Verifying Stateful Timed CSP using Implicit Clocks and Zone Abstraction

Jun Sun, Yang Liu, Jin Song Dong and Xian Zhang

School of Computing,
National University of Singapore
{sunj,liuyang,dongjs,zhangxi5}@comp.nus.edu.sg

Abstract. In this work, we study model checking of compositional real-time systems. A system is modeled using mutable data variables as well as a compositional timed process. Instead of explicitly manipulating clock variables, a number of compositional timed behavioral patterns are used to capture quantitative timing requirements, e.g. delay, timeout, deadline, timed interrupt, etc. A fully automated abstraction technique is developed to build an abstract finite state machine from the model. The idea is to dynamically create/delete clocks, and maintain/solve a constraint on the clocks. The abstract machine weakly bi-simulates the model and, therefore, LTL model checking or trace-refinement checking are sound and complete. We enhance our home-grown PAT model checker with the technique and show its usability via the verification of benchmark systems.

1 Introduction

Specification and verification of real-time systems are important research topics which have practical implications. During the last decade or so, a popular approach for specifying real-time systems is based on the notation Timed Automata [1, 23]. Timed Automata are powerful in designing real-time models with explicit clock variables. Real-time constraints are captured by explicitly setting/reseting clock variables. A number of automatic verification support for Timed Automata have proven to be successful (e.g. UPPAAL [20], KRONOS [4] and RED [36]).

Models based on Timed Automata often adapt a simple structure, e.g. a network of Timed Automata with no hierarchy [20]. The benefit is that efficient model checking is made feasible. Nonetheless, designing and verifying compositional real-time systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. High-level requirements for real-time systems are often stated in terms of *deadline*, *time out*, and *timed interrupt* [18, 11, 22]. In industrial case studies of real-time system verification, system requirements are often structured into phases, which are then composed sequentially, in parallel and alternatively [14, 19]. Unlike statecharts (with clocks) or timed process algebras, Timed Automata lack high-level compositional patterns for hierarchical design. As a result, users often need to manually cast those terms into a set of clock variables with carefully calculated clock constraints. The process is tedious and error-prone.

Contributions We investigate an alternative approach for modeling and verifying compositional real-time systems. In this work, a system is modeled using a compositional timed process as well as mutable data variables and data operations. A rich set of process constructs are supported, a number of which are adapted from Timed CSP [30]. Additional behavioral patterns which are useful in modeling and analyzing real-time systems are introduced. Examples are $deadline$ (which constrains a process to terminate within some time units), $timed\ interrupt$, etc. Instead of explicitly manipulating clock variables (as in Timed Automata), the time related process constructs are designed to build on implicit clocks. Further, we augment a system model with mutable variables and data structures (e.g. arrays, stacks, queues, or any user created data types), synchronous/asynchronous channels, etc.

In order to offer efficient mechanical verification support, a fully automated abstraction technique is developed to build an abstract finite state machine from the model. The idea is to dynamically create clocks (only if necessary) to capture constraints introduced by the timed process constructs. A clock may be shared for many constructs in order to reduce the number of clocks. Further, the clocks are deleted as early as possible. During system exploration, a constraint on the active clocks is maintained and solved using techniques based on Difference Bound Matrix (DBM [7]). We show that the abstraction is finite state and is subject to model checking. Further, it weakly bi-simulates the concrete model and, therefore, we may perform sound and complete LTL-X (i.e. LTL without the next operator) model checking or refinement checking upon the abstraction. We enhance our home-grown PAT model checker [33] (available at http://pat.comp.nus.edu.sg) with the technique and show its usability via automated verification of benchmark systems. We compare PAT with UPPAAL to show that our technique offers complementary support for analysis of real-time systems.

Section Organization The remainder of the paper is organized as follows. Section 2 presents the syntax and operational semantics of a subset of our modeling language. Section 3 presents the zone abstraction using dynamical clocks. Section 4 discusses the soundness of the abstraction and its implication on model checking. Section 5 discusses automation of the technique in the PAT model checker. Section 6 reviews related work and discusses future research direction.

2 Language Syntax and Operational Semantics

In this section, we introduce the compositional language to model real-time systems and then define its operational semantics. Let Σ be the set of event names.

Definition 1 (LTS). *A labeled transition system is 3-tuple $\mathcal{L} = (S, init, \rightarrow)$ where S is a set of system configurations, $init : S$ is an initial system configuration and $\rightarrow$: $S \times \Sigma \times S$ is a labeled transition relation.*

A run of an LTS is a finite or infinite sequence of alternating configurations/events, i.e. $\langle s_0, e_0, s_1, e_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all i. An execution is a sequence of events $\langle e_0, e_1, \cdots \rangle$ such that there exists a run $\langle s_0, e_0, s_1, e_1, \cdots \rangle$. For simplicity, we write $c \xrightarrow{x}$ to mean that there exists c' such that $c \xrightarrow{x} c'$.

2.1 Syntax

Definition 2 (Timed process). *A timed process (hereafter process) is defined by the following grammar*[1].

$$
\begin{array}{ll}
P = \mathit{Stop} \mid \mathit{Skip} & \text{– primitives} \\
\mid\ e \rightarrow P & \text{– event prefixing} \\
\mid\ [b]P & \text{– state guard} \\
\mid\ P \mid Q & \text{– general choice} \\
\mid\ P \parallel Q & \text{– parallel composition} \\
\mid\ P;\ Q & \text{– sequential composition} \\
\mid\ \mathit{Wait}[d] & \text{– delay} \\
\mid\ P\ \mathit{timeout}[d]\ Q & \text{– timeout} \\
\mid\ P\ \mathit{interrupt}[d]\ Q & \text{– timed interrupt} \\
\mid\ P\ \mathit{deadline}[d] & \text{– deadline} \\
\mid\ P \mathrel{\widehat{=}} Q & \text{– process definition}
\end{array}
$$

where P and Q range over processes, $e \in \Sigma$ is an observable event, b is a Boolean expression on global variables or process parameters and d is an integer constant.

Process Stop does nothing but idling. Process Skip terminates (possibly after some idling). Process $e \rightarrow P$ engages in event e first and then behaves as P. Notice that e may be an abstract event or a data operation, e.g. written in the form of $e\{x = 5;\ y = 3;\ \}$ or an external C# program. The data operation may update data variables (and is assumed to be executed atomically). For simplicity, the resultant data valuation is written as $e(V)$. A guarded process is written as $[b]P$. If b is true, then it behaves as P, else it idles until b becomes true. Process $P \mid Q$ offers a choice between P and Q. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization or shared variables. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately.

A number of timed process constructs can be used to capture common real-time system behavior patterns. Without loss of generality, we assume d is an integer constant. Process $\mathit{Wait}[d]$ idles for exactly d time units. In process $P\ \mathit{timeout}[d]\ Q$, the first observable event of P shall occur before d time units elapse (since the process starts). Otherwise, Q takes over control after exactly d time units elapse. Process $P\ \mathit{interrupt}[d]\ Q$ behaves exactly as P (which may engage in multiple observable events) until d time units elapse, and then Q takes over control. Process $P\ \mathit{deadline}[d]$ constrains P to terminate before d time units. We remark additional process constructs (e.g. if-then-else, while, etc.) can be defined using the above. In this setting, clock variables are made implicit and hence they cannot be compared with each other directly, which potentially allows efficient clock manipulation and hence system verification.

Definition 3 (System model). *A system model is a 3-tuple $\mathcal{S} = (\mathit{Var}, \mathit{init}, P)$ where Var is a set of global variables, init is the initial valuation of the variables and P is a process.*

[1] Hiding, external/internal choice, waituntil and more are skipped for simplicity. It should be clear that the discussion applies to those operators.

Example 1 (Fischer's Algorithm). The following models Fischer's mutual exclusion algorithm.

$$
\begin{array}{lll}
\textbf{var } x & = -1; & \\
\textbf{var } ct & = 0; & \\
Proc(i) & = [x = -1]Active(i) & \\
Active(i) & = (update.i\{x = i\} \rightarrow Skip)deadline[\delta]; & \\
 & \quad Wait[\epsilon]; & \\
 & \quad if\ (x = i)\ \{ & \\
 & \qquad cs.i\{ct = ct + 1\} \rightarrow & \\
 & \qquad exit.i\{ct = ct - 1;\ x = -1\} \rightarrow Proc(i) & \\
 & \quad \}\ else\ \{ & \\
 & \qquad Proc(i) & \\
 & \quad \} & \\
Protocol & = Proc(0) \parallel Proc(1) \parallel Proc(2); &
\end{array}
$$

where δ and ϵ are two integer constants with $\delta < \epsilon$; x and ct are global variables. The protocol is modeled as process $Protocol$, which is the parallel composition of three processes. Each of the three processes attempts to enter the critical section when x is -1, i.e. no other process is currently attempting. Once the process is active, it sets x to its identity i within δ time units (captured by $deadline[\delta]$). Then it idles for ϵ time units (captured by $Wait[\epsilon]$) and then checks whether x is still i. If so, it enters the critical section and leaves later. Otherwise, it restarts from the beginning. □

2.2 Semantics

In order to define the operational semantics of a system model, we define the notion of a configuration to capture the global system state during system execution.

Definition 4 (System configuration). *A system configuration is a pair $c = (V, P)$ where V is a variable valuation function and P is a process.*

A transition of the system is of the form $c \xrightarrow{x} c'$ where c and c' are the system configurations before and after the transition respectively. We adopt the following naming convention for transition labels: t denotes a non-negative real number; τ denotes an invisible event; ✓ is the event of process termination; $e \in \Sigma \cup \{\checkmark\}$ is an observable event; $x \in \Sigma \cup \{\tau, \checkmark\}$. For instance, $c \xrightarrow{t} c'$ denotes a transition of t time units elapsing. In the following, we present the firing rules which are associated with the timed process constructs, adopting the approach in [29].

$$\frac{t \leq d}{(V,\ Wait[d]) \xrightarrow{t} (V,\ Wait[d - t])}\ [\ de1\ ] \qquad \frac{}{(V,\ Wait[0]) \xrightarrow{\tau} (V, Skip)}\ [\ de2\ ]$$

The above captures behaviors of process $Wait[d]$. Rule $de1$ states that the process may idle for any amount of time as long as it is less than or equal to d time units; Rule $de2$ states that the process terminates immediately after d becomes 0.

$$\frac{(V, P) \xrightarrow{e} (V', P')}{(V, P\ timeout[d]\ Q) \xrightarrow{e} (V', P')}\ [\ to1\ ]$$

$$\frac{(V,P) \xrightarrow{\tau} (V',P')}{(V,P\ timeout[d]\ Q) \xrightarrow{\tau} (V',P'\ timeout[d]\ Q)}\ [\ to2\ ]$$

$$\frac{(V,P) \xrightarrow{t} (V,P'), t \leq d}{(V,P\ timeout[d]\ Q) \xrightarrow{t} (V,P'\ timeout[d-t]\ Q)}\ [\ to3\ ]$$

$$\frac{}{(V,P\ timeout[0]\ Q) \xrightarrow{\tau} (V,Q)}\ [\ to4\ ]$$

If an observable event x can be engaged by P, then $P\ timeout[d]\ Q$ becomes P' (rule $to1$). An invisible transition does not solve the $choice$ (rule $to2$). If P may idle for less than or equal to d time units, so is the composition (rule $to3$). When d becomes 0, Q takes over control by a silent transition (rule $to4$).

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V,P\ interrupt[d]\ Q) \xrightarrow{x} (V',P'\ interrupt[d]\ Q)}\ [\ it1\ ]$$

$$\frac{(V,P) \xrightarrow{t} (V,P'), t \leq d}{(V,P\ interrupt[d]\ Q) \xrightarrow{t} (V,P'\ interrupt[d-t]\ Q)}\ [\ it2\ ]$$

$$\frac{}{(V,P\ interrupt[0]\ Q) \xrightarrow{\tau} (V',Q)}\ [\ it3\ ]$$

Rule $it1$ states that if P engages in event x, $P\ interrupt[d]\ Q$ becomes $P'\ interrupt[d]\ Q$. Rule $it2$ states that if P may idle for less than or equal to d time units, so is the composition. When d time units elapse, Q takes over by a τ-transition.

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V,P\ deadline[d]) \xrightarrow{x} (V',P'\ deadline[d])}\ [\ dl1\ ]$$

$$\frac{(V,P) \xrightarrow{t} (V,P'), t \leq d}{(V,P\ deadline[d]) \xrightarrow{t} (V,P'\ deadline[d-t])}\ [\ dl2\ ]$$

Intuitively, $P\ deadline[d]$ behaves exactly as P except that it must terminate before d time units. The rest of the rules are straightforward extensions of those introduced in [29], which are presented in Appendix A.

Definition 5 (Concrete transition system). *Let $\mathcal{S} = (Var, init, P)$ be a system model. The concrete transition system corresponding to $\mathcal{S}$ is an LTS $\mathcal{L}_c^{\mathcal{S}} = (C_c, init_c, \rightarrow)$ where C_c is the set of reachable concrete system configurations, $init_c$ is the initial configuration $(init, P)$ and $\rightarrow$ is the smallest transition relation closed under the firing rules.*

3 Zone Abstraction

For the sake of model checking, we assume that all variables have finite domains and the process forbids unbounded non-tail recursion. Nonetheless, the number of concrete configurations (and hence the concrete transition system) is infinite because of the time transitions. In the following, we apply zone abstraction to build an abstract configuration system. Different from zone abstraction applied to Timed Automata [7, 38], we dynamically create/delete a set of clocks to precisely encode the timing requirements. We show that the abstract transition system is finite state and subject to model checking.

3.1 Clock Activation and De-activation

A clock is a variable ranging from 0 to some bounded natural number. Given a configuration (V, P), a clock is necessary to measure time elapsing if, and only if, a timed process is (e.g. $Wait[d]$, $P\ timeout[d]\ Q$, $P\ interrupt[d]\ Q$, or $P\ deadline[d]$) has been enabled. If a timed process (say $Wait[d]$) is enabled, we associate a clock (say tm) with the process to record time elapsing (written as $Wait[d]_{tm}$). The timing requirements can be captured using a constraint on the valuation of the clock. During system execution, multiple clocks may be used to capture quantitative timing constraints. A clock may become irrelevant as soon as the related process takes a transition. For instance, if P in $P\ timeout[d]_{tm}\ Q$ engages in an observable event, then the process transforms to P' and clock tm becomes irrelevant. It is known that model checking of real-time systems is exponential in the number of clocks. Therefore, it is desirable to use clocks only necessary and discharge them as early as possible.

Definition 6 (Abstract system configuration). *An abstract system configuration is a triple (V, P, D), where V is a variable valuation, P is a process and D is a zone.*

A *zone* is the maximal set of clock valuations satisfying a set of primitive clock constraints. A primitive constraint on a clock is of the form $tm \sim d$ where tm is a timer, d is a constant and $\sim$ is $\geq$, $=$, or $\leq$. Because clocks are implicit, clock readings cannot be compared directly. A zone is not empty if, and only if, the constraint is not false.

Next, we show how to systematically activate and de-activate clocks using process $Wait[d]$ and $P\ timeout[d]\ Q$ as examples. Let t be a fresh clock. Given an abstract configuration, we define function $\mathcal{A}(P, t)$ to recursively determine whether a clock is necessary and associate the clock with the relevant process constructs. A clock is necessary if and only if one (or more) timed pattern has just been enabled. For instance,

$$\begin{aligned}\mathcal{A}(\,Wait[d]_{t'}, t) &= Wait[d]_{t'}\\ \mathcal{A}(\,Wait[d], t) &= Wait[d]_{t}\end{aligned}$$

where $Wait[d]_{t'}$ denotes that the timed pattern is associated with a clock t', whereas $Wait[d]$ denotes that it has not been associated with a clock. The intuition is for the former case, $\mathcal{A}$ does nothing and t is not used (since it is not necessary to introduce another clock); for the latter case, $\mathcal{A}$ associates t the the timer pattern. The following shows how to apply $\mathcal{A}$ to process $P\ timeout[d]\ Q$.

$$\begin{aligned}\mathcal{A}(P\ timeout[d]_{t'}, t) &= P\ timeout[d]_{t'}\ Q\\ \mathcal{A}(P\ timeout[d]\ Q, t) &= \mathcal{A}(P)\ timeout[d]_{t}\ \mathcal{A}(Q)\end{aligned}$$

$$
\begin{array}{lll}
\mathcal{A}(P \mid Q, t) & = \mathcal{A}(P, t) \mid \mathcal{A}(Q, t) & \\
\mathcal{A}(P \parallel Q, t) & = \mathcal{A}(P, t) \parallel \mathcal{A}(Q, t) & \\
\mathcal{A}(P;\ Q, t) & = \mathcal{A}(P, t);\ Q & \\
\mathcal{A}(P, t) & = \mathcal{A}(Q, t) & - \text{if } P \mathrel{\widehat{=}} Q \\
\mathcal{A}(\mathit{Wait}[d], t) & = \mathcal{A}(\mathit{Wait}[d]_t) & \\
\mathcal{A}(P\ \mathit{timeout}[d]\ Q, t) & = \mathcal{A}(P, t)\ \mathit{timeout}[d]_t\ \mathcal{A}(Q, t) & \\
\mathcal{A}(P\ \mathit{interrupt}[d]\ Q, t) & = \mathcal{A}(P, t)\ \mathit{interrupt}[d]_t\ \mathcal{A}(Q, t) & \\
\mathcal{A}(P\ \mathit{deadline}[d], t) & = \mathcal{A}(P, t)\ \mathit{deadline}[d]_t &
\end{array}
$$

Fig. 1. Clock activation: $\mathcal{A}(P, t)$ is P except the above cases.

If a clock t' has already been associated with $P\ \mathit{timeout}[d]\ Q$, then function $\mathcal{A}$ simply returns the abstract configuration. Otherwise, it is associated with t and further $\mathcal{A}$ is applied to the sub-processes P and Q recursively. The complete definition of function $\mathcal{A}$ is presented in Figure 1. In an abuse of notation, given an abstract configuration $c = [V, P]_D$, we write $\mathcal{A}(c)$ to be $[V, \mathcal{A}(P)]_{D \wedge t=0}$ if t is used; otherwise $\mathcal{A}(c)$ is simply c.

A runtime clock may later be discarded when the time-related process has evolved such that the reading of the clock is no longer relevant. For instance, the clock associated with $P\ \mathit{timeout}[d]\ Q$ can be discarded when P engages in an observable event. It should be clear that we can identify the set of active runtime clocks by a similar procedure. To minimize clocks, all in-active runtime clocks, and the associated timing constraints, shall be pruned from D. Notice that t_G is never pruned. We assume a function $\mathcal{D}$ which performs clock de-activation in a sound and complete way.

3.2 Zone Abstraction

We define $D^{\uparrow} = \{t + d \mid t \in D \wedge d \in \mathbb{R}_+\}$, i.e. the zone obtained by delaying arbitrary amount of time. Notice that all clocks take the same pace. Next, we define function ι to compute the zone which can be reached by idling from a given abstract system configuration [38], presented in Figure 2. Given the current zone D, process $P\ \mathit{timeout}[d]_{tm}\ Q$ may keep idling as long as P may keep idling and the reading of clock tm is less or equal to d (so that $\mathit{timeout}$ has not occur). The rest are similarly defined.

In the following, we define the firing rules based on the abstract system configurations. The idea is to eliminate time transitions altogether and use the timing constraint to ensure that the time-related process constructs behave correctly. An abstract transition is of the form $(V, P, D) \overset{x}{\hookrightarrow} (V', P', D')$, where $x \in \Sigma \cup \{\checkmark, \tau\}$.

$$
\frac{}{(V,\ \mathit{Wait}[d]_{tm}, D) \overset{\tau}{\hookrightarrow} (V, \mathit{Skip}, D^{\uparrow} \wedge tm = d)}\ [\ \mathit{ade}\ ]
$$

Process $\mathit{Wait}(d)$ idles for exactly d time units and then engages in event τ and the process transforms to Skip. Intuitively, it should be clear that this is 'equivalent' to the concrete firing rules. We will define what equivalence means later in this section.

$$
\begin{array}{lll}
\iota(V, \mathit{Stop}, D) & = D^{\uparrow} & \\
\iota(V, \mathit{Skip}, D) & = D^{\uparrow} & \\
\iota(V, e \rightarrow P, D) & = D^{\uparrow} & \\
\iota(V, [b]P, D) & = D^{\uparrow} & \\
\iota(V, P \mid Q, D) & = \iota(V, P, D) \wedge \iota(V, Q, D) & \\
\iota(V, P \parallel Q, D) & = \iota(V, P, D) \wedge \iota(V, Q, D) & \\
\iota(V, P;\ Q, D) & = \iota(V, P, D) & \\
\iota(V, \mathit{Wait}[d]_{tm}, D) & = D^{\uparrow} \wedge tm \leq d & \\
\iota(V, P\ \mathit{timeout}[d]_{tm}\ Q, D) & = \iota(V, P, D) \wedge tm \leq d & \\
\iota(V, P\ \mathit{interrupt}[d]_{tm}\ Q, D) & = \iota(V, P, D) \wedge tm \leq d & \\
\iota(V, P\ \mathit{deadline}[d]_{tm}, D) & = \iota(V, P, D) \wedge tm \leq d & \\
\iota(V, P, D) & = \iota(V, Q, D) & \text{– if } P \mathrel{\widehat{=}} Q
\end{array}
$$

Fig. 2. Idling calculation

$$
\frac{(V, P, D) \stackrel{\tau}{\hookrightarrow} (V', P', D')}{(V, P\ \mathit{timeout}[d]_{tm}\ Q, D) \stackrel{\tau}{\hookrightarrow} (V', P'\ \mathit{timeout}[d]_{tm}\ Q, D' \wedge tm \leq d)}\ [\ \mathit{ato}1\ ]
$$

$$
\frac{(V, P, D) \stackrel{x}{\hookrightarrow} (V', P', D')}{(V, P\ \mathit{timeout}[d]_{tm}\ Q, D) \stackrel{x}{\hookrightarrow} (V', P', D' \wedge tm \leq d)}\ [\ \mathit{ato}2\ ]
$$

$$
\frac{}{(V, P\ \mathit{timeout}[d]_{tm}\ Q, D) \stackrel{\tau}{\hookrightarrow} (V, Q, tm = d \wedge \iota(V, P, D))}\ [\ \mathit{ato}3\ ]
$$

Depending on when the first event of P takes place and whether it is observable, process $P\ \mathit{timeout}[d]\ Q$ behaves differently in three ways. An observable transition of P must occur no later than d time units since the process is enabled (rule $\mathit{ato}1$ and $\mathit{ato}2$). If the first transition is observable, then the *choice* is resolved (rule $\mathit{ato}2$). If it is silent, then the it transforms to $P'\ \mathit{timeout}[d]\ Q$. If P may delay more than d time units (captured by the constraint $\iota(V, P, D)$), then it times out after exactly d time units (rule $\mathit{ato}3$). The constraint $tm = d \wedge \iota(V, P, D)$ means that the delay is exactly d time units and P must be idling during the period.

$$
\frac{(V, P, D) \stackrel{x}{\hookrightarrow} (V', P', D')}{(V, P\ \mathit{interrupt}[d]_{tm}\ Q, D) \stackrel{x}{\hookrightarrow} (V', P'\ \mathit{interrupt}[d]_{tm}\ Q, D' \wedge tm \leq d)}\ [\ \mathit{ait}1\ ]
$$

$$
\frac{}{(V, P\ \mathit{interrupt}[d]_{tm}\ Q, D) \stackrel{\tau}{\hookrightarrow} (V, Q, tm = d \wedge \iota(V, P, D))}\ [\ \mathit{ait}2\ ]
$$

Process $P\ \mathit{interrupt}[d]\ Q$ behaves differently in two ways. Transitions of P must take place no later than d time units since the process is enabled (rule $\mathit{ait}1$). If P may delay more than d time units (captured by the constraint $\iota(V, P, D)$), then it is interrupted after exactly d time units (rule $\mathit{ait}2$).

Fig. 3. A simple example

$$\frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D'), x \neq \checkmark}{(V, P\ deadline[d]_{tm}, D) \overset{x}{\hookrightarrow} (V', P'\ deadline[d]_{tm}, D' \wedge tm \leq d)}\ [\ adl\ ]$$

Process $P\ deadline[d]$ behaves exactly as P except that any transition must occur before d time units.

The rest of the firing rules is present in Appendix B. A transition is valid if, and only if, it conforms to the firing rules and the resultant zone is not empty. Intuitively, this means that a transition must be allowed by the untimed system and at the same time satisfy the additional timing requirement.

Definition 7 (Abstract transition system). *Let $\mathcal{S} = (Var, init, P)$ be a system model. The abstract transition system corresponding to $\mathcal{S}$ is an LTS $\mathcal{L}_a^{\mathcal{S}} = (C_a, init_a, \hookrightarrow)$ where C_a is the set of reachable valid abstract system configurations, $init_a$ is the initial configuration $(init, P, true)$ and $\hookrightarrow$ is the smallest transition relation satisfying $\forall\, c, c' : C_a.\ c \overset{e}{\hookrightarrow} c' \Leftrightarrow \mathcal{A}(c) \overset{e}{\hookrightarrow} \mathcal{D}(c')$.*

Example 2 (A simple example). Assume a model $(\varnothing, \varnothing, P)$ with no variable and P is $(a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]\ c \rightarrow Stop$. The abstract transition system is shown in Figure 3, where transition label τ is skipped for simplicity. Let $\langle t_1, t_2 \rangle$ be a sequence of clocks. The following illustrates how to construct the abstract transition system. Let s_0 be $(\varnothing, P, true)$.

- Step 1: apply $\mathcal{A}$ to s_0 to get

  $$s_1 = (\varnothing, (a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow Stop, t_1 = 0)$$

- Step 2: apply rule $ait1$ to s_1 to get

  $$s_2 = (\varnothing, (Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow Stop, 0 \leq t_1 \leq 3)$$

  Notice that $(t_1 = 0)^{\uparrow}$ equals to $t_1 \geq 0$.
- Step 3: apply $\mathcal{D}$ to s_2. The result is exactly s_2. We obtain the transition from state 1 to state 2.
- Step 4: apply rule $ait2$ to s_1 to get

  $$s_3 = (\varnothing, (c \rightarrow Stop), t_1 \geq 0 \wedge t_1 = 3)$$

  Notice that $\iota(\varnothing, a \rightarrow Wait[5];\ b \rightarrow Stop, t_1 = 0)$ is $t_1 \geq 0$.
- Step 5: apply $\mathcal{D}$ to s_3 to get $s_4 = (\varnothing, (c \rightarrow Stop), true)$. We remark that because t_1 becomes inactive, it is pruned from the constraint. This generates the transition from state 1 to state 3.

– Step 6: apply $\mathcal{A}$ to s_2 to get

$$s_5 = (\varnothing, (\mathit{Wait}[5]_{t_2};\ b \rightarrow \mathit{Stop})\ \mathit{interrupt}[3]_{t_1}\ c \rightarrow \mathit{Stop},\\ 0 \leq t_1 \leq 3 \wedge t_2 = 0)$$

– Step 7: apply rule $ait1$ to s_5, we get

$$s_6 = (\varnothing, (\mathit{Skip};\ b \rightarrow \mathit{Stop})\ \mathit{interrupt}[3]_{t_1}\ c \rightarrow \mathit{Stop}, 0 \leq t_1 \leq 3 \wedge t_2 = 5)$$

Notice that the timing constraint is false given that all timers take the same pace. Refer to next section on how this is discovered systematically.

– Step 8: apply rule $ait2$ to s_5 to get

$$s_7 = (\varnothing, c \rightarrow \mathit{Stop}, t_1 \geq 0 \wedge t_2 \geq 0 \wedge t_2 \leq 5 \wedge t_1 = 3)$$

– Step 9: apply $\mathcal{D}$ to s_7 to get s_4. Notice that both clocks are inactive and therefore pruned. This generates the transition from state 2 to state 3.
– Lastly, we generate the transition from state 3 to state 4. Notice that this transition involves no quantitative timing.

3.3 Zone Operations

In order to construct and verify the abstract transition system, we need efficient and sound procedures to manipulate zones. For instance, we need to determine whether a zone is empty or not. The procedure must be sound (so that a valid configuration is not missed) and complete (so that invalid configurations are ruled out).

A zone D can be equivalently represented as a difference bound matrices (DBM). Let $\{t_1, t_2, \cdots, t_n\}$ be a set of n clocks. Let t_0 be a dummy clock whose value is always 0. A DBM representing a constraint on the clocks contains $n+1$ rows, each of which contains $n+1$ elements. Let D^i_j represent entry (i, j) in the matrix. A DBM represents the constraint: $\forall\, i : 0\,..\,n.\ \forall\, j : 0\,..\,n.\ t_i - t_j \leq D^i_j$. The most important property of DBM is that there is a relatively efficient procedure to compute a unique canonical form. Given a DBM in canonical form, checking whether the zone is empty or not is as easy as looking up an entry in the matrix. DBM has been well studied [7, 2, 3]. In the following, we briefly introduce the relevant DBM operations/properties. We skip the discussion on rest of the zone operations (e.g. $D^{\uparrow}$, adding a constraint, etc.) as they resemble the discussion in [3].

Calculate canonical form In theory, there are infinite different timing constraints which represent the zone. For instance, $0 \leq t_1 \leq 3 \wedge 0 \leq t_1 - t_2 \leq 3$ is equivalent to $0 \leq t_1 \leq 3 \wedge 0 \leq t_1 - t_2 \leq 3 \wedge t_2 \leq 1000$. In order to systematically compare two zones, we compute their unique canonical forms. In other words, we compute the tightest bound on each clock difference. If the clocks are viewed as vertices in a weighted graph and the clock difference as the label on the edge connecting two clocks, the tightest clock difference is the shortest path between the respective vertices. The Floyd-Warshall algorithm [12] thus can be used to compute the canonical from. Given that this algorithm is cubic in the number of clocks, it is desirable to reduce the number of clocks. Besides, the algorithm must be invoked if necessary and ideally (if possible) the result of performing an operation on a canonical DBM should be canonical.

Check satisfiability In order to construct the abstract transition system, it is essential to check whether a zone is empty. Given the DBM representing a zone, it is unsatisfiable if, and only if, there is a clock which has a negative difference from itself, i.e. $t_k - t_k < 0$ for some k so that the constraint is false. If the DBM is in canonical form, then there exists at least one D_i^i which is negative. Further, it can be shown that the DBM is false if, and only if, D_0^0 is negative. Therefore, we compute the canonical form whenever it is necessary to check for satisfiability.

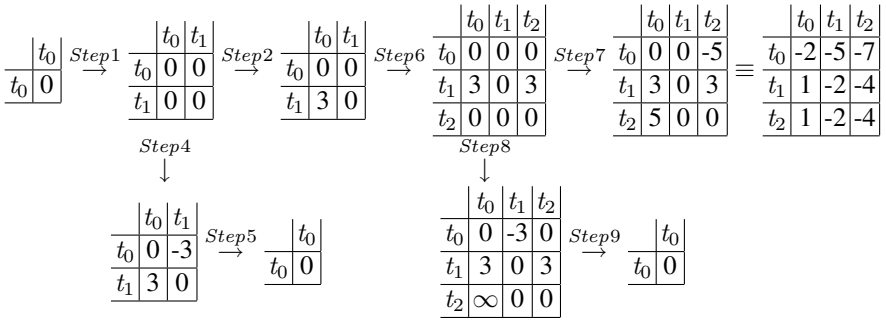
Add clocks In our setting, clocks may be introduced during system exploration. We remark that clocks are a constant set in Timed Automata. Assume the new clock is t_k and the given DBM is canonical. The following shows how the DBM is updated with entries for t_k. For all i, $D_k^i = D_0^i$ and $D_i^k = D_i^0$ as the new clock always starts with value 0. By a simple argument, it can be shown the resultant DBM is canonical.

	t_0	t_1	$\cdots$	t_i	$\cdots$	t_{k-1}	$\mathbf{t_k}$
t_0	0	d_1^0	$\cdots$	d_i^0	$\cdots$	d_{k-1}^0	$\mathbf{0}$
t_1	d_0^1	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^1}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
t_i	d_0^i	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^i}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
t_{k-1}	d_0^{k-1}	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^{k-1}}$
$\mathbf{t_k}$	$\mathbf{0}$	$\mathbf{d_1^0}$	$\cdots$	$\mathbf{d_i^0}$	$\cdots$	$\mathbf{d_{k-1}^0}$	$\mathbf{0}$

Prune clocks Because entries in a canonical DBM represent the tightest bound on clock differences, pruning clocks is simply to remove the relevant row and column in the table. It should be clear that the remaining DBM is canonical, i.e. the bounds can not be possibly tightened with less constraints.

Notice that the number of reachable timing constraints in canonical form are finite as proved in [7]. As a result, the abstraction system is finite state and therefore subject to model checking[2].

Example 3 (DBM manipulation example). The following illustrates how the DBM is transformed through system exploration in Example 2.



The DBM obtained after Step 7 is indeed false, i.e. after applying the Floyd-Warshall algorithm, D_0^0 is -2. □

[2] assume that the variable domains are finite and the reachable process expressions are finite.

4 System Verification

In this section, we prove that our abstraction is sound and complete with respect to a number of properties. The abstract transition system is shown to be equivalent to the concrete transition system using a specialized bi-simulation relationship [21]. We then show that two different system verification methods are sound.

In the concrete transition system, if a configuration (V', P') can be reached from (V, P) by idling only, we write $(V, P) \rightsquigarrow (V', P')$. By a simple argument, it can be shown that if $(V, P) \rightsquigarrow (V', P')$, then $V = V'$. We write $(V, P) \overset{x}{\rightsquigarrow} (V', P')$ if, and only if, there exists (V, P_1), (V', P_2) such that $(V, P) \rightsquigarrow (V, P_1)$ and $(V, P_1) \xrightarrow{x} (V', P_2)$ and $(V', P_2) \rightsquigarrow (V', P')$.

Definition 8 (Time abstract bi-simulation). *Let $\mathcal{S} = (Var, init, P)$ be a model. Let $\mathcal{L}_c^{\mathcal{S}} = (C_c, init_c, \rightarrow)$ and $\mathcal{L}_a^{\mathcal{S}} = (C_a, init_a, \hookrightarrow)$ be the concrete and abstract transition systems. $\mathcal{L}_c$ and $\mathcal{L}_a$ are time abstract bi-similar (hereafter bi-similar) if, and only if, there exists a binary relation $\mathcal{R} : C_c \rightarrow C_a$ such that $(init_c, init_a) \in \mathcal{R}$ and $\forall\, x : \Sigma \cup \{\checkmark, \tau\}$; $c = (V_c, P_c)$; $a = (V_a, P_a, D_a)$ such that $(c, a) \in \mathcal{R}$ implies,*

- *$V_c = V_a$,*
- *if $c \overset{x}{\rightsquigarrow} c'$, then for some a', $a \overset{x}{\hookrightarrow} a'$ and $(c', a') \in \mathcal{R}$.*
- *if $a \overset{x}{\hookrightarrow} a'$, then for some c', $c \overset{x}{\rightsquigarrow} c'$ and $(c', a') \in \mathcal{R}$.*

We say that c and a are bi-similar, written as $c \sim a$, if, and only if, there exists $\mathcal{R}$ such that the transition systems are bi-similar. Notice that $\mathcal{L}_c$ and $\mathcal{L}_a$ are bi-similar if, and only if, $init_c \sim init_a$.

Theorem 1. *Let $\mathcal{S} = (Var, init, P)$ be a system model. $\mathcal{L}_c^{\mathcal{S}}$ and $\mathcal{L}_a^{\mathcal{S}}$ are time abstract bi-similar.* □

By definition, it suffices to construct a binary relation which satisfies the condition. We present the proof based on structural induction in Appendix C. Time abstract bi-simulation is strong enough to guarantee soundness on verification of a number of useful properties.

LTL-X Model Checking In this setting, the properties are linear temporal logic formulae without the next operator (i.e. LTL-X), constituted by propositions on global variables. Notice that no clocks are allowed in the property. The philosophy is that a critical property may often be independent of the speed of the hardware on which the system is deployed, whereas the model of the implementation shall incorporate known hardware limitations.

Example 4. Given Example 1, the following are some critical properties.

$\Box ct \leq 1$	– safety property
$\Box(x = i \Rightarrow \Diamond cs.i)$	– liveness property

where $\Box$ and $\Diamond$ read as 'always' and 'eventually'. The first property precisely states mutual exclusion, i.e. at all time, there must not be 2 or more processes in the critical section. The second states that if process i is attempting to access the shared resource, it must eventually do so.

In order to reflect model checking results on the abstract transition system to the original system, we need to establish that the abstract transition system is equivalent to the concrete one with respect to LTL-X formulae. The idea is to show stutter equivalence between traces of the abstract system and the concrete system. Given two traces $tr_1 = \langle V_0, V_1, \cdots \rangle$ and $tr_2 = \langle V'_0, V'_1, \cdots \rangle$, tr_1 and tr_2 are stutter equivalent if, and only if, tr_1 and tr_2 can be partitioned into blocks, so that the variable valuation in the k-th block in tr_1 is the same as those in the k-th block of tr_2. Formally, tr_1 is stutter equivalent to tr_2 if, and only if, there are two infinite sequences of integers $0 < i_0 < i_1 < \cdots$ and $0 < j_0 < j_1 < \cdots$ such that for every block $k \geq 0$ holds $V_{s_{i_k}} = V_{s_{i_k+1}} = \cdots = V_{s_{i_{k+1}-1}} = V'_{s_{j_k}} = V'_{s_{j_k+1}} = \cdots = V'_{s_{j_{k+1}-1}}$. It is known that tr_1 satisfies an LTL-X property if, and only if, tr_2 does.

Let ϕ be such a property, we write $\mathcal{L} \vdash \phi$ to denote that the labeled transition system $\mathcal{L}$ satisfies ϕ, i.e. every trace of $\mathcal{L}$ satisfies ϕ.

Lemma 1. *Let $\mathcal{S} = (Var, init, P)$ be a system model. For every trace of the concrete transition system $\mathcal{L}_c$, there is a stutter equivalent trace of the abstract transition system $\mathcal{L}_a$ and vice versa.*

The above lemma can be proved by structural induction or implied from Theorem 1. Consequently, the following theorem can be proved straightforwardly.

Theorem 2. *Let $\mathcal{S} = (Var, init, P)$ be a system model. Let ϕ be a LTL-X formula constituted by propositions on Var. $\mathcal{L}_c^{\mathcal{S}} \vDash \phi$ if, and only if, $\mathcal{L}_a^{\mathcal{S}} \vDash \phi$.* □

Refinement Checking In this setting, we investigate an alternative verification schema for finite system executions. That is, to verify whether the system satisfies the property by showing a refinement relationship between the system and a model which models the property. A variety of refinement relationships have been studied, e.g. trace-refinement, stable failures refinement and failures/divergence refinement [16]. In order to check refinement between two (timed) models, time abstraction must be applied to both models.

Example 5. Given the model presented in Example 1, a natural question is whether ϵ and δ are necessary or their values would make a difference. Equivalently, the former is to ask whether $(init, uProcotol)$ where $init = \{x \mapsto -1, ct \mapsto 0\}$ and $uProcotol$ defined as follows, trace-refines the original one $(init, Procotol)$.

$$
\begin{array}{lll}
uProc(i) & = [x = -1]uActive(i) \\
uActive(i) & = update.i\{x = i\} \rightarrow \\
& \quad if\ (x = i)\ \{ \\
& \qquad cs.i\{ct = ct + 1\} \rightarrow \\
& \qquad exit.i\{ct = ct - 1;\ x = -1\} \rightarrow uProc(i) \\
& \quad \}\ else\ \{ \\
& \qquad uProc(i) \\
& \quad \} \\
uProtocol & = uProc(0) \parallel uProc(1) \parallel uProc(2);
\end{array}
$$

By showing trace refinement in both directions, we may establish trace equivalence. Or, the users may change the value of ϵ and δ check for equivalence. □

Let $\mathcal{L}$ be an LTS. A finite sequence of observable events, e.g. $\langle x_0, x_1, \cdots, x_m \rangle$, is a trace of $\mathcal{L}$ if, and only if, there exists a finite execution $\langle c_0, e_0, c_1, e_1, \cdots, e_n, c_{n+1} \rangle$ such that $\langle e_0, e_1, \cdots, e_n \rangle \upharpoonright \{\tau\} = \langle x_0, x_1, \cdots, x_m \rangle$ where $tr \upharpoonright X$ removes the events in X from the sequence tr. The set of all traces of $\mathcal{L}$ is written as $traces(\mathcal{L})$.

Given a finite trace tr and a configuration c in $\mathcal{L}$, we write c/tr to denote the set of system configurations that can be reached from c via trace tr or idling. Because of nondeterminism, multiple configurations can be reached via the same trace. The refusals are the sets of observable event sets which may be *refused*.

$$refusals(c) = \{X : \mathbb{P}\, \Sigma \mid \forall\, e : X \ \nexists\, c' \ c \overset{e}{\rightsquigarrow} c'\}$$

where $\mathbb{P}\, \Sigma$ is the power sets of Σ. The failures of $\mathcal{L}$ is defined as follows.

$$failures(\mathcal{S}) = \{(tr, X) \mid tr \in traces(\mathcal{L}) \wedge X \in refusals(init/tr)\}$$

If (tr, X) is a failure of the model, this means that the model can engage in the sequence of events recorded by tr, and then refuse to perform any event in X.

Definition 9. *Let $\mathcal{S}_i = (Var_i, init_i, P_i)$ where $i \in \{1, 2\}$ be two system models. $\mathcal{S}_1$ trace-refines $\mathcal{S}_2$ if, and only if, $traces(\mathcal{L}_c^{\mathcal{S}_1}) \subseteq traces(\mathcal{L}_c^{\mathcal{S}_2})$. $\mathcal{S}_1$ refines $\mathcal{S}_2$ in the failures semantics if, and only if, $traces(\mathcal{L}_c^{\mathcal{S}_1}) \subseteq traces(\mathcal{L}_c^{\mathcal{S}_2})$ and $failures(\mathcal{L}_c^{\mathcal{S}_1}) \subseteq failures(\mathcal{L}_c^{\mathcal{S}_2})$.*

In the following, we argue that it is sound and complete to show stable failures refinement (i.e. assuming both models are divergence-free) between the abstraction transition systems in order to show failures refinement between the concrete models.

Theorem 3. *Let $\mathcal{S}_i$ where $i \in \{1, 2\}$ be two models. $\mathcal{S}_1$ refines $\mathcal{S}_2$ in stable failures semantics iff $traces(\mathcal{L}_a^{\mathcal{S}_1}) \subseteq traces(\mathcal{L}_a^{\mathcal{S}_2})$ and $failures(\mathcal{L}_a^{\mathcal{S}_1}) \subseteq failures(\mathcal{L}_a^{\mathcal{S}_2})$.* □

By Theorem 1, it should be clear that our abstraction preserves failures. Intuitively, this is because not only observable transitions but also τ-transitions are preserved by the abstraction. The theorem can then be proved straightforwardly. We remark that it is clear the failures refinement subsumes trace-refinement and, therefore, it too can be supported by only checking the abstract transition systems.

5 Implementation and Evaluation

PAT is a self-contained environment for system specification, simulation and verification. It supports multi-languages targeting concurrent/distributed systems. The techniques presented in this paper have been implemented in PAT. PAT verifies LTL properties using an on-the-fly automata-based approach [35]. PAT verifies refinement relationship using an on-the-fly simulation checking approach [32]. In the following, we present the experiments results on two bench models. The models and PAT are available at http://pat.comp.nus.edu.sg.

Table 1 shows the experiment results on the Fischer's mutual exclusion algorithm and a railway control system [38]. The data is obtained with Intel Core 2 Quad 9550

Model	Size	Property	States/Transitions	PAT (s)
Fischer	4	$\Box ct \leq 1$	3452/8305	0.22
Fischer	5	$\Box ct \leq 1$	26496/73628	2.49
Fischer	6	$\Box ct \leq 1$	207856/654776	27.7
Fischer	7	$\Box ct \leq 1$	1620194/5725100	303
Fischer	4	$\Box(x = i \Rightarrow \Diamond cs.i)$	5835/16776	0.53
Fischer	5	$\Box(x = i \Rightarrow \Diamond cs.i)$	49907/169081	5.83
Fischer	6	$\Box(x = i \Rightarrow \Diamond cs.i)$	384763/1502480	70.5
Fischer	4	$Protocol\ refines\ uProtocol$	7741/18616	5.22
Fischer	5	$Protocol\ refines\ uProtocol$	72140/201292	126.3
Fischer	6	$Protocol\ refines\ uProtocol$	705171/2237880	3146
Railway Control	4	deadlock-free	853/1132	0.11
Railway Control	5	deadlock-free	4551/6115	0.42
Railway Control	6	deadlock-free	27787/37482	3.07
Railway Control	7	deadlock-free	195259/263641	24.2
Railway Control	8	deadlock-free	1563177/2111032	223.1
Railway Control	4	$\Box(appr.1 \rightarrow \Diamond leave.1)$	1504/1985	0.16
Railway Control	5	$\Box(appr.1 \rightarrow \Diamond leave.1)$	8137/10862	0.95
Railway Control	6	$\Box(appr.1 \rightarrow \Diamond leave.1)$	50458/67639	6.58
Railway Control	7	$\Box(appr.1 \rightarrow \Diamond leave.1)$	359335/482498	58.63

Table 1. Experiment results

CPU at 2.83GHz and 2GB memory. In both examples, PAT performs reasonably well. It handles 10^7 states/transition in a few hours, which is comparable to existing model checkers [17, 28]. Further, a simple experiment shows that the computational overhead of calculating clocks/DBMs is around one third of the overall time.

The data on UPPAAL [20] or RED [36] verifying the same models has been omitted from the table. Because UPPAAL and PAT are based on a different modeling language, the results must be taken with a grain of salt. The state graph generated from a PAT model may contain unnecessary τ-transitions introduced by the compositional process constructs, e.g. the τ in rule $ato3$. In hand-crafted UPPAAL models, however, the τ-transitions may be removed by carefully manipulating the clock guards and grouping clock guards and events on the same transition. In such a setting, verification of the UPPAAL is faster (by a factor related to the number of such τ-transitions). However, our experiment show that if we manually construct a PAT model and a UPPAAL model with the same state graph, then PAT and UPPAAL have a similar performance.

6 Conclusion

This work is related to specification and verification of real-time systems. Compositional specification based on process algebras for real-time systems has been studied extensively, e.g. the algebra of timed processes ATP [31, 24], CCS + real time [37] and timed CSP [26, 30]. Verification support has been developed for these specification language. For instance, a preliminary PVS encoding of Timed CSP was presented

in [5], which rely heavily on user interaction for formal proving of real-time systems. In [38], a constraint solving method was proposed to verify CCS + real time. A number of verification support for ATP were evidenced in [25, 6]. The modeling language Timed Automata [1] gathered more attention later on, especially in terms of mechanical verification. Several model checkers have been developed with Timed Automata (or a simplified version named timed safety automata [15]) being the core of their input languages [20, 4, 34]. The zone abstraction is closely related to works presented in [38], where a similar compositional abstraction method is discussed for CCS + real time. The difference is that we use implicit clocks and make the specification fully compositional. The soundness discussion of our abstraction is inspired by [21]. A remotely related modeling language is statecharts [13] with clocks, which too is compositional. This work follows the approach of Timed CSP and significantly extends the notion to cover a wide range of application domains. We developed a self-contained toolkit PAT to verify our models. To the best of our knowledge, there are few verification support for Timed CSP, e.g. the theorem proving approach documented in [5], the translation to UPPAAL models [8, 9] and the approach based on constraint solving [10]. The PAT model checker is the first dedicated verification tool support for Timed CSP models adapting advanced verification techniques for real-time systems. In addition, PAT complements UPPAAL with the ability to check full LTL-X property and check refinement relationship. PAT is remotely related to the Spin model checker (on automata-based LTL model checking) [17] and the FDR refinement checker (on refinement checking) [28].

We remark that verification on CSP-based models has been traditional based on refinement checking [27], e.g. using the FDR checker [28]. One research direction we are currently investigating is to check timed refinement relationship between two timed models. The main challenge is that abstraction must be applied separately to two timed models and yet preserve timed trace/failures equivalence.

References

1. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *CAV'99*, volume 1633 of *LNCS*, pages 341–353. Springer, 1999.
3. J. Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.
4. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.
5. P. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, 1999.
6. E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems. In *CAV'01*, volume 2102 of *LNCS*, pages 391–395. Springer, 2001.
7. D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

8. J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Wang Yi. Timed Patterns: TCOZ to Timed Automata. In *ICFEM'04*, volume 3308 of *LNCS*, pages 483–498. Springer, 2004.
9. J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Wang Yi. Timed Automata Patterns. *IEEE Trans. Software Eng.*, 34(6):844–859, 2008.
10. J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *ICFEM 2006*, volume 4260 of *LNCS*, pages 342–359. Springer, 2006.
11. J. S. Dong, B. P. Mahony, and N. Fulton. Modeling Aircraft Mission Computer Task Rates. In *FM'99*, page 1855, 1999.
12. R. W. Floyd. Algorithm 97: Shortest Path. *Commun. ACM*, 5(6):345, 1962.
13. D. Harel. Some Thoughts on Statecharts, 13 Years Later. In *CAV'97*, volume 1254 of *LNCS*, pages 226–231. Springer, 1997.
14. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/video Protocol: an Industrial Case Study using UPPAAL. In *RTSS'97*, pages 2–13, 1997.
15. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.
16. C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
17. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.
18. L. M. Lai and P. Watson. A Case Study in Timed CSP: The Railroad Crossing Problem. In *HART'97*, pages 69–74, 1997.
19. K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing Real-time Embedded Software using UPPAAL-TRON: an Industrial Case Study. In *EMSOFT 2005*, pages 299–306, 2005.
20. K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
21. K. G. Larsen and Wang Yi. Time-abstracted Bisimulation: Implicit Specifications and Decidability. *Information and Computation*, 134(2):75–101, 1997.
22. M. Lindahl, P. Pettersson, and Y. Wang. Formal Design and Analysis of a Gearbox Controller. *STTT'01*, 3(3):353–368, 2001.
23. N. A. Lynch and F. W. Vaandrager. Action Transducers and Timed Automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.
24. X. Nicollin and J. Sifakis. The Algebra of Timed Processes, ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.
25. X. Nicollin, J. Sifakis, and S. Yovine. Compiling Real-Time Specifications into Extended Automata. *IEEE Trans. Software Eng.*, 18(9):794–804, 1992.
26. G. M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. In *ICALP86*, volume 226 of *LNCS*, pages 314–323. Springer, 1986.
27. A. W. Roscoe. On the expressive power of csp refinement. *Formal Asp. Comput.*, 17(2):93–112, 2005.
28. A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In *TACAS '95*, volume 1019 of *LNCS*, pages 133–152. Springer, 1995.
29. S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116(2):193–213, 1995.
30. S. Schneider. *Concurrent and Real-time Systems*. John Wiley and Sons, 2000.
31. J. Sifakis. The Compositional Specification of Timed Systems - A Tutorial. In *CAV'99*, volume 1633 of *LNCS*, pages 2–7. Springer, 1999.

32. J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA 2008*, pages 307–322. Springer.
33. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. *CAV'09*, 5643, 2009.
34. S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying Abstractions of Timed Systems. In *'CONCUR'96*, volume 1119 of *LNCS*, pages 546–562. Springer, 1996.
35. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *Proc. of the Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society, 1986.
36. F. Wang, R. Wu, and G. Huang. Verifying Timed and Linear Hybrid Rule-Systems with RED. In *SEKE'05*, pages 448–454, 2005.
37. Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *ICALP'91*, volume 510 of *LNCS*, pages 217–228. Springer, 1991.
38. Wang Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-time Communicating Systems by Constraint-Solving. In *FORTE'94*, pages 243–258. Chapman & Hall, 1994.

Appendix A: Concrete Operational Semantics

The following are firing rules associated with process constructs other than those discussed in Section 2. They are extension of those presented previously by Schneider in [29]. $\alpha P \subseteq \Sigma \cup \{\checkmark\}$ is the alphabet of process P; $init(V, P)$ is the set of enabled events, as defined in [29]. In an abuse of notations, we use $\star$ to denote any event in $\Sigma \cup \{\tau, \checkmark\}$ or a real number.

$$\frac{}{(V, Stop) \xrightarrow{t} (V, Stop)}\,[\ st\ ] \qquad \frac{}{(V, Skip) \xrightarrow{\checkmark} (V, Skip)}\,[\ sk1\ ]$$

$$\frac{}{(V, Skip) \xrightarrow{t} (V, Skip)}\,[\ sk2\ ] \qquad \frac{}{(V, e \rightarrow P) \xrightarrow{t} (V, e \rightarrow P)}\,[\ as1\ ]$$

$$\frac{}{(V, e \rightarrow P) \xrightarrow{e} (e(V), P)}\,[\ as2\ ] \qquad \frac{}{(V, [b]P) \xrightarrow{t} (V, [b]P)}\,[\ gu1\ ]$$

$$\frac{V \vDash b}{(V, [b]P) \xrightarrow{\tau} (V, P)}\,[\ gu2\ ] \qquad \frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \mid Q) \xrightarrow{x} (V', P')}\,[\ ex1\ ]$$

$$\frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \mid Q) \xrightarrow{x} (V', Q')}\,[\ ex2\ ] \qquad \frac{\begin{array}{l}(V, P) \xrightarrow{t} (V, P'),\\ (V, Q) \xrightarrow{t} (V, Q')\end{array}}{(V, P \mid Q) \xrightarrow{t} (V, P' \mid Q')}\,[\ ex3\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P'), x \notin \alpha Q}{(V, P \parallel Q) \xrightarrow{x} (V', P' \parallel Q)}\,[\ pa1\ ] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q'), x \notin \alpha P}{(V, P \parallel Q) \xrightarrow{x} (V', P \parallel Q')}\,[\ pa2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V, P'), (V, Q) \xrightarrow{x} (V, Q'), x \in (\alpha P \cap \alpha Q) \cup \mathbb{R}_+}{(V, P \parallel Q) \xrightarrow{x} (V, P' \parallel Q')}\,[\ pa3\ ]$$

$$\frac{(V,P) \xrightarrow{\checkmark} (V,P')}{(V,P;\ Q) \xrightarrow{\tau} (V,Q)}\ [\ pa4\ ] \qquad \frac{(V,P) \xrightarrow{t} (V',P'), \checkmark \notin init(V,P)}{(V,P;\ Q) \xrightarrow{t} (V',P';\ Q)}\ [\ se1\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P'), \checkmark \notin init(V,P)}{(V,P;\ Q) \xrightarrow{x} (V',P';\ Q)}\ [\ se2\ ] \qquad \frac{(V,Q) \xrightarrow{\star} (V',Q'), P \mathrel{\widehat{=}} Q}{(V,P) \xrightarrow{\star} (V',Q')}\ [\ def\ ]$$

Appendix B: Abstract Operational Semantics

The following are abstract firing rules associated with process constructs other than those discussed in Section 2.

$$\frac{}{(V, Skip, D) \xhookrightarrow{\checkmark} (V, Stop, D^{\uparrow})}\ [\ aki\ ] \qquad \frac{V \vDash b}{(V,[b]P,D) \xhookrightarrow{\tau} (V,P,D^{\uparrow})}\ [\ agu\ ]$$

$$\frac{}{(V, e\{prg\} \rightarrow P, D) \xhookrightarrow{e} (prg(V), P, D^{\uparrow})}\ [\ aev\ ]$$

$$\frac{(V,P,D) \xhookrightarrow{x} (V',P',D')}{(V,P \mid Q,D) \xhookrightarrow{x} (V',P',D' \wedge \iota(V,Q,D))}\ [\ aex1\ ]$$

$$\frac{(V,Q,D) \xhookrightarrow{x} (V',Q',D)}{(V,P \mid Q,D) \xhookrightarrow{x} (V',Q',D' \wedge \iota(V,P,D))}\ [\ aex2\ ]$$

$$\frac{(V,P,D) \xhookrightarrow{e} (V',P',D'), e \notin \alpha Q}{(V,P \parallel Q,D) \xhookrightarrow{e} (V',P' \parallel Q,D' \wedge \iota(V,Q,D))}\ [\ apa1\ ]$$

$$\frac{(V,Q,D) \xhookrightarrow{e} (V',Q',D'), e \notin \alpha P}{(V,P \parallel Q,D) \xhookrightarrow{e} (V',P \parallel Q',D' \wedge \iota(V,P,D))}\ [\ apa2\ ]$$

$$\frac{(V,P,D) \xhookrightarrow{e} (V,P',D'), (V,Q,D) \xhookrightarrow{e} (V,Q',D''), e \in \alpha P \cap \alpha Q}{(V,P \parallel Q,D) \xhookrightarrow{e} (V,P' \parallel Q',D' \wedge D'')}\ [\ apa3\ ]$$

$$\frac{(V,P,D) \xhookrightarrow{x} (V',P',D'), x \neq \checkmark}{(V,P;\ Q,D) \xhookrightarrow{x} (V',P';\ Q,D' \wedge (\checkmark \notin init(V,P) \vee D))}\ [\ ase1\ ]$$

$$\frac{(V,P,D) \xhookrightarrow{\checkmark} (V',P',D')}{(V,P;\ Q,D) \xhookrightarrow{\tau} (V,Q,D \wedge D')} \qquad \frac{(V,P,D) \xhookrightarrow{x} (V',P',D'), Q \mathrel{\widehat{=}} P}{(V,Q,D) \xhookrightarrow{x} (V',P',D')}$$

Appendix C: Proof of Theorem 1

Let $\mathcal{S} = (Var, i, P)$ be the model; $\mathcal{L}_c$ and $\mathcal{L}_a$ be the concrete and abstract transition system respectively. By definition, it suffices to construct a binary relation which satisfies the condition. The theorem is proved by structural induction on the all types of process expressions. The following are the base cases.

- $Stop$: $\mathcal{R} = \{(i, Stop) \mapsto (i, Stop, true)\}$. Trivially true.
- $Skip$: $\mathcal{R} = \{(i, Skip) \mapsto (i, Skip, true), (i, Stop) \mapsto (init, Stop, true)\}$. Trivially true.
- $Wait[d]$: $\mathcal{R} = \{(i, Wait[d]) \mapsto (i, Wait[d], true), (i, Skip) \mapsto (i, Skip, true), (i, Stop) \mapsto (i, Stop, true)\}$. The transition $(i, Wait[d]) \overset{\tau}{\rightsquigarrow} (i, Skip)$ of $\mathcal{L}_c$ corresponds to the transition $(i, Waid[d], true) \overset{\tau}{\hookrightarrow} (i, Skip, true)$. Notice that the clock introduced by function $\mathcal{A}$ would be pruned by $\mathcal{D}$. The rest is trivial.

Next, we prove the induction step.

- $e \rightarrow P$: $(i, e \rightarrow P)$ and $(i, e \rightarrow P, true)$ are bi-similar since $(i, e \rightarrow P) \overset{e}{\rightsquigarrow} (prg(i), P)$ (by rule $as1$ and $as2$) and $(i, e \rightarrow P, true) \overset{e}{\hookrightarrow} (e(i), P, true)$ (by rule aev), and $(e(i), P) \sim (e(i), P, true)$ (by hypothesis).
- $[b]P$: if $i \vDash b$, then $[b]P$ behaves exactly as P (rule $gu2$ and rule agu), hence by hypothesis, $(i, [b]P) \sim (i, [b]P, true)$. If $i \nvDash b$, then $[b]P$ behaves exactly as $Stop$ (rule $gu1$ and no abstract firing rule), hence $(i, [b]P) \sim (i, [b]P, true)$.
- $P \mid Q$: $P \mid Q$ behaves either as P or Q, in both cases, by hypothesis $(i, P \mid Q) \sim (i, P \mid Q, true)$.
- $P \parallel Q$: there is one-to-one correspondence on the concrete firing rules (rule $pa1$, $pa2$ and $pa3$) and the abstract firing rules ((rule $apa1$, $apa2$ and $apa3$)). It is clear that by hypothesis $(i, P \parallel Q) \sim (i, P \parallel Q, true)$.
- $P;\ Q$. Similarly as above.
- $P\ timeout[d]\ Q$: let the associated clock be tm. We show that each abstract transition is possible if, and only if, there is a corresponding concrete transition $(i, P) \rightsquigarrow (i', P')$. Rule $ato1$ is applicable if, and only if, $tm \leq d$ and (i, P, D) may perform a τ-transition. By hypothesis, (i, P, D) may perform a τ-transition if, and only if, (i, P) does. By rule $to2$, $to3$ and $to4$, a τ of P may happen if, and only if, $tm \leq d$. Therefore, we conclude rule $ato1$ is applicable if, and only if, there is a corresponding concrete transition. Similarly, we argue that rule $ato2$ and $ato3$ are applicable if, and only if, there is a corresponding concrete transition. This concludes that $(i, P\ timeout[d]\ Q) \sim (i, P\ timeout\ Q, true)$.
- $P\ interrupt[d]\ Q$: Similarly as above.
- $P\ deadline[d]$: Similarly as above.
- $P \mathrel{\widehat{=}} Q$: By induction.