TTM/PAT: A Tool for Modelling and Verifying Timed Transition Models

Jonathan S. Ostroff, Chen-Wei Wang and Simon Hudon

Technical Report CSE-2013-05

June 3 2013

Department of Computer Science and Engineering
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

TTM/PAT: a Tool for Modelling and Verifying Timed Transition Models

Jonathan Ostroff, Chen-Wei Wang, Simon Hudon
EECS, York University

June 4, 2013

Abstract

Timed Transition Models (TTMs) are event based descriptions for specifying and verifying real-time systems in a discrete setting. While the verification of TTMs has been supported in tools such as Uppaal and SAL, the manual encoding requires substantial effort before a TTM can be checked. We propose a convenient and expressive textual syntax for TTMs and a corresponding one-step operational semantics. Modules allow for compositional reasoning and include an interface in which monitored and controlled variables are declared. Events in a module can be specified, individually, as spontaneous, fair or real-time. An event action specifies a before-after predicate by a set of (possibly non-deterministic) assignments and (possibly nested) conditionals. The TTM assertion language, LTL, allows references to event occurrences, including clock ticks (thus allowing for a check that the behaviour is non-zeno). We describe a tool that includes an editor with static type checking, a graphical simulator and a LTL verifier (as a plug-in for the PAT toolset). The tool automatically derives the tick transition and implicit event clocks so that the burden of manual encoding is removed. The TTM tool performs significantly better on a nuclear shutdown system than the manually encoded version in Uppaal and SAL.

Contents

1 Introduction

Checking the correctness of real-time systems is both challenging and important for industrial applications. A variety of theories and tools have been suggested to support the design of correct real-time systems. Modelling languages should be able to specify requirements precisely, concisely and in intuitively appealing ways. A variety of methods have been suggested in the model checking context, as surveyed in [SLD$^+$13]. Uppaal and RTS (also called stateful timed CSP) are representative examples of state-of-the-art tools compared in the survey.

Uppaal provides a graphical description language and simulation facility. Uppaal models systems as a collection of timed automata with real-valued clocks, communicating through channels and shared variables. The assertion language TCTL is a subset of CTL (branching time logic), but fairness constraints are not supported.

By contrast, in RTS, systems are described using a timed process algebra allowing for hierarchical descriptions. There are no explicit clocks. Instead, real-time requirements are stated in terms of deadlines and timed interrupts. Also, processes may communicate via shared variables. The assertion language includes process refinement checking and linear time temporal logic (LTL). A single fairness assumption can be adopted on all events (or processes). RTS is a plug-in developed within the PAT toolset.

In this paper, we describe the TTM language and tool which is developed as a plug-in of the PAT toolset. In contrast to Uppaal and RTS, the TTM language is an event-based method along the lines of Event-B [Abr10], but with real-time features including event lower and upper time bounds and the ability to use explicit timers. In the past, TTMs have been manually encoded into the Uppaal or SAL model checkers [LPZ06]. Our paper makes the following technical contributions:

- We propose a convenient and expressive textual syntax for TTMs and a corresponding one-step operational semantics.
- The TTM language addresses a variety of needs. (1) Modules allow for compositional reasoning and include an interface in which **in**, **out** and **share** variables are declared. (2) Events in a module can be specified, individually, as spontaneous, fair or real-time. (3) An event action specifies a before-after predicate with a set of simultaneous and possibly nondeterministic assignments structured in nested conditionals. (4) The specification language is LTL and includes the ability to refer to the occurrence of an event, including a tick of the clock; this also allows for a check that the behaviour is non-zeno. (5) Where explicit timers are used in verifying a property, the timers can be checked for monotonicity, thus checking that the timers remain in synch with the global clock and that the properties express real-time constraints accurately.
- Tool support for the TTM language is provided as a plug-in to the PAT toolset. The TTM tool has an editor that does static type checking, an integrated simulator and a verifier for LTL model-checking. The tool automatically derives the tick transition and implicit event clocks so that the burden of manual encoding is removed. The TTM tool performs significantly better on a nuclear shutdown system than the manually encoded version in Uppaal and SAL.

The rest of the paper is organized as follows. Section 2 introduces the syntax of the TTM language and discusses the use of fairness assumptions and modular reasoning. Section 3 uses a small pacemaker example to illustrate modelling and reasoning about real-time properties. Section 4 provides a one-step operational semantics for TTMs, used in the development of the tool as a PAT plug-in. Section 5 uses two examples to compare the performance of the TTM tool to that of the Uppaal and RTS tools. In Section 6, we conclude with a discussion of future developments to the tool, based on the performance results.

2 TTMs: an Introductory Example

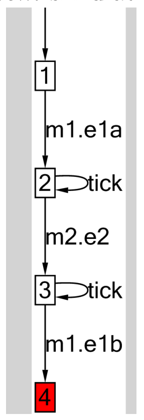
The original tool for describing TTMs and model-checking was graphical [Ost99, MP92]. To introduce the new textual syntax in the TTM/PAT tool, we use example Ping-Pong in Fig. 1; this example also allows us to describe fairness constraints, and to illustrate modular descriptions and reasoning.

Keyword **module** introduces a module template (e.g. template $M1$ in Fig. 1). Module templates have an interface, local variables, and a set of events in the style of Event-B ([Abr10]). Keyword **instances** list the modules which are instances of the templates. For example, module $m1$ (in Fig. 1) is an instance

```
type BIT = 0..1 end

module M1
interface
  y: in BIT;
  x: out BIT = 0;
  z: out BIT = 0
local pc : INT = 0//program counter
events
  e1a[0,*] just
  when pc==0
  do x := 1, pc:=1 end

  e1b just
  when y==1 && pc==1
  do z := 1, pc := 2 end
end

module M2
interface
  y: out BIT = 0; x: in BIT
 events
   e2 just
   when x==1
   do y:= 1
   end
end

module ENV1//arbitrary environment
interface
  y: out BIT = 0
 events
   demonic do y::BIT end
 end

module ENV2
interface
  x: out BIT = 0
 events
   demonic do x:: BIT end
 end
```

```
instances
 m1 = M1 (in y, out x, out z)
 m2 = M2(out y, in x)
 env1 = ENV1(out y)
 env2 = ENV2(out x)
end

composition
   system = m1 || m2
// modular validity
   m1_env = m1 || env1
   m2_env = m2 || env2
end

#assert system typecorrect;

//check global property
#define z1 z==1;
#assert system |= <>z1;

//check via modular validity
#define x1 x==1;
#define y1 y==1;
#assert m1_env |= <>[]x1;
#assert m1_env |= <>[]y1 -> <>z1;
#assert m2_env |= <>[]x1 -> <>[]y1;
```

Display of *system* simulation in the



tool:

Figure 1: Modular reasoning for TTM Ping-Pong

of a template $M1$. Keyword **composition** provides the systems (composed of modules that execute in parallel with each other) that will be modelchecked. For example, for Ping-Pong we have $system = m1||m2$. Keyword **assert** introduces linear time temporal logic properties that will be checked. For example, we can check that: $system \models \Diamond(z == 1)$.

A module interface declares a list of variables, their types and their mode (either **in**, **out**, or **share**). The variable types may be enumerated sets, BOOL, INT, integer intervals and arrays of these basic types. Some static type checking is supported. Also, in model checking, all variables of finite types may be checked for type correctness as one of the supported verification conditions (see the use of keyword **typecorrect** in the figure for an example). We may also compose an array of modules from the same template (see the Fischer example in the sequel).

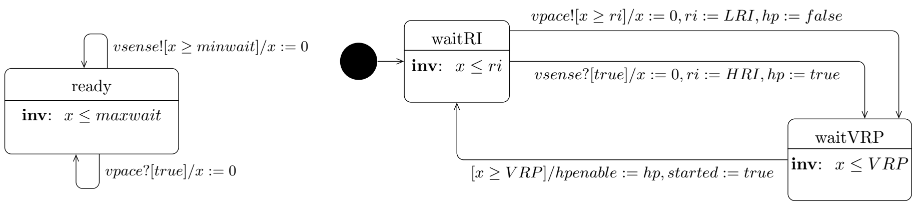
All modules in a composition must be *interface compatible*, which we now briefly define. If a variable v is declared in a composition of modules $m1$ and $m2$, then the types must be identical, and if one module specifies an **out** mode for v, then the other module must specify an **in** mode for v. Also, both modules may declare variable v as **share**, in which case they can both read and write to v. Only one module in a composition can declare v as **out** (that module is also responsible for initializing v).

Events can be spontaneous, fair (just or compassionate), or have lower and upper time bounds (illustrated in the pacemaker example). For example, event $env1.demonic[0, *]$ is a spontaneous event, i.e., it may be taken (or not) in any state. It has no fairness scheduling constraints. Its lower time bound is zero and it has no upper time bound; hence it has no timing constraints. Its action (or body) is the demonic assignment $y :: BIT$—meaning, that an element of BIT is assigned, non-deterministically to variable y.

The events in modules $m1$ and $m2$ are declared **just** to ensure progress; this achieves the goal that eventually $z == 1$ (i.e. $system \models \Diamond(z == 1)$). Justice and compassion have the standard meanings in [MP92]. An event (e.g. $m1.e1b$) has a guard (e.g. $y == 1 \&\& pc == 1$) and an action (e.g. the simultaneous assignment $z := 1, pc := 2$). Timed events may also start and stop local or global timers.

The model checker supports linear time temporal logic. We can thus directly check that the system $m1||m2$ satisfies the global property $\Diamond(z == 1)$, or we can follow a modular approach. Both approaches are shown in Fig. 1 (see the "asserts").

In the the modular approach, a temporal logic formula φ_1 is modularly valid for a module m_1 (written $m1 \models_m \varphi_1$) if the composition $m_1||m \models \varphi_1$, for every module m that is interface compatible with m_1. We check for modular validity of φ_1 by running module m_1 in parallel with an interface-compatible module representing an arbitrary environment that does demonic assignments on all the **in** and **share** variables of m_1 (see module $m1\_env$ in Fig. 1). For example, $\Diamond(z == 1)$ is not modularly valid for module m_1 because there is no guarantee that eventually $y == 1$ (thus allowing event $m1.e1b$ to be taken). However, $\Diamond\Box(y == 1) \rightarrow \Diamond(z == 1)$ is modularly valid, i.e., if we can rely on the environment to eventually establish $y == 1$ permanently, then module $m1$ guarantees to establish $\Diamond(z == 1)$. In the Ping-Pong example, we modelcheck the two modular specifications φ_1 and φ_2 (for modules $m1$ and $m2$, respectively):


$vsense![x \geq minwait]/x := 0$
ready
inv: $x \leq maxwait$
$vpace?[true]/x := 0$
waitRI
inv: $x \leq ri$
$vpace![x \geq ri]/x := 0, ri := LRI, hp := false$
$vsense?[true]/x := 0, ri := HRI, hp := true$
waitVRP
inv: $x \leq VRP$
$[x \geq VRP]/hpenable := hp, started := true$

(a) Model of Heart (b) Model of Ventricle Controller

Figure 2: Uppaal/timed-automata model for a pacemaker in VVI mode

$$\frac{m1 \models_m \varphi_1 \qquad m2 \models_m \varphi_2 \qquad \models \varphi_1 \land \varphi_2 \rightarrow \Diamond(z == 1)}{m1 || m2 \models \Diamond(z == 1)}$$

$$\begin{aligned} \varphi_1 &: \quad \Diamond\Box(x == 1) \land [\Diamond\Box(y == 1) \rightarrow \Diamond(z == 1)] \\ \varphi_2 &: \quad \Diamond\Box(x == 1) \rightarrow \Diamond\Box(y == 1) \end{aligned}$$

The global property $\Diamond(z == 1)$ follows from the temporal logic theorem $\varphi_1 \land \varphi_2 \rightarrow \Diamond(z == 1)$, and the modular checks. This approach was used to check the DRT example in [Ost99].

TTM/PAT provides static checking of the textual descriptions, simulations of the composed systems, and modelchecking of the LTL properties. Fig. 1 provides an example of a legal execution of the Ping-Pong system in the tool's simulator. A counter-example to an LTL property may be examined in the simulator, which is helpful in debugging modelling errors.

3 A Small Pacemaker Example

In this section, we use a small pacemaker example to illustrate the real-time features of TTMs. A cardiac pacemaker is an electronic device implanted into the body to regulate the heart beat by delivering electrical stimuli (called paces) over leads with electrodes that are in contact with the heart. The pacemaker may also detect natural cardiac stimulations, called senses.

Uppaal model. Uppaal is a state-of-the-art integrated environment for specification and verification of real-time systems modelled as networks of timed automata, extended with data types (such as bounded integers and arrays) [ABB+01]. An Uppaal model of the VVI mode of a pacemaker (developed in [JLS10]) is shown in Fig. 2 with a model of the heart (on the left) and with a timed automaton for the ventricle controller (on the right). Each timed automaton in Fig. 2 is a template that may be instantiated (e.g. $VC = Ventricle()$). A system may be represented a parallel composition of such instances.

The Uppaal heart model is ready to accept a pacing signal (over synchronous channel *vpace*) whenever it is sent by the controller, and it can choose to deliver a sensing signal (over channel *vsense*) at any time. Time bounds of events are expressed via state invariants and clock guards, i.e., transition guards referring to clocks. The invariant in the *ready* state is $x \leq maxwait$ where x is a local clock variable (and $maxwait = 1200ms$). The invariant imposes an upper time bound by which some transition out of the state must be taken. The guard $x \geq minwait$ (where $minwait = 200ms$) imposes a lower time bound on when a sense event is generated.

A pacemaker in the VVI mode operates in a timing cycle that begins with a paced or sensed ventricular event. The basis of the timing cycle is the lower rate interval (LRI=1000ms), which is the maximum amount of time between two consecutive events in the ventricle. If the LRI elapses and no sensed event occurred since the beginning of the cycle, a pace is delivered and the cycle is reset. On the other hand, if a heart beat is sensed, the cycle is reset without delivering a pace.

At the beginning of each cycle, there is a ventricular refractory period (VRP=400ms): chaotic electrical activity in the heart immediately following a heart beat that may lead to spurious detection of sensed events and can thus interfere with future pacing. For this reason, sensing is disabled during the VRP period. Once the VRP period is over, a sensed ventricular event inhibits the pacing and resets the LRI, starting the new timing cycle.

Hysteresis pacing can be enabled in the VVI mode, when the pacemaker will delay pacing beyond the LRI to give the heart a chance of resuming normal operation. In that case, the timing cycle is set to a larger value, namely the hysteresis rate interval (HRI=1200ms). It becomes enabled after a natural heart beat has been sensed. In [JLS10], hysteresis pacing is applied after a ventricular sense is received, and disabled after a pacing signal is sent.

The ventricle controller has two states: *waitRI* and $waitVRP$. The controller starts from *waitRI* and waits for a ventricular sensing or pacing event. If sensing does not occur before the *ri* (rate interval) period ends, the ventricle controller sends a pacing signal to the heart, resets clock x to zero, resets *ri* to LRI, and sets *hp* (hysteresis pulsing flag) to false, indicating that hysteresis pacing is not used in this case. On the other hand, if a ventricular sense event occurs before the current LRI cycle ends, *ri* is reset to HRI instead, allowing a longer period to elapse before pacing. In state $waitVRP$, the controller waits for a VRP period to elapse. It returns to the *waitRI* state after a VRP period. Other auxiliary controller variables *hpenable* and *started* are used in the specification.

The requirements are that the heart must pace somewhere between VRP and either LRI or HRI (depending on whether hysteresis pacing is off or on). We could translate the Uppaal model into a corresponding TTM. Instead, we present below a more realistic model than the Uppaal version in Fig. 2. We have developed an even more realistic model that separates monitored and controlled variables properly, and considers electrical noise that could be confused with heart beats during the refractory period. Space considerations prevent us from presenting it.

Issues in the Uppaal model. The heart in the Uppaal model in Fig. 2 is tightly coupled with the controller via synchronous channels. A simulation of the Uppaal model thus shows that the heart acting on its own deadlocks immediately due to this tight coupling. It is also unrealistic for the computer controller to respond precisely in sync with a sensing event. In the TTM model, we thus decouple the heart from the computer controller so that there are states intervening between the heart generating a sense signal and the computer responding to that signal (this could also be done in Uppaal). The TTM heart model can thus beat naturally (or not at all) and may also be paced (if a *pace* signal is received from the computer controller). The heart can thus be simulated on its own in the absence of a controller. It is possible for bad things to happen, non-deterministically, such as missing a beat (or not beating at all). This is done via the use of a *spontaneous* event, i.e., an event that has no upper time bound (i.e. deadline) and no fairness assumptions (i.e. justice or compassion). For example, see the natural heart beat event $hbn[\mathrm{VRP}, *]$ below.

Also, in the Uppaal model, the requirements are written in terms of the phenomena of the computer controller, not in terms of the phenomena of the heart. For example the Uppaal property $\mathrm{A}\Box(\neg VC.hpenable \rightarrow VC.x \leq LRI)$ states that the heart beats within LRI ticks of the clock in non-hysteresis pacing. That is, whenever the controller auxiliary variable $VC.hpeanble$ holds, then the computer clock must be at most LRI. Uppaal properties are written in a subset of branching time logic CTL (called TCTL). The sub-formulas can refer to states but not to the occurrence of events. On the other hand, in TTMs, the requirements will be written directly in terms of natural or paced events in the heart (in linear time temporal logic LTL). In Uppaal, lower (and upper) time bounds are written as clock guards (and invariants on states). Whereas, in TTMs, events are directly annotated with lower and upper time bounds (and the necessary clocks are implicitly included).

TTM model. Using the TTM syntax, we define constants and a timer t for the cardiac cycle as follows:

```
#define VRP 400;      | timers              | share initialization
#define LRI 1000;     |   t: 0..(HRI+1)     |   sense: BOOL = false
#define HRI 1200;     |         enabledinit |   pace: BOOL = false
                      | end                 | end
```

Timer t has a range from zero to HRI+1, and it is initially zero and enabled. When the timer reaches one beyond HRI + 1, it stops counting up and the monotonicity status $mono(t)$ predicate becomes false. This predicate hold so long as timer t is not stopped or restarted and that it is ticking in sync with a global *tick* of the clock (see Section 4). The *tick* transition is an implicit transition (i.e. it is not given by the text of the TTM) representing the ticking of a global clock. The tick transition increments timers and implicit clocks associated with events. We have also defined two shared variables *sense* and *pace*. The heart module template is described as follows:

```
module HEART
interface
    pace: share BOOL; sense: share BOOL
local
    ri : INT = HRI ; last_ri: INT = HRI; pc:
        INT = 0
events
    hbn[VRP, *] // natural heart beat
        when !pace && pc==0
        do sense := true, ri := HRI, last_ri:=
            ri, pc := 1
        end

hbp[0,0] // paced heart beat
  when pace && VRP <= t && pc==0
  do pace := false, ri := LRI , last_ri := ri,
      pc := 1
  end

new_cycle[0,0]
  when pc==1
  start t
  do pc := 0
  end
```

The interface of module template *HEART* declares the access to shared variables *sense* and *pace*. The local variable *ri* (rate interval) is either HRI or LRI depending on whether hysteresis pacing is enabled. Likewise *last_ri* records the last value of the rate interval. These two are auxiliary variables: they annotate the system state without affecting the behaviour (they are not used in event guards), and are used in the temporal logic specifications. The local variable *pc* (program counter) is used as a sequencing mechanism for events.

The heart module has a natural heartbeat (event *hbn*) and a paced heartbeat (event *hbp*). If there is a natural heart beat, then the *sense* flag is set, *ri* is set to HRI and the last rate interval is also recorded in *last_ri*. After the VRP period, it is also possible for a paced heart beat to be taken if the *pace* flag is set. Thus $pace \wedge \text{VRP} \leq t$ is part of the guard of the urgent event $hbp[0, 0]$. After either a natural or paced heart beat, the timer t is restarted by the *new_cycle* event and the cardiac cycle begins again.

A natural heart beat might occur at any time after the ventricle refractory delay VRP, or it might never occur. Thus the lower time bound of *hbn* is VRP and the upper time bound is $*$ (i.e ∞). If the upper time bound is $*$ then we have a spontaneous event (i.e. an event that is not urgent or forced to occur). We can thus accommodate a variety of fairness assumptions, including spontaneous events, just or compassionate events and real-time events that must occur between their lower and upper time bound. An urgent event $e[0, 0]$ is one that must occur before the next *tick* of the global clock (provided its guard continuously remains true).

The requirements can now be formulated using linear time temporal logic in terms of the phenomena of the environment (i.e. the heart) as follows:

- R1: $\Box\Diamond((H.hbn \vee H.hbp) \wedge VRP \leq t \leq HRI)$.[1] Infinitely often, a natural or paced heart beat occurs and they occur between VRP and HRI time units from each other.
- R2: $\Box(H.hbn \Rightarrow \text{VRP} \leq t \leq H.last\_ri)$. A natural heartbeat occurs only in the interval [VRP, *H.last_ri*] in the cardiac cycle. *H.last_ri* records the required rate interval in the heart for the last complete cycle, either

[1] *H.hbn* designates the event *hbn* in module instance *H*. The same works for *local* variables as well.

LRI or HRI, depending on whether hysteresis pacing has been properly enabled. Thus, $\Box(H.last\_ri = LRI \lor H.last\_ri = HRI)$ also holds.
- R3: $\Box(H.hbp \Rightarrow t = H.last\_ri)$. A paced heart beat occurs only if the timer t is at the relevant rate interval.

The ventricle controller will have to estimate $H.ri$ (which, as opposed to $H.last\_ri$, relates to the current cycle) in order to ensure that the heart paces according to this requirement.

In the Uppaal model there may be a concern that some event (either in the heart module or elsewhere) might illegally set the timer t to a value making the specification trivially true. Of course, in a small system, inspection of the timed automata might re-assure us that all is well. Nevertheless, it would be nice to be able to check that timers tick monotonically and uninterruptedly without any outside interference. Thus each TTM timer must be equipped with a corresponding monotonicity predicate $mono(t)$ that holds so long as timer t is not stopped or restarted (see section 4). We may thus check $(\Box\Diamond H.new\_cycle) \land \Box(H.new\_cycle \to t = 0)$ and $\Box(H.new\_cycle \Rightarrow mono(t)\,\mathcal{U}((H.hbn \lor H.hbp) \land (VRP \leq t \leq HRI))$, which guarantees that there is an appropriate heart beat in each cardiac cycle.

Uppaal does not provide a direct way of checking for zeno behaviour (which we must do given that we are using events with upper time bound 0). We can check that time always progresses with the LTL formula $\Box\Diamond tick$. The tick event is implicit. That is, it is automatically constructed by the tool with the precise semantics described in Section 4. The ability to refer to the occurrence of events in the TTM assertions makes it possible to specify the required behaviour more directly than in Uppaal. The next step is to devise a ventricle controller that will satisfy the requirements.

```
module VENTRICLE_CONTROLLER
interface
    pace : share BOOL; sense: share BOOL
local
    ri : INT = HRI; pc: INT = 0
events
    vpace[0,0]
        when pc==0 && !sense && t==ri
        do ri := LRI, pace := true, pc:= 1
        end

    vsense[0,0]
        when pc==0 && sense
        do ri := HRI, sense := false, pc :=1
        end

    compute_delay[1,1]
        when pc==1
        do pc:= 0
        end
end
```

The controller constructs its own estimate VC.ri of the heart's rate interval $H.ri$. We may now compose the heart together with the controller as follows:

```
instances
    H = HEART(share pace, share sense)
    VC = VENTRICLE_CONTROLLER (
           share pace, share sense)
end

composition
    System = H || VC
    Heart = H
end
```

The above syntax is accepted by the tool discussed in the sequel and all the requirements check in a few seconds.

Concrete syntax of event e:

```
event_id [l,u] just
    when grd
    start t_1, t_2
    stop t_3, t_4
    do v_1 := exp_1,
        if condition then v_2 := exp_2 else
            skip fi,
        v_3 :: 1..4
    end
```

Abstract syntax of the event e:

- $e.id \in \text{ID}$;
- $e.l \in \mathbb{N}$;
- $e.u \in \mathbb{N} \cup \{\infty\}$
- $e.fair \in \{\text{spontaneous, just, compassionate}\}$
- $e.grd \in \text{STATE} \times \text{TIME} \to \text{BOOL}$;
- $e.start \subseteq T$;
- $e.stop \subseteq T$;
- $e.action \in \text{STATE} \times \text{TIME} \leftrightarrow \text{STATE}$;

Figure 3: Concrete and Abstract syntax of TTM events

4 Operational Semantics

Sections 2 and 3 provide some examples of the new concrete textual syntax for TTMs. In this section, we provide a one-step operational semantics for a composition $m_1||m_2||\cdots||m_n$ of module instances. Following [Ost99] and using the mathematical conventions of Event-B [Abr10], let the abstract syntax of a TTM $\mathcal{M}$ be given by a 5-tuple, i.e., $\mathcal{M} = (V, s_0, T, t_0, E)$ where 1) V is a set variable identifiers whether local or declared in a module interface, 2) T is a set of timer identifiers and 3) E is a set of events, 4) $s_0 \in \text{STATE}$ is the TTM's initial variable assignment, with $\text{STATE} \triangleq V \to \text{VALUE}$ and 5) $t_0 \in \text{TIME}$ is the TTM's initial timer assignment, with $\text{TIME} \triangleq T \to \mathbb{N}$. Set E contain events that may change the state. The abstract syntax of each event $e \in E$ is specified by a record as shown on the right of the Fig. 3 (where we use the dot notation "." to access the records' fields).

The event guard grd is any boolean expression in V and T. In the above example, $V = \{v_1, v_2, v_3, \cdots\}$ and $T = \{t_1, t_2, t_3, t_4, \cdots\}$. The function $boundt \in T \to \mathbb{N}$ and $type \in T \to \mathbb{P}(\mathbb{N})$ provide, respectively, the upper bound and type of each timer. For example, if timer t_1 is declared in the TTM as $t_1 : 0..5$, then $boundt(t_1) = 5$ and $type(t_1) = \{0..6\}$. As will be detailed below, timers count up to one beyond the specified bound at which point they remain fixed until they are restarted.

The event must be taken between its lower time bound l and upper time bound u, provided that its guard remains true. The event action involves simultaneous assignments to $v_1, v_2, \cdots$. The notation $v_3 :: 1..4$ is an example of a demonic assignment in which v_3 takes any value from 1 to 4. All the assignments in the event action are applied simultaneously in one step.

In an assignment $y := exp$, the expression on the right may use primed (e.g. x') and unprimed (e.g. x) state variables as well as the initial value of timers. A variable with a prime refers to the variable's value in the next state and a variable without prime refers to its value in the current state. The use of primed variables in expressions allows for simpler and more expressive descriptions of state changes. The state changes effected by an event e is described in the abstract syntax by a before-after predicate $e.action$. The concrete syntax also allows for assignments to be embedded in (possibly nested) conditional statements.[2]

In order to build a TTM tool with the PAT framework, we provide a one-step operational semantics in *labelled transition systems* (LTS).

Definition: (Labelled Transition System (LTS)) An LTS is a 4-tuple $\mathcal{L} = (\Pi, \pi_0, \mathbf{T}, \rightarrow)$ where 1) Π is a set of system configurations; 2) $\pi_0 \in \Pi$ is an initial configuration; 3) $\mathbf{T}$ is a set of transitions names; and 4) $\rightarrow \subseteq \Pi \times \mathbf{T} \times \Pi$ is a transition relation.

We now describe the LTS semantics of TTMs. Let $E_{id} \triangleq \{e \in E \bullet e.id\}$ be the set of event names (identifiers). A configuration $\pi \in \Pi$ is defined by a 6-tuple (s, t, m, c, x, p), where:

- $s \in$ STATE is a value assignment for all the variables of the system. The state can be read and changed by any transition corresponding to an event in E.
- $t \in$ TIME is a value assignment for the timers of the system. Events (and hence their corresponding transitions) may only start, stop and read timers. As will be discussed below, we introduce a special transition, called *tick*, which also changes the timers. Timers t_i that are stopped have values $boundt(t_i) + 1$.
- $m \in T \rightarrow$BOOL records the status of monotonicity of each timer. Suppose event e_1 in a TTM starts t_1. In LTL we might write $\Box(\ e_1 \wedge t_1 = 0 \ \rightarrow\ \Diamond(q \wedge t_1 \leq 4)\ )$ (note that $t_1 = 0$ is redundant) to specify that q becomes true within 4 time units of event e_1 occurring. However, other events might stop or restart t_1 before q is satisfied hence breaking the synchronicity between t_1 and a global clock.[3] Instead, we express the intended property as $\Box(\ e_1 \wedge t_1 = 0 \ \Rightarrow\ m(t_1)\, \mathcal{U}\, (q \wedge t_1 \leq 4)\ )$. The expression $m(t_1)$ (standing for monotonicity of t_1) holds in any state where t_1 is not stopped or being reset. We explain monotonicity further below.
- $c \in E_{id} \rightarrow \mathbb{N} \cup \{-1\}$ is a value assignment for a clock implicitly associated with each event. These clocks are used to decide whether an event has been enabled for long enough and whether it is urgent. An event $e \in E$ is enabled when its clock's value is between the event's lower time bound ($e.l$) and its upper time bound ($e.u$). Furthermore, the type of $c(e.id)$ is $\{-1, 0, ...e.u\}$. When an event's clock is disabled, as opposed to the convention used with timers, the clock's value is -1.
- $x \in E_{id} \cup \{\perp\}$ is used as a sequencing mechanism to ensure that each transition e is immediately preceded by an $e\#$ transition whose only function

[2]With all the complexity of structures allowed by the syntax of actions, sequential composition is not allowed. This is in an effort to make actions into specifications rather than implementations. This would allow us to generalize TTMs to allow an Event-B style of symbolic reasoning.

[3]Suppose that event e_2 also starts t_1, that e_3 establishes q and that the events occur in the following order: $\pi_0 \overset{e_1}{\rightarrow} \underset{t_1=0}{\pi_1} \overset{tick^3}{\rightarrow} \underset{t_1=3}{\pi_4} \overset{e_2}{\rightarrow} \underset{t_1=0}{\pi_5} \overset{tick^2}{\rightarrow} \underset{t_1=2}{\pi_7} \overset{e_3}{\rightarrow} \underset{t_1=2\ \wedge\ q}{\pi_8} \cdots$. This execution satisfies the first LTL formula but does not satisfy the intended specification: when q becomes true, $t_1 = 2$ but it is 5 ticks away from the last occurrence of e_1.

is to update the monotonicity record m. For example, in the following execution $\cdots \xrightarrow{e_1} \underset{x=\bot}{\pi_1} \xrightarrow{e_2\#} \underset{x=e_2}{\pi_2} \xrightarrow{e_2} \underset{x=\bot}{\pi_3} \rightarrow \cdots$, suppose in π_1 the value of timer t_2 is 3 and that e_2 restarts t_2. Then, in π_2, we have $x = e_2 \ \wedge \ t_2 = 3 \ \wedge \ m(t_2) = \text{false}$. In π_3, we have $x = \bot \ \wedge \ t_2 = 0 \ \wedge \ m(t_2) = \text{true}$. In order to record the breaking of monotonicity, the $e_2\#$ transition sets $m(t_2)$ to false, which gets set back to true in the following configuration. The precise effect of these transitions will be described below.

- $p \in E_{id} \cup \{tick, \bot\}$ holds the name of the last event to be taken at each configuration. It is $\bot$ in the initial configuration as no event has yet occurred. It allows us to refer to events in LTL formula in order to state that they have just occurred. For instance, in the formula above, $(s, t, m, p) \models e_1 \wedge t_1 = 0$ (which reads: the configuration *satisfies* the formula) evaluates to $p = e_1 \wedge t(t_1) = 0$.

Given a timed transition model $\mathcal{M}$, the transitions of its corresponding LTS is given as $\mathbf{T} = E_{id} \cup E\# \cup \{tick\}$. As explained above, for each event $e \in E$, we introduce a monotonicity breaking transition $e.id\#$. We thus define $E\# \triangleq \{e \in E \bullet e.id\#\}$. The *tick* transition represents one tick of a global clock. Explicit timers and event lower and upper time bounds are described with respect to this tick transition. We define the enabling condition of event $e \in E$ as $e.en \triangleq e.grd \ \wedge \ e.l \le e.c \le e.u$, where $e.c$ evaluates to $c(e.id)$ in a configuration whose clock component is c. Thus an event is enabled in a configuration that satisfies its guard and where the event's implicit clock is between its lower and upper time bound.

The initial configuration is defined as $\pi_0 = (s_0, t_0, m_0, c_0, \bot, \bot)$, where s_0 and t_0 come from the abstract syntax of the TTM. m_0 and c_0 are given by:

$$m_0(t_i) \ \equiv \ t_0(t_i) = 0$$

$$c_0(e_i.id) \ = \ \begin{cases} 0 & (s_0, t_0) \models e_i.grd \\ -1 & (s_0, t_0) \not\models e_i.grd \end{cases}$$

for each $t_i \in T$ and $e_i \in E$. It is implicit in the above formula that $m_0(t_i)$ depends only on whether or not t_i is initially enabled (specified by the keywords **enabledinit** and **disabledinit**). If **enabledinit**, $t_0(t_i) = 0$; otherwise, if **disabledinit**, $t_0(t_i) = boundt(t_i) + 1$.

An execution σ of the LTS is an infinite sequence, alternating between configurations and transitions, written as $\pi_0 \xrightarrow{\tau_1} \pi_1 \xrightarrow{\tau_2} \pi_2 \rightarrow \cdots$ where $\tau_i \in \mathbf{T}$ and $\pi_i \in \Pi$.

Below, we provide constraints on each one-step relation ($\pi \xrightarrow{e} \pi'$) in an execution. If an execution σ satisfies all these constraints then we call σ a *legal* execution. We let $\Sigma_{\mathcal{L}}$ denote the set of all legal executions of the labelled transition system $\mathcal{L}$. The set $\Sigma_{\mathcal{L}}$ provides a precise and complete definition of the behaviour of $\mathcal{L}$.

If a state-formula q holds in a configuration π, then we write $\pi \vDash q$. In some formulas, such as guards, all the components of a configuration are not necessary. We express this by dropping some components of the configuration on the left of the double turnstile ($\models$), as in $(s_0, t_0) \models e.grd$. Given a temporal logic property φ and an LTS $\mathcal{L}$, we write $\mathcal{L} \vDash \varphi$ iff $\forall \sigma \in \Sigma_{\mathcal{L}} \bullet \sigma \vDash \varphi$. The three possible transition steps are:

$$(s, t, m, c, \bot, p) \overset{e\#}{\to} (s, t, m', c, e, p) \tag{4.1}$$

$$(s, t, m, c, e, p) \overset{e}{\to} (s', t', m', c', \bot, e) \tag{4.2}$$

$$(s, t, m, c, \bot, p) \overset{tick}{\to} (s, t', m', c', \bot, tick) \tag{4.3}$$

Each of the above transitions have side conditions which we now enumerate.

4.1 Taking $e\#$

The monotonicity breaking transition $e\#$, specified in (4.1), is taken only if $(s, t, c) \vDash e.en$ and the x-component of the configuration is $\bot$. For each $t \in T$, $m'(t) \equiv t \notin e.start \wedge m(t)$. This ensures that, for timer t, just before it is (re)started, $m(t) =$ false. It is set back to true by the immediately following event, e, and it remains true as long as t is not restarted and has not reached its upper bound. Transition $e\#$ modifies only m and x in the configuration, and thus maintains the truth of $(s, t, c) \vDash e.en$.

4.2 Taking e

The transition e, specified in (4.2), is taken only if $(s, t, c) \vDash e.en$ and the x-component of the configuration is e. The component s' of the next configuration in an execution is determined nondeterministically by $e.action$, which is a relation rather than a function. This means that any next configuration that satisfies the relation can be part of a valid execution, i.e., s' is only constrained by $(s, t, s') \in e.action$. The other components are constrained deterministically. The following function tables specify the updates to m, t and c upon occurrence of transition e:

For each timer $t_i \in T$		$m'(t_i)$	$t'(t_i)$
$t_i \in e.start$	$t_i \in e.stop$	*impossible*	
	$t_i \notin e.stop$	true	0
$t_i \notin e.start$	$t_i \in e.stop$	false	$boundt(t_i) + 1$
	$t_i \notin e.stop$	$m(t_i)$	$t(t_i)$

For each event $e_i \in E$		$c'(e_i.id)$
$(s', t') \not\models e_i.grd$		-1
$(s', t') \models e_i.grd$	$(s, t) \models e_i.grd \ \wedge \ \neg e_i = e$	$c(e_i.id)$
	$(s, t) \not\models e_i.grd \ \vee \ e_i = e$	0

In the above, we start and stop the implicit clock of e_i as a consequence of executing e, according to whether $e_i.grd$ becomes true, is false (i.e. becomes or remains false) or remains true. Since event e_i becomes enabled $e_i.l$ units after its guard becomes true, this allows us to know when to consider e_i as enabled, i.e., ready to be taken. As a special case, the implicit clock of event e (under consideration) is restarted when $e.grd$ remains true.

4.3 Taking *tick*

The tick transition, specified in (4.3), is taken only if $\forall e \in E \bullet c(e.id) < e.u$ and the x-component of the configuration is $\perp$ (thus preventing *tick* from intervening between any $e\#$ and e pair). For any timer $t_i \in T$, the updates to t', m' and c' are:

$$
\begin{aligned}
t'(t_i) &= (t(t_i) \downarrow boundt(t_i)) + 1 \\
m'(t_i) &\equiv \neg\ (t(t_i) = boundt(t_i){+}1)
\end{aligned}
$$

For each event $e \in E$		$c'(e.id)$
$(s',t') \not\models e.grd$		-1
$(s',t') \models e.grd$	$(s,t) \not\models e.grd$	0
	$(s,t) \models e.grd$	$c(e.id) + 1$

Thus, *tick* increments timers and implicit clocks to their upper bounds. Transition *tick* also marks timers as non-monotonic when they reach their upper bound and reset clocks when the corresponding events are disabled.

4.4 Scheduling

So far, we constrained executions so that the state can change only in controlled ways but it is still possible that a given execution would make no progress. To make progress, we need to assume fairness. Up to now, we have kept the fairness and real-time constraints decoupled. In the current implementation of TTM/PAT, the possible scheduling constraints on TTM events are restricted to the following four:

- Spontaneous event. Even when it is enabled, the event might never be taken. This is assumed when no fairness keyword is given and the upper time bound is * or unspecified.
- Just event scheduling (also known as weak fairness). For any execution $\sigma \in \Sigma_{\mathcal{L}}$, if the event eventually becomes continuously enabled, it has to occur infinitely many times, that is $\sigma \models \Diamond\Box e.en \rightarrow \Box\Diamond e$. This is assumed when the keyword **just** is given next to the event and the upper time bound is * or unspecified. We use $e.en$ and not $e.grd$ in the fairness formula as the event can only be taken $e.l$ units after its guard became true.
- Compassionate event scheduling (also known as strong fairness). For any execution $\sigma \in \Sigma_{\mathcal{L}}$, if the event becomes enabled infinitely many times, it has to occur infinitely many times, that is $\sigma \models \Box\Diamond e.en \rightarrow \Box\Diamond e$. This is assumed when the keyword **compassionate** is given next to the event and the upper time bound is * or unspecified.

- Real-time event scheduling. The (finite) upper time bound (u) of the event e is taken as a deadline: if the event's guard is true for u units of time, it has to occur within u units of after the guard becomes true or after the last occurrence of e. To achieve this effect, the event e is treated as just. Since *tick* won't occur as long as e is urgent (i.e. $e.c = e.u$), transition e will be forced to occur (unless some other event occurs and disables it).

To accurately model time, the *tick* transition is treated as compassionate in the LTS. This ensures that time progresses except in cases of zeno-behaviors (discussed below).

Spontaneous events cannot be used to establish liveness properties. Justice and compassion are strong enough assumptions to establish liveness properties but not real-time properties. Finally, real-time events can establish both liveness and real-time properties.

The above semantics allows for zeno behaviours, i.e., it is possible to have executions in which the tick transition does not occur infinitely often (at some point, time stops). Zeno behaviour occurs only where there are loops involving events with zero upper time bound (i.e. e[0,0]). We could ban e[0,0] events altogether, but that would eliminate behaviours that are feasible and useful, e.g., where we describe a finite sequence of immediately urgent events (not in a loop). We can check that the system is non-zeno by checking that the system satisfies $\Box\Diamond tick$.

The abstract TTM semantics provided above can be implemented efficiently. For example, in the abstract semantics every event e is preceded by a breaker of monotonicity $e\#$. Most of the $e\#$ events do not change the configuration monotonicity component m and can thus be safely omitted from the reachability graph thereby shrinking it.

5 Performance

We have implemented the TTM textual language and semantics as a plug-in of the PAT toolset. In this section we report on two performance experiments on the plug-in. All experiments were conducted on a 64-bit Windows 7 PC with Intel(R) Core(TM) i7 CPU 860 @ 2.80 GHz (16.0 GB RAM). The tool (and experimental data) may be downloaded at `https://wiki.cse.yorku.ca/project/ttm/`.

In section 5.1, we use the DRT (delayed trip reactor) nuclear shutdown example for the performance comparison. In [LPZ06], the DRT was manually encoded and checked in the Uppaal and SAL model-checkers. In that experiment, Uppaal performed better than SAL. We compare the performance of the TTM/PAT plug-in with these manual encodings and find that TTM/PAT performs significantly better. As reported in [LPZ06], the need to manually encode the TTM was a time consuming task. Thus the TTM plug-in has two advantages. The TTM encoding to the modelchecker is done automatically and its performance is significantly better.

Assertion	TCTL of UPPAAL	LTL of TTM/PAT
Invariantly p	$S \models A\Box\ p$	$S \models \Box\ p$
Eventually p	$S \models A\Diamond\ p$	$S \models \Diamond\ p$
Whenever p, eventually q	$S \models p\ \longrightarrow\ q$	$S \models \Box\ (p \Rightarrow (\Diamond\ q))$
Infinitely often p	$S \models true\ \longrightarrow\ p$	$S \models \Box\Diamond\ p$
Referring to transition state	$M.state$	$pc = state$
Non-zenoness	$\times$	$S \models \Box\Diamond\ tick$
p until q, assuming eventually q	$\times$	$S \models p\,\mathcal{U}\,q$
p until q	$\times$	$S \models q\,\mathcal{R}\,p$
Nesting of temporal operators	$\times$	e.g., $\Box\ (\Diamond\ p \Rightarrow (p\mathcal{U}q))$
Referring to occurrences of event e	$\times$	e
Timer t has increased monotonically	$\times$	$mono\ (t)$
Eventually henceforth p	$\times$	$S \models \Diamond\Box\ p$
S possibly maintains p	$S \models E\Box\ p$	inverse of $S \models \Diamond\ (\neg p)$
S possibly reaches p	$S \models E\Diamond\ p$	S **reaches** p
Nesting of path quantifiers	$\times$	$\times$
$\forall\Diamond\ \forall\Box\ p$	$\times$	$\times$

Table 1: TTM vs. UPPAAL: Language of Assertions

We also use Fischer's mutual exclusion algorithm to compare the native verification performance of the TTM/PAT plug-in versus Uppaal and the RTS/PAT plug-in (see Section 5.2 and Table 3). The RTS plug-in has an explicit clock digitization mode and an efficient clock zone mode. The performance of the TTM plug-in is within a linear factor of the RTS digital mode. However, as expected, the Uppaal and RTS clock zone mode performed significantly better than the TTM plug-in. We are currently examining how to extend the TTM plug-in with these more efficient modes. In each case, the extra performance comes at some expressive cost. For example, in TTMs we have just and fair events not supported by Uppaal. Uppaal's subset of the CTL specification language is less expressive than the LTL supported by TTMs.

Table 1 shows that Uppaal's TCTL specification language is less expressive than that of TTM/PAT. There are temporal properties such as $\Diamond\Box p$ that can be specified and verified in the TTM plug-in but not in Uppaal. Also, non-zenoness and timer monotinicty can be checked directly in the specification language.

5.1 Delayed Reactor Trip System

The DRT (delayed trip reactor) shutdown system is illustrated in Fig. 4. The old implementation of the DRT used timers, comparators and logic gates as shown in the figure. 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. A high-level state/event description ($SPEC$) of the code that replaces the analogue system is shown to the right of Fig. 4 ([LPZ06]). When the reactor pressure and power exceed acceptable safety limits in a specified way, we want the DRT control system to

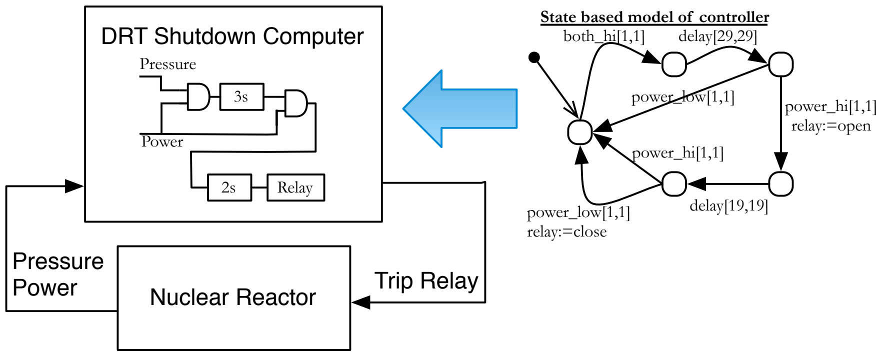


Figure 4: DRT System: Context Diagram and Transition Diagram of Controller

shut down the reactor. Otherwise, we want the control system to be reset to its initial monitoring state.

In [LPZ06], the SPEC level TTM description of the controller is refined into a lower level PROG description that is closer to code used in a cyclic executive. Translations to PVS are used to show that PROG refines SPEC. The reactor itself is represented by a TTM that can change the power and pressure levels arbitrarily every .1 seconds (1 tick of the clock), by using a demonic assignment setting them to either low or high. The system thus consists of the controller (either SPEC or PROG) executing in parallel with plant (the reactor). The two essential properties that the system must satisfy are:

Response Formula F_{res}: Henceforth, if Power and Pressure simultaneously exceed their threshold values for at least 2 clock ticks, and 30 ticks later Power exceeds its threshold for another 2 ticks, then within 30 to 32 ticks, open the reactor relay for at least 20 ticks.

Recovery Formula F_{rec}: Henceforth, if the relay is open for 20 ticks, and after the 20th tick the power is low for at least 2 ticks, then the relay is closed before the 22nd tick.

With the help of an observer and timers, the response formula F_{res} is represented in LTL by a liveness property $\Box p \rightarrow \Diamond q$ where p and q use timers to capture the timed response (see [LPZ06] for the details). Likewise, the recovery formula F_{rec} can be reduced to a safety property $\Box\neg(T_w = 2 \wedge \mathit{relay} = \mathit{open})$ where T_w is a timer describing a state in which the power has returned to normal for 2 ticks of the clock, but the relay is still open.

Both *SPEC* and *PROG* did not satisfy F_{res} due to an error in the observer. Thus, verification of F_{res} should produce counter-examples in any modelchecker. Also, it was discovered that there was an error in the controller (in both *SPEC* and *PROG*) as the recovery property was not satisfied. The revised and corrected descriptions of the controller are $SPEC_r$ and $PROG_r$, respectively.

To generate large reachability graphs, multiple controllers were run in parallel with each other. In one such response example, the number of states checked

Property	Controller Model	TTM: $\Box\Diamond$ *tick*	Result	TTM/PAT	Uppaal	SAL
F_{res}: *System Response*	$SPEC$	11	×	11	13	25
	$PROG$	31	×	32	24	407
	$SPEC_r$	5	×	3	12	15
	$PROG_r$	14	×	9	21	330
F_{ires}: *Initialized System Response*	$SPEC$	.5	✓	.4	.9	11
	$PROG$	1	✓	1	1	20
	$SPEC_r$	.3	✓	.2	.4	7
	$PROG_r$	.8	✓	.6	1	13
	$SPEC_{r1}\|\|SPEC_{r2}$	16	✓	11	62	235
	$PROG_{r1}\|\|PROG_{r2}$	109	✓	70	76	>1h
F_{rec}: *System Recovery*	$SPEC$	.3	×	.08	.1	6
	$PROG$	.8	×	.2	.3	7
	$SPEC_r$	.1	✓	.07	.2	4
	$PROG_r$	.3	✓	.07	.6	5
	$SPEC_{r1}\|\|SPEC_{r2}$	22	×	.06	145	18
	$PROG_{r1}\|\|PROG_{r2}$	142	×	.1	11	>1h

Table 2: TTM/PAT vs. Uppaalvs. SAL: Delayed Reactor Trip System

was 421,442 states and 821,121 transitions (in 70 seconds). These systems and their LTL specifications (some valid and some invalid) provide a rich set of examples to test the performance of the various model-checkers. In [LPZ06], the TTMs were manually encoded into the Uppaal and SAL model-checkers. The authors of [LPZ06] show that, in general, Uppaal performed better than SAL given its real-time features.

The encoding of TTMs into Uppaal and SAL is itself a time-consuming process, as it has to be done manually. This is where the new TTM/PAT tool is useful as the encoding is automatic. What about performance? In Table 2, we compare TTM/PAT to the encodings in SAL and Uppaal for response and recovery, and for the various versions of the controller. The 4th column labelled "Result" has a checkmark where the LTL property is valid; else, the model-checker produces a counter-example.

In general, TTM/PAT significantly out-performs both SAL and Uppaal. There is only one exception in the second row for F_{res}. TTM/PAT finds the formula invalid in 9 seconds versus 18 seconds for Uppaal (this is not shown in the table). However, it takes TTM/PAT 32 seconds to find the counter-example versus 24 seconds for Uppaal.

5.2 Fischer's Mutual Exclusion Algorithm

In [SLD+13], a comparison was performed between RTS (a PAT plugin) and Uppaal using the Fischer mutual exclusion algorithm. In this section, we compare the performance of the TTM plugin to RTS and Uppaal on the Fischer example. This is only one sample point and many more examples would be needed for a proper comparison of the three tools. See appendix A for the experiment's site where the Uppaal and RTS details are made available.

The PAT toolset provides both digitization (using BDDs) and more efficient symbolic clock zone algorithms (using difference bound matrices) for checking RTS models. The TTM plugin uses the explicit state algorithms of the toolset. We thus expect that the TTM performance will be similar to digitization but slower than clock zones. To obtain better performance, we have the choice of either using Uppaal or the difference bound matrix techniques. The experiment in this section was performed to help us with that choice.

We build the TTM model (see Appendix A) to be as close as possible to the RTS model at the experiment's site. The system under verification consists of n copies of processes running in parallel, using the indexed parallel composition syntax:

composition *fischer* = || *i*: 1..*n* @ *PROCESS*(**share** *x*, **share** *c*, **in** *i*) **end**

All processes, each identified by an integer i, share two global variables: mutex x and counter c. The mutex may hold the identifier of any process or -1 to denote system idleness. The counter records the number of processes that have entered their critical section. Process instances execute as follows: 1) await the system being idle; 2) signal that it intends to enter its critical section (CS) by updating x to its identifier i within δ ticks of the clock; 3) delay for exactly ϵ ticks of the clock. 1,2 and 3 are repeated as long as the process gets overtaken. 4) enter its CS, incrementing counter c; and 5) exit from its its CS, decrementing counter c, if applicable. We name the state between 2) and 3) *request*, that between 3) and 4) *wait* and that between 4) and 5) *cs*. The following properties are commonly relevant for mutual exclusion algorithms.

P1: Mutual Exclusion. At most one process is in its CS at any time: $\Box\ (c \leq 1)$.

P2: Liveness A process successfully signalling its intent of entering the CS will get to wait in the appropriate state for access: $\Box(request \rightarrow \Diamond wait)$.

P3: Starvation freedom A process successfully signalling its intent of entering the CS eventually does enter its CS: $\Box(request \rightarrow \Diamond cs)$.

When checking liveness properties in RTS, we can specify whether to check: all, event level weak and strong fairness, process level weak and strong fairness and global fairness. Only global fairness (a very strong assumption) successfully established the starvation freedom property. The same situation applies to the TTM model. Uppaal does not have the facilities to check properties under any fairness assumptions. TTMs provide the option to choose fairness on an event by event basis. In Table 3, we compare the performance of the three tools without any fairness assumptions. TTM/PAT also provides the ability to check for non-zeno behavior, a facility not provided by the other tools. Results for this property are also reported by the table.

Our experiment shows that, in determining that properties **P1** and **P2** are valid, the clock zone mode of RTS is faster than Uppaal (see Table 3). The speed of TTM/PAT is within a factor between 3 and 4 of the digitization mode of RTS. TTM/PAT is almost as fast as Uppaal in producing counter examples

Property	Result	n	Uppaal	PAT/RTS		TTM/PAT
				clock zone	digitization	
non-zenoness: $\Box\Diamond\ tick$	✓	4	not directly supported			.5
		5				4
		6				31
		7				230
		8				>1h
P1 mutual exclusion: $\Box\ (c \leq 1)$	✓	4	.04	.12	.08	.26
		5	.1	.2	.4	1.9
		6	.8	2	3	14
		7	14	21	28	104
		8	563	250	244	768
		9	>1h	2918	>1h	>1h
P2 liveness: $\Box(request \Rightarrow \Diamond wait)$	✓	4	.06	.07	.1	.3
		5	.2	.3	.8	3
		6	4	3	6	24
		7	181	29	58	177
		8	>1h	307	>1h	>1h
P3 liveness: $\Box(request \Rightarrow \Diamond cs)$	×	4	.2	.06	.09	.01
		5	.2	.3	.9	.01
		6	.3	3	19	.03
		7	.2	70	942	.04
		8	.2	2277	>1h	.03

Table 3: TTM/PAT vs. RTS/PAT vs. Uppaal: Fischer's Algorithm

for property **P3**. The results in Table 3 suggest that both the techniques used in clock zones of RTS and those of Uppaal would provide good enhancements for the verification of TTMs. In both cases, this would come at the cost of some expressivity. For the comparison with Uppaal see Table 1. RTS clock zones do not cover the use of "P within $[l, u]$", which forces process P to terminate between l and u units of time; the lower time bound is the problematic part to implement.

6 Conclusion

We have proposed a convenient and expressive textual syntax for TTMs. We have also provided a corresponding operational semantics that is used to build tool support as a plug-in of the PAT toolset. The TTM assertion language, LTL, allows references to event occurrences, including clock ticks (thus allowing for a check that the behaviour is non-zeno). Tool support includes an editor with static type checking, a graphical simulator and a LTL verifier. The TTM tool performs significantly better on a nuclear shutdown system than the manually encoded version in Uppaal and SAL.

We can improve the performance of the tool either by using the clock zone algorithms of RTS or the timed automata of Uppaal. In either case, this would come at the cost of expressiveness. For the comparison with Uppaal see Table 1.

In RTS, the construct "P within $[l, u]$", which forces process P to terminate between l and u units of time, is not supported by the clock zone algorithms; the lower time bound is the problematic part to implement. In future work, we intend to explore the clock zone algorithms of RTS as these are already available in the PAT toolset.

The TTM/PAT tool already supports an assume-guarantee style of compositional reasoning (see Section 2). The use of linear time temporal logic better supports compositional reasoning than branching time logic [Var01]. Event actions are specified as before-after predicates allowing us to enhance compositional reasoning using methods developed in UNITY [CM89].

Acknowledgements: Our sincere thanks to Prof. Yang Liu (School of Computer Engineering, Nanyang Technological University) and to Prof. Jun Sun (Singapore University of Technology and Design) for their development of the PAT toolset and for their help in developing the TTM plug-in.

References

[ABB$^+$01] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, W. Yi. Uppaal - Now, Next, and Future. In *MOVEP*. LNCS 2067, pp. 99–124. 2001.

[Abr10] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[CM89] K. M. Chandy, J. Misra. *Parallel program design - a foundation.* Addison-Wesley, 1989.

[JLS10] E. Jee, I. Lee, O. Sokolsky. Assurance Cases in Model-Driven Development of the Pacemaker Software. In Margaria and Steffen (eds.), *ISoLA 2010: Part II.* Volume LNCS 6416. 2010.

[LPZ06] M. Lawford, V. Pantelic, H. Zhang. Towards Integrated Verification of Timed Transition Models. *Fundamenta Informaticae* 70(1,2):75–110, 2006.

[MP92] Z. Manna, A. Pnueli. *The Temporal Logic of ractive and Concurrent Systems: Specification.* Springer–Verlag, 1992.

[Ost99] J. S. Ostroff. Composition and Refinement of Discrete Real-Time Systems. *ACM Transaction on Software Engineering Methodology* 8(1):1–48, 1999.

[SLD$^+$13] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, E. André. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transaction on Software Engineering Methodology* 22(1):3:1–3:29, 2013.

[Var01] M. Y. Vardi. Branching vs. Linear Time: Final Showdown. In *TACAS 2001.* Volume LNCS 2031, Springer-Verlag, pp. 1–22. 2001.

```
// Fischer's mutual exclusion algorithm

#define idle −1;
#define n 9;
#define delta 3;
#define epsilon 4;

share initialization
  x: INT = idle //semaphore
  c: INT = 0
end

module PROCESS
  interface
    x: share INT
    c: share INT//no. of processes
                //inside critical region
    i: in INT
  local
    pc: INT = 0
  events

    await[0, 0]
    when pc==0 && x==idle
    do pc := 1
    end

    update[0, delta]
    when pc == 1
    do x := i, pc := 2
    end

    delay[epsilon, epsilon]
    when pc==2
    do if(x == i) then pc := 3
                  else pc := 0
       fi
    end

    enter
    when pc==3 && x==i
    do c++, pc:=4
    end

    exit
    when pc==4
    do c−−, pc:=0, x:=idle
    end
end

composition
  fischer = ||i:1..n @
         PROCESS(share x, share c, in i)
end

#define cgreater1 c>1;
#define pc1 fischer[0].pc==1;
#define pc2 fischer[0].pc==2;
#define pc4 fischer[0].pc==4;

#assert fischer |= []!cgreater1;//mutex
#assert fischer |= [](pc1 −> <> pc2);
#assert fischer |= [](pc1 −> <> pc4);
#assert fischer |= []<>tick;
```

Figure 5: TTM model of the Fischer mutual exclusion algorithm

A Appendix

Fig. 5 provides the TTM model of the Fischer algorithm. The TTM model corresponds to the RTS and the Uppaal models provided at the experiment site at `http://www.comp.nus.edu.sg/~pat/rts/`. The urgent *await*$[0, 0]$ event in Fig. 5 matches an equivalent urgent transition in the RTS model. The Uppaal model provided at the above site did not make that event urgent. However, we used both the RTS and the Uppaal models as they appear at the experimental site.