr) = s2‘action_time(pr)
&  s1‘action_time(ps) = s2‘action_time(ps)
&  ps1(s1) = ps2(s2))
=>
   ((tick?(a22))
       OR (EXISTS (s2a: S_state),
             (a2: S_action), (s2b: S_state), (s2c: S_state):
         (   sitc(r2, ps1)(s1, s2a) & dd1(s2a, a2, s2b)
           & sitc(r2, ps1)(s2b, s2c) & ps2(trans(a22, s2)) = ps1(s2c)
          & pa2(a22) = pa1(a2) & Inv_11(s2c) & Inv_21(trans(a22,s2)) & Inv_1(s2c) & Inv_2(trans(a22, s2))
           & s2c‘action_time(alpha) = trans(a22, s2)‘action_time(alpha)
           & s2c‘action_time(rho1) = trans(a22, s2)‘action_time(rho1)
           & s2c‘action_time(rho2) = trans(a22, s2)‘action_time(rho2)
           & s2c‘action_time(gamma) = trans(a22, s2)‘action_time(gamma)
           & s2c‘action_time(wr) = trans(a22, s2)‘action_time(wr)
           & s2c‘action_time(ws) = trans(a22, s2)‘action_time(ws)
           & s2c‘action_time(pr) = trans(a22, s2)‘action_time(pr)
           & s2c‘action_time(ps) = trans(a22, s2)‘action_time(ps)
           & ps1(s2c) = ps2(trans(a22, s2))
           & (    (s2c‘activity = a AND
```

```
                    (trans(a22, s2)‘basic‘c1 = 0 &
                      trans(a22, s2)‘basic‘c2 = 0)
                     AND
                     (s2c‘action_time(mu) =
                       trans(a22, s2)‘action_time(mu1)))
               OR (s2c‘activity = b AND
                    trans(a22, s2)‘basic‘c1 >= 1 AND
                     trans(a22, s2)‘basic‘c1 <= 29 AND
                      ((s2c‘action_time(omega29) =
                         fintime(trans(a22, s2)‘basic‘c1)
                         AND trans(a22, s2)‘action_time(mu1) = one)
                        OR
                        (s2c‘action_time(omega29) + one =
                          fintime(trans(a22, s2)‘basic‘c1)
                          AND trans(a22, s2)‘action_time(mu1) = zero)))
               OR (s2c‘activity = c AND trans(a22, s2)‘basic‘c1 = 30)
               OR (s2c‘activity = d AND
                    trans(a22, s2)‘basic‘c2 >= 1 AND
                     trans(a22, s2)‘basic‘c2 <= 19 AND
                      ((s2c‘action_time(omega19) =
                         fintime(trans(a22, s2)‘basic‘c2)
                         AND trans(a22, s2)‘action_time(mu2) = one)
                        OR
                        (s2c‘action_time(omega19) + one =
                          fintime(trans(a22, s2)‘basic‘c2)
                          AND trans(a22, s2)‘action_time(mu2) = zero)))
               OR (s2c‘activity = e AND trans(a22, s2)‘basic‘c2 = 20))))
OR   (EXISTS (s2d: S_state):
             ps2(trans(a22, s2)) = ps1(s2d) AND Is_tau?(pa2(a22))
          AND sitc(r2, ps1)(s1, s2d) AND Inv_11(s2d) AND Inv_21(trans(a22,s2)) AND Inv_1(s2d)
          AND Inv_2(trans(a22, s2))
          AND s2d‘action_time(alpha) = trans(a22, s2)‘action_time(alpha)
          AND s2d‘action_time(rho1) = trans(a22, s2)‘action_time(rho1)
          AND s2d‘action_time(rho2) = trans(a22, s2)‘action_time(rho2)
          AND s2d‘action_time(gamma) = trans(a22, s2)‘action_time(gamma)
          AND s2d‘action_time(wr) = trans(a22, s2)‘action_time(wr)
         AND s2d‘action_time(ws) = trans(a22, s2)‘action_time(ws)
          AND s2d‘action_time(pr) = trans(a22, s2)‘action_time(pr)
          AND s2d‘action_time(ps) = trans(a22, s2)‘action_time(ps)
          AND ps1(s2d) = ps2(trans(a22, s2))
          AND (    (s2d‘activity = a AND
                     (trans(a22, s2)‘basic‘c1 = 0 &
                       trans(a22, s2)‘basic‘c2 = 0)
                       AND
                      (s2d‘action_time(mu) =
                        trans(a22, s2)‘action_time(mu1)))
                OR (s2d‘activity = b AND
                     trans(a22, s2)‘basic‘c1 >= 1 AND
                      trans(a22, s2)‘basic‘c1 <= 29 AND
                       ((s2d‘action_time(omega29) =
                          fintime(trans(a22, s2)‘basic‘c1)
                          AND trans(a22, s2)‘action_time(mu1) = one)
                         OR
                         (s2d‘action_time(omega29) + one =
                           fintime(trans(a22, s2)‘basic‘c1)
                           AND trans(a22, s2)‘action_time(mu1) = zero)))
                OR (s2d‘activity = c AND trans(a22, s2)‘basic‘c1 = 30)
                OR (s2d‘activity = d AND
                     trans(a22, s2)‘basic‘c2 >= 1 AND
                      trans(a22, s2)‘basic‘c2 <= 19 AND
                       ((s2d‘action_time(omega19) =
                          fintime(trans(a22, s2)‘basic‘c2)
                          AND trans(a22, s2)‘action_time(mu2) = one)
                         OR
                         (s2d‘action_time(omega19) + one =
                           fintime(trans(a22, s2)‘basic‘c2)
                           AND trans(a22, s2)‘action_time(mu2) = zero)))
                OR (s2d‘activity = e AND trans(a22, s2)‘basic‘c2 = 20))))
```

```
sim21_221: LEMMA    (a22 = tick
%  &  (tick?(tick))
&  enabled_tick(s2)
&  s1‘activity = b
&  s2‘basic‘c1 >= 1
&  s2‘basic‘c1 <= 29
&  ((s1‘action_time(omega29) = fintime(s2‘basic‘c1) AND
          s2‘action_time(mu1) = one)
         OR
         (s1‘action_time(omega29) + one = fintime(s2‘basic‘c1) AND
           s2‘action_time(mu1) = zero))
&  Inv_11(s1)
&  Inv_21(s2)
&  Inv_1(s1)
&  Inv_2(s2)
&  s1‘action_time(alpha) = s2‘action_time(alpha)
&  s1‘action_time(rho1) = s2‘action_time(rho1)
&  s1‘action_time(rho2) = s2‘action_time(rho2)
&  s1‘action_time(gamma) = s2‘action_time(gamma)
&  s1‘action_time(wr) = s2‘action_time(wr)
&  s1‘action_time(ws) = s2‘action_time(ws)
&  s1‘action_time(pr) = s2‘action_time(pr)
&  s1‘action_time(ps) = s2‘action_time(ps)
&  ps1(s1) = ps2(s2))
=>
    EXISTS (s2a: S_state),
              (a2: S_action), (s2b: S_state), (s2c: S_state):
         (   sitc(r2, ps1)(s1, s2a) & dd1(s2a, a2, s2b)
           & sitc(r2, ps1)(s2b, s2c) & ps2(trans(tick, s2)) = ps1(s2c)
           & pa2(tick) = pa1(a2) & Inv_11(s2c) & Inv_21(trans(tick,s2)) & Inv_1(s2c) & Inv_2(trans(tick, s2))
           & s2c‘action_time(alpha) = trans(tick, s2)‘action_time(alpha)
           & s2c‘action_time(rho1) = trans(tick, s2)‘action_time(rho1)
           & s2c‘action_time(rho2) = trans(tick, s2)‘action_time(rho2)
           & s2c‘action_time(gamma) = trans(tick, s2)‘action_time(gamma)
           & s2c‘action_time(wr) = trans(tick, s2)‘action_time(wr)
           & s2c‘action_time(ws) = trans(tick, s2)‘action_time(ws)
           & s2c‘action_time(pr) = trans(tick, s2)‘action_time(pr)
           & s2c‘action_time(ps) = trans(tick, s2)‘action_time(ps)
           & ps1(s2c) = ps2(trans(tick, s2))
           & (    (s2c‘activity = a AND
                    (trans(tick, s2)‘basic‘c1 = 0 &
                      trans(tick, s2)‘basic‘c2 = 0)
                     AND
                     (s2c‘action_time(mu) =
                       trans(tick, s2)‘action_time(mu1)))
               OR (s2c‘activity = b AND
                    trans(tick, s2)‘basic‘c1 >= 1 AND
                     trans(tick, s2)‘basic‘c1 <= 29 AND
                      ((s2c‘action_time(omega29) =
                         fintime(trans(tick, s2)‘basic‘c1)
                         AND trans(tick, s2)‘action_time(mu1) = one)
                        OR
                        (s2c‘action_time(omega29) + one =
                          fintime(trans(tick, s2)‘basic‘c1)
                          AND trans(tick, s2)‘action_time(mu1) = zero)))
               OR (s2c‘activity = c AND trans(tick, s2)‘basic‘c1 = 30)
               OR (s2c‘activity = d AND
                    trans(tick, s2)‘basic‘c2 >= 1 AND
                     trans(tick, s2)‘basic‘c2 <= 19 AND
                      ((s2c‘action_time(omega19) =
                         fintime(trans(tick, s2)‘basic‘c2)
                         AND trans(tick, s2)‘action_time(mu2) = one)
                        OR
                        (s2c‘action_time(omega19) + one =
                          fintime(trans(tick, s2)‘basic‘c2)
                          AND trans(tick, s2)‘action_time(mu2) = zero)))
```

```
             OR (s2c‘activity = e AND trans(tick, s2)‘basic‘c2 = 20)))



sim21_4121: LEMMA   (a22 = mu2
&  enabled_general(mu2, s2)
&  enabled_time(mu2, s2)
&  s1‘activity = d
&  s2‘basic‘c2 >= 1
&  s2‘basic‘c2 <= 19
&  ((s1‘action_time(omega19) = fintime(s2‘basic‘c2) AND
          s2‘action_time(mu2) = one)
         OR
         (s1‘action_time(omega19) + one = fintime(s2‘basic‘c2) AND
           s2‘action_time(mu2) = zero))
&  Inv_11(s1)
&  Inv_21(s2)
&  Inv_1(s1)
&  Inv_2(s2)
&  s1‘action_time(alpha) = s2‘action_time(alpha)
&  s1‘action_time(rho1) = s2‘action_time(rho1)
&  s1‘action_time(rho2) = s2‘action_time(rho2)
&  s1‘action_time(gamma) = s2‘action_time(gamma)
&  s1‘action_time(wr) = s2‘action_time(wr)
&  s1‘action_time(ws) = s2‘action_time(ws)
&  s1‘action_time(pr) = s2‘action_time(pr)
&  s1‘action_time(ps) = s2‘action_time(ps)
&  ps1(s1) = ps2(s2))
                    =>
    ((EXISTS (s2a: S_state),
              (a2: S_action), (s2b: S_state), (s2c: S_state):
         (   sitc(r2, ps1)(s1, s2a) & dd1(s2a, a2, s2b)
           & sitc(r2, ps1)(s2b, s2c) & ps2(trans(mu2, s2)) = ps1(s2c)
           & pa2(mu2) = pa1(a2) & Inv_11(s2c) & Inv_21(trans(mu2,s2)) & Inv_1(s2c) & Inv_2(trans(mu2, s2))
           & s2c‘action_time(alpha) = trans(mu2, s2)‘action_time(alpha)
           & s2c‘action_time(rho1) = trans(mu2, s2)‘action_time(rho1)
           & s2c‘action_time(rho2) = trans(mu2, s2)‘action_time(rho2)
           & s2c‘action_time(gamma) = trans(mu2, s2)‘action_time(gamma)
            & s2c‘action_time(wr) = trans(mu2, s2)‘action_time(wr)
           & s2c‘action_time(ws) = trans(mu2, s2)‘action_time(ws)
           & s2c‘action_time(pr) = trans(mu2, s2)‘action_time(pr)
           & s2c‘action_time(ps) = trans(mu2, s2)‘action_time(ps)
           & ps1(s2c) = ps2(trans(mu2, s2))
           & (    (s2c‘activity = a AND
                    (trans(mu2, s2)‘basic‘c1 = 0 &
                      trans(mu2, s2)‘basic‘c2 = 0)
                     AND
                     (s2c‘action_time(mu) =
                       trans(mu2, s2)‘action_time(mu1)))
               OR (s2c‘activity = b AND
                    trans(mu2, s2)‘basic‘c1 >= 1 AND
                     trans(mu2, s2)‘basic‘c1 <= 29 AND
                      ((s2c‘action_time(omega29) =
                         fintime(trans(mu2, s2)‘basic‘c1)
                         AND trans(mu2, s2)‘action_time(mu1) = one)
                        OR
                        (s2c‘action_time(omega29) + one =
                          fintime(trans(mu2, s2)‘basic‘c1)
                          AND trans(mu2, s2)‘action_time(mu1) = zero)))
               OR (s2c‘activity = c AND trans(mu2, s2)‘basic‘c1 = 30)
               OR (s2c‘activity = d AND
                    trans(mu2, s2)‘basic‘c2 >= 1 AND
                     trans(mu2, s2)‘basic‘c2 <= 19 AND
                      ((s2c‘action_time(omega19) =
                         fintime(trans(mu2, s2)‘basic‘c2)
                         AND trans(mu2, s2)‘action_time(mu2) = one)
                        OR
                        (s2c‘action_time(omega19) + one =
```

```
                        fintime(trans(mu2, s2)`basic`c2)
                        AND trans(mu2, s2)`action_time(mu2) = zero)))
              OR (s2c`activity = e AND trans(mu2, s2)`basic`c2 = 20))))
or   (tick?(mu2))
or   (EXISTS (s2d: S_state):
             ps2(trans(mu2, s2)) = ps1(s2d) AND Is_tau?(pa2(mu2))
          AND sitc(r2, ps1)(s1, s2d) AND Inv_11(s2d) AND Inv_21(trans(mu2,s2)) AND Inv_1(s2d)
          AND Inv_2(trans(mu2, s2))
          AND s2d`action_time(alpha) = trans(mu2, s2)`action_time(alpha)
          AND s2d`action_time(rho1) = trans(mu2, s2)`action_time(rho1)
          AND s2d`action_time(rho2) = trans(mu2, s2)`action_time(rho2)
           AND s2d`action_time(gamma) = trans(mu2, s2)`action_time(gamma)
          AND s2d`action_time(wr) = trans(mu2, s2)`action_time(wr)
          AND s2d`action_time(ws) = trans(mu2, s2)`action_time(ws)
          AND s2d`action_time(pr) = trans(mu2, s2)`action_time(pr)
          AND s2d`action_time(ps) = trans(mu2, s2)`action_time(ps)
          AND ps1(s2d) = ps2(trans(mu2, s2))
          AND (    (s2d`activity = a AND
                     (trans(mu2, s2)`basic`c1 = 0 &
                       trans(mu2, s2)`basic`c2 = 0)
                      AND
                      (s2d`action_time(mu) =
                        trans(mu2, s2)`action_time(mu1)))
                OR (s2d`activity = b AND
                     trans(mu2, s2)`basic`c1 >= 1 AND
                      trans(mu2, s2)`basic`c1 <= 29 AND
                       ((s2d`action_time(omega29) =
                          fintime(trans(mu2, s2)`basic`c1)
                          AND trans(mu2, s2)`action_time(mu1) = one)
                         OR
                         (s2d`action_time(omega29) + one =
                           fintime(trans(mu2, s2)`basic`c1)
                           AND trans(mu2, s2)`action_time(mu1) = zero)))
                OR (s2d`activity = c AND trans(mu2, s2)`basic`c1 = 30)
                OR (s2d`activity = d AND
                     trans(mu2, s2)`basic`c2 >= 1 AND
                      trans(mu2, s2)`basic`c2 <= 19 AND
                       ((s2d`action_time(omega19) =
                          fintime(trans(mu2, s2)`basic`c2)
                          AND trans(mu2, s2)`action_time(mu2) = one)
                         OR
                         (s2d`action_time(omega19) + one =
                           fintime(trans(mu2, s2)`basic`c2)
                           AND trans(mu2, s2)`action_time(mu2) = zero)))
                OR (s2d`activity = e AND trans(mu2, s2)`basic`c2 = 20))))


sim21_42: LEMMA    (NOT (NOT (tick?(a22))) AND enabled_tick(s2)
&  (s1`activity = d AND
         s2`basic`c2 >= 1 AND
          s2`basic`c2 <= 19 AND
           ((s1`action_time(omega19) = fintime(s2`basic`c2) AND
              s2`action_time(mu2) = one)
             OR
             (s1`action_time(omega19) + one = fintime(s2`basic`c2) AND
               s2`action_time(mu2) = zero)))
&  Inv_11(s1)
&  Inv_21(s2)
&  Inv_1(s1)
&  Inv_2(s2)
&  s1`action_time(alpha) = s2`action_time(alpha)
&  s1`action_time(rho1) = s2`action_time(rho1)
&  s1`action_time(rho2) = s2`action_time(rho2)
&  s1`action_time(gamma) = s2`action_time(gamma)
&  s1`action_time(wr) = s2`action_time(wr)
&  s1`action_time(ws) = s2`action_time(ws)
&  s1`action_time(pr) = s2`action_time(pr)
&  s1`action_time(ps) = s2`action_time(ps)
```

```
&  ps1(s1) = ps2(s2))
  =>
    (EXISTS (s2a: S_state),
             (a2: S_action), (s2b: S_state), (s2c: S_state):
         (   sitc(r2, ps1)(s1, s2a) & dd1(s2a, a2, s2b)
           & sitc(r2, ps1)(s2b, s2c) & ps2(trans(a22, s2)) = ps1(s2c)
           & pa2(a22) = pa1(a2) & Inv_11(s2c) & Inv_21(trans(a22,s2)) & Inv_1(s2c) & Inv_2(trans(a22, s2))
           & s2c`action_time(alpha) = trans(a22, s2)`action_time(alpha)
           & s2c`action_time(rho1) = trans(a22, s2)`action_time(rho1)
           & s2c`action_time(rho2) = trans(a22, s2)`action_time(rho2)
           & s2c`action_time(gamma) = trans(a22, s2)`action_time(gamma)
           & s2c`action_time(wr) = trans(a22, s2)`action_time(wr)
           & s2c`action_time(ws) = trans(a22, s2)`action_time(ws)
           & s2c`action_time(pr) = trans(a22, s2)`action_time(pr)
           & s2c`action_time(ps) = trans(a22, s2)`action_time(ps)
           & ps1(s2c) = ps2(trans(a22, s2))
           & (    (s2c`activity = a AND
                    (trans(a22, s2)`basic`c1 = 0 &
                      trans(a22, s2)`basic`c2 = 0)
                     AND
                     (s2c`action_time(mu) =
                       trans(a22, s2)`action_time(mu1)))
                OR (s2c`activity = b AND
                     trans(a22, s2)`basic`c1 >= 1 AND
                      trans(a22, s2)`basic`c1 <= 29 AND
                       ((s2c`action_time(omega29) =
                          fintime(trans(a22, s2)`basic`c1)
                          AND trans(a22, s2)`action_time(mu1) = one)
                         OR
                         (s2c`action_time(omega29) + one =
                           fintime(trans(a22, s2)`basic`c1)
                           AND trans(a22, s2)`action_time(mu1) = zero)))
                OR (s2c`activity = c AND trans(a22, s2)`basic`c1 = 30)
                OR (s2c`activity = d AND
                     trans(a22, s2)`basic`c2 >= 1 AND
                      trans(a22, s2)`basic`c2 <= 19 AND
                       ((s2c`action_time(omega19) =
                          fintime(trans(a22, s2)`basic`c2)
                          AND trans(a22, s2)`action_time(mu2) = one)
                         OR
                         (s2c`action_time(omega19) + one =
                           fintime(trans(a22, s2)`basic`c2)
                           AND trans(a22, s2)`action_time(mu2) = zero)))
                OR (s2c`activity = e AND trans(a22, s2)`basic`c2 = 20))))



Sim1_2: LEMMA  wsesim?(LAMBDA (s1: S_state, s2: I_state):
                   Inv_11(s1) & Inv_21(s2) & Inv_1(s1) & Inv_2(s2)
                 & s1`action_time(alpha) = s2`action_time(alpha)
                 & s1`action_time(rho1) = s2`action_time(rho1)
                 & s1`action_time(rho2) = s2`action_time(rho2)
                 & s1`action_time(gamma) = s2`action_time(gamma)
                 & s1`action_time(wr) = s2`action_time(wr)
                 & s1`action_time(ws) = s2`action_time(ws)
                 & s1`action_time(pr) = s2`action_time(pr)
                 & s1`action_time(ps) = s2`action_time(ps)
                 & ps1(s1) = ps2(s2)
                 & (    (s1`activity = a AND
                          (s2`basic`c1 = 0 & s2`basic`c2 = 0) AND
                           (s1`action_time(mu) = s2`action_time(mu1)))
                     OR (s1`activity = b AND
                          s2`basic`c1 >= 1 AND
                           s2`basic`c1 <= 29 AND
                            ((s1`action_time(omega29) =
                               fintime(s2`basic`c1)
```

```
                          AND s2‘action_time(mu1) = one)
                         OR
                         (s1‘action_time(omega29) + one =
                           fintime(s2‘basic‘c1)
                           AND s2‘action_time(mu1) = zero)))
                  OR (s1‘activity = c AND s2‘basic‘c1 = 30)
                  OR (s1‘activity = d AND
                      s2‘basic‘c2 >= 1 AND
                       s2‘basic‘c2 <= 19 AND
                        ((s1‘action_time(omega19) =
                           fintime(s2‘basic‘c2)
                           AND s2‘action_time(mu2) = one)
                          OR
                          (s1‘action_time(omega19) + one =
                            fintime(s2‘basic‘c2)
                            AND s2‘action_time(mu2) = zero)))
                  OR (s1‘activity = e AND s2‘basic‘c2 = 20)))

Sim2_1: LEMMA  wsesim?(converse(LAMBDA (s1: S_state, s2: I_state):
                        Inv_11(s1) & Inv_21(s2) & Inv_1(s1) & Inv_2(s2)
                      & s1‘action_time(alpha) = s2‘action_time(alpha)
                      & s1‘action_time(rho1) = s2‘action_time(rho1)
                      & s1‘action_time(rho2) = s2‘action_time(rho2)
                      & s1‘action_time(gamma) = s2‘action_time(gamma)
                      & s1‘action_time(wr) = s2‘action_time(wr)
                      & s1‘action_time(ws) = s2‘action_time(ws)
                      & s1‘action_time(pr) = s2‘action_time(pr)
                      & s1‘action_time(ps) = s2‘action_time(ps)
                      & ps1(s1) = ps2(s2)
                      & (    (s1‘activity = a AND
                               (s2‘basic‘c1 = 0 & s2‘basic‘c2 = 0) AND
                                (s1‘action_time(mu)
                                 =
                                 s2‘action_time(mu1)))
                         OR (s1‘activity = b AND
                              s2‘basic‘c1 >= 1 AND
                               s2‘basic‘c1 <= 29
                               AND
                               ((s1‘action_time(omega29)
                                 =
                                 fintime(s2‘basic‘c1)
                                 AND
                                 s2‘action_time(mu1) = one)
                                OR
                                (s1‘action_time(omega29) + one
                                 =
                                 fintime(s2‘basic‘c1)
                                 AND
                                 s2‘action_time(mu1) = zero)))
                         OR (s1‘activity = c AND s2‘basic‘c1 = 30)
                         OR (s1‘activity = d AND
                              s2‘basic‘c2 >= 1 AND
                               s2‘basic‘c2 <= 19
                               AND
                               ((s1‘action_time(omega19)
                                 =
                                 fintime(s2‘basic‘c2)
                                 AND
                                 s2‘action_time(mu2) = one)
                                OR
                                (s1‘action_time(omega19) + one
                                 =
                                 fintime(s2‘basic‘c2)
                                 AND
                                 s2‘action_time(mu2) = zero)))
                         OR (s1‘activity = e AND s2‘basic‘c2 = 20))))
```

```
RR:Wsebisim = LAMBDA(s1,s2):

 ((Inv_11(s1) & Inv_21(s2) & Inv_1(s1) & Inv_2(s2))
& s1‘action_time(alpha)=s2‘action_time(alpha)
& s1‘action_time(rho1)=s2‘action_time(rho1)
& s1‘action_time(rho2)=s2‘action_time(rho2)
& s1‘action_time(gamma)=s2‘action_time(gamma)
& s1‘action_time(wr)=s2‘action_time(wr)
& s1‘action_time(ws)=s2‘action_time(ws)
& s1‘action_time(pr)=s2‘action_time(pr)
& s1‘action_time(ps)=s2‘action_time(ps)
& ps1(s1) = ps2(s2))
&
((s1‘activity=a and (s2‘basic‘c1=0 &s2‘basic‘c2=0) and (s1‘action_time(mu)=s2‘action_time(mu1))) or
(s1‘activity=b and s2‘basic‘c1>=1 and s2‘basic‘c1<=29 and
   ((s1‘action_time(omega29)=fintime(s2‘basic‘c1) and s2‘action_time(mu1)=one) or
   (s1‘action_time(omega29)+ one =fintime(s2‘basic‘c1) and s2‘action_time(mu1)=zero))) or
(s1‘activity=c and s2‘basic‘c1=30 )  or
(s1‘activity=d and s2‘basic‘c2>=1 and s2‘basic‘c2<=19 and
   ((s1‘action_time(omega19)=fintime(s2‘basic‘c2) and s2‘action_time(mu2)=one) or
   (s1‘action_time(omega19)+one=fintime(s2‘basic‘c2) and s2‘action_time(mu2)=zero))) or
(s1‘activity=e and s2‘basic‘c2=20))


weakequi: LEMMA weakequivalence(RR)(start,start)(s1,s2);

END sebisimul
```

Bibliography

[AAG+00] L.de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. Mocha: Exploiting modularity in model checking. Technical report, University of California, Berkeley and University of Pennsylvania, 2000.

[AHR00] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Using tame to prove invariants of automata models: Two case studies. Technical report, Naval Research Laboratory, August 2000.

[BS00] Ramesh Bharadwaj and Steve Sims. Salsa: Combining constraint solvers with bdds for automatic invariant checking. In *Lecture Notes in Computer Science*. Naval Research Laboratory, Springer, 2000.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to pvs. Technical report, Computer Science Laboratory, SRI International, April 1995.

[EGP00] E.M.Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[HA98] Constance Heitmeyer and Myla Archer. Mechanical verification of timed automata: A case study. Technical report, Naval Research Laboratory, April 1998.

[Law92] Mark Stephen Lawford. Transformational equivalence of timed transition models. Master's thesis, University of Toronto, January 1992.

[Law97] M. Lawford. *Model Reduction of Discrete Real-Time Systems*. PhD thesis, Dept. of Elec. & Comp. Eng., Univ. of Toronto, Canada, January 1997.

[LFM00] M. Lawford, P. Froebel, and G. Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown

system. McMaster University and Ontario Power Generation, August 2000.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. Technical report, Aalborg University and Uppsala University, 1997.

[LW95] M. Lawford and W.M. Wonham. Equivalence preserving transformations of timed transition models. *IEEE Trans. Autom. Control*, 40:1167–1179, July 1995.

[LW00] M. Lawford and H. Wu. Verification of real-time control software using PVS. In P. Ramadge and S. Verdu, editors, *Proceedings of the 2000 Conference on Information Sciences and Systems*, volume 2, pages TP1–13–TP1–17, Princeton, NJ, March 2000. Dept. of Electrical Engineering, Princeton University.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[OS97] S. Owre and N. Shankar. Abstract datatypes in pvs. Technical report, SRI International, June 1997.

[Ost95] Jonathan S. Ostroff. Abstraction and composition of discrete real-time systems. Technical report, York University, 1995.

[Ost98] Jonathan S. Ostroff. Composition and refinement of discrete real-time systems. Technical report, York University, 1998.

[Par81] D. Park. Concurrency and automata on infinite sequences. In , pages 167–183. Berlin, West Germany: Springer-Verlag, 1981. LNCS-104.

[SORSC99a] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park CA 94025, September 1999.

[SORSC99b] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. Pvs prover guide. Technical report, Computer Science Laboratory, SRI International, September, 1999.

[Wu01] Hongyu Wu. Formal verification of real-time software. Master's thesis, McMaster University, 2001.