Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP[1]

Jun Sun, Yang Liu
Singapore University of Technology and Design
Jin Song Dong, Yan Liu, Ling Shi and Étienne André
National University of Singapore

Modeling and verifying complex real-time systems are challenging research problems. The *de facto* approach is based on Timed Automata, which are finite state automata equipped with clock variables. Timed Automata are deficient in modeling hierarchical complex systems. In this work, we propose a language called *Stateful Timed CSP* and an automated approach for verifying Stateful Timed CSP models. Stateful Timed CSP is based on Timed CSP and is capable of specifying hierarchical real-time systems. Through dynamic zone abstraction, *finite-state* zone graphs can be generated automatically from Stateful Timed CSP models, which are subject to model checking. Like Timed Automata, Stateful Timed CSP models suffer from Zeno runs, i.e., system runs which take infinitely many steps within finite time. Unlike Timed Automata, model checking with non-Zenoness in Stateful Timed CSP can be achieved based on the zone graphes. We extend the PAT model checker to support system modeling and verification using Stateful Timed CSP and show its usability/scalability via verification of real-world systems.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Model Checking; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems; D.2.1 [**Requirements/Specifications**]: Tools

General Terms: Algorithms, Languages, Verification

Additional Key Words and Phrases: Stateful Timed CSP, Zone Abstraction, Non-Zenoness, PAT

1. INTRODUCTION

The correctness of safety-critical computer-based systems is crucial. Real-world systems often depend on quantitative timing. Modeling and verification of real-time systems are challenging research topics which have important practical implications. The choice of language for real-time system modeling is an important factor in the success of the entire system analysis or development. The language should cover facets of the requirements and the model should reflect a system *intuitively and exactly* (up to abstraction of irrelevant details). It should have a semantic model suitable to study the behaviors of the system and to establish the validity of desired properties.

Many languages have been proposed to model real-time systems, e.g., algebra of timed processes [Nicollin and Sifakis 1994], Timed CCS [Yi 1991], Timed C-SP [Schneider 2000], etc. The most popular one is Timed Automata [Alur and

[1]This article is a significant extension of [Sun et al. 2009]. In [Sun et al. 2009], Stateful Timed CSP is proposed to model and verify (with zone abstraction) hierarchical real-time systems. This article extends [Sun et al. 2009] with complete concrete/abstract operational semantics; formal proof of all theorems; additional examples models and verification case studies. Most importantly, we solve the problem of verification with non-Zenoness assumption.

Dill 1994; Lynch and Vaandrager 1996] and its variant Timed Safety Automata [Henzinger et al. 1994]. Timed Automata are finite state automata equipped with real-valued clocks. They are powerful as they allow explicit representation of real-time through the manipulation of clock variables. Real-time constraints are captured by clock constraints on system transitions, setting or resetting clocks, etc. Verification tools for Timed Automata based models have proven to be successful, e.g., UPPAAL [Larsen et al. 1997], KRONOS [Bozga et al. 1998].

Models based on Timed Automata often have a simple structure. For instance, the input models of the popular UPPAAL checker are networks of Timed Automata with no hierarchy [Larsen et al. 1997]. While a simple structure may lead to efficient model checking, it may not be ideal as designing and verifying hierarchical complex real-time systems are becoming an increasingly urgent task due to the widespread applications and increasing complexity of real-time systems. High-level requirements for real-time systems are often stated in terms of *deadline*, *timeout*, and *timed interrupt* [Lai and Watson 1997; Dong et al. 1999; Lindahl et al. 2001]. In practice, system requirements are often structured into phases, which are then composed in many different ways. Unlike Statecharts equipped with clocks [Harel and Gery 1997] or timed process algebras [Nicollin and Sifakis 1994; Yi 1991; Schneider 2000], Timed Automata lack high-level compositional patterns for hierarchical design. Users often need to manually cast high-level requirements into a set of clock variables with carefully calculated clock constraints. This process is tedious and error-prone. On the other hand, real-time system modeling based on timed process algebras often suffers from lack of language features (e.g., shared variables) or automated tool support.

In this work, we propose an alternative approach to model and verify real-time systems. In particular, we make the following technical contributions.

—We propose a language named *Stateful Timed CSP* to model hierarchical real-time systems. Stateful Timed CSP extends Timed CSP [Schneider 2000] with language constructs to manipulate data structures and data operations in order to support real-world applications. More importantly, it supports a rich set of timed process constructs to capture timed system behavior patterns, e.g., *delay*, *deadline*, *timeout*, *timed interrupt*, etc.

—We develop a fully automatic method to model check Stateful Timed CSP models. Different from Timed Automata, Stateful Timed CSP relies on *implicit* clocks. For instance, a process which has a deadline is intuitively written as P *deadline*$[d]$. Intuitively speaking, an implicit clock starts ticking once process P is *activated* (i.e., P has the control and is ready to perform some action) and P must terminate when its reading is d. As a result, abstraction and verification techniques designed for Timed Automata are not directly applicable. Inspired by the previous work on zone abstraction [Dill 1989], we propose *dynamic zone abstraction.* The idea is to dynamically create/prune clocks (only if necessary) to capture constraints introduced by the timed process constructs. We prove that dynamic zone abstraction produces an abstract model, i.e., a zone graph, which is both finite-state and property preserving, so that it is subject to temporal logic based model checking or refinement checking.

—We develop an approach to verify Stateful Timed CSP models with the assumption of non-Zenoness, i.e., infinitely many steps always take infinite time. Zeno executions are unrealistic for obvious reasons and therefore must be ruled out during verification. In the setting of Timed Automata, it is non-trivial to decide if a run of the zone graph corresponds to a non-Zeno run [Tripakis 1999; Herbreteau et al. 2010]. We show that the zone graph in our setting can be used to decide non-Zeno runs so that we can verify Stateful Timed CSP models with the assumption of non-Zenoness.

—Lastly, we enhance the PAT model checker [Sun et al. 2009] with the techniques and show its usability via modeling of complex systems as well as automated verification of real-world and benchmark systems.

The remainder of the article is organized as follows. Section 2 presents relevant definitions. Section 3 presents the syntax and operational semantics of Stateful Timed CSP. Section 4 presents dynamic zone abstraction. Section 5 discusses how to perform model checking with the assumption of non-Zenoness. Section 6 discusses our implementation in the PAT model checker. Section 7 reviews related work. Section 8 concludes the work and discusses future research direction.

2. PRELIMINARIES

Let $\mathbb{R}_+$ denote the set of non-negative real numbers. Throughout the article, τ denotes an unobservable event; $\checkmark$ denotes the special event of process termination; $\epsilon \in \mathbb{R}_+$ denotes the event of idling for exactly ϵ time units; Σ denotes the set of observable events such that $\tau \notin \Sigma$ and $\checkmark \in \Sigma$; $\Sigma_\tau = \Sigma \cup \{\tau\}$. Furthermore, the following naming convention is adopted: $e \in \Sigma$ denotes an observable event; $a \in \Sigma_\tau$ denotes an observable event or τ; $x \in \Sigma_\tau \cup \mathbb{R}_+$.

Definition 2.1. *A labeled transition system (LTS) is a tuple $\mathcal{L} = (S, init, \Sigma_\tau, T)$ where S is a set of states; $init \in S$ is an initial state and $T : S \times \Sigma_\tau \times S$ is a labeled transition relation.*

$\mathcal{L}$ is finite if and only if S is finite. Without loss of generality, we assume that an LTS is always *reduced* so that every $s \in S$ is reachable from *init*. We write $s \xrightarrow{x} s'$ to denote $(s, x, s') \in T$ when T is clear from the context. An event a is enabled at state s if there exists s' such that $s \xrightarrow{a} s'$. State s is a deadlock state if and only if there is no enabled events at s. A run of $\mathcal{L}$ is a sequence of alternating states/events $\langle s_0, a_0, s_1, a_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all i. The run is *complete* if it is an infinite sequence or the last state in the sequence is a deadlock state. The set of complete runs of $\mathcal{L}$ is written as $runs(\mathcal{L})$.

Definition 2.2. *A timed transition system (TTS) is a tuple $\mathcal{T} = (S, init, \mathbb{R}_+ \cup \Sigma_\tau, T)$ such that S is a set of states; $init \in S$ is an initial state; $T : S \times (\mathbb{R}_+ \cup \Sigma_\tau) \times S$ is a labeled transition relation.*

There are two kinds of transitions in $\mathcal{T}$, i.e., event transitions $s \xrightarrow{a} s'$ and time transitions $s \xrightarrow{\epsilon} s'$. For simplicity, we write $s \xrightarrow{\epsilon, a} s'$ or $(s, (\epsilon, a), s') \in T$ to denote that there exists s_0 such that $s \xrightarrow{\epsilon} s_0 \xrightarrow{a} s'$. State s is a deadlock state if and only if there does not exist ϵ, a and s' such that $s \xrightarrow{\epsilon, a} s'$. A run of $\mathcal{T}$ is a sequence ρ of the form $\langle s_0, (\epsilon_0, a_0), s_1, (\epsilon_1, a_1), \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{\epsilon_i, a_i} s_{i+1}$ for all i.

The run is *complete* if it is an infinite sequence or the last state in the sequence is a deadlock state. Given ρ, we say that $\langle s_0, a_0, s_1, a_1, \cdots \rangle$ is an un-timed run of $\mathcal{T}$. The set of complete un-timed runs of $\mathcal{T}$ is written as $runs(\mathcal{T})$. In the following, we focus only on complete runs and refer them simply as runs.

DEFINITION 2.3. *A run* $\rho = \langle s_0, (\epsilon_0, a_0), s_1, (\epsilon_1, a_1), \cdots \rangle$ *is* non-Zeno *if and only if the following is satisfied.*

—*If* ρ *is infinite, then* $(\epsilon_i + \epsilon_{i+1} + \cdots)$ *for all* i *is unbounded.*

—*If* ρ *is finite, assume* s *is the last state in* ρ*, there exists* $\epsilon > n$ *such that* $s \xrightarrow{\epsilon} s'$ *for all* $n \in \mathbb{R}_+$.

A run is Zeno if and only if it is not non-Zeno. That is, a run is Zeno if and only if it contains infinitely many steps taken in a finite time interval or it reaches a deadlock state where time elapsing is bounded. For obvious reasons, Zeno runs are unrealistic. A TTS is non-empty if and only if it allows at least one non-Zeno run.

DEFINITION 2.4. *A* time-abstract bi-simulation relation *between a TTS* $\mathcal{T} = (S_t, init_t, \Sigma_\tau \times \mathbb{R}_+, T_t)$ *and an LTS* $\mathcal{L} = (S_u, init_u, \Sigma_\tau, T_u)$ *is a relation* $\mathcal{R} \subseteq S_t \times S_u$ *satisfying the following condition.*

C1: If $(s_0, s_1) \in \mathcal{R}$ *and* $(s_0, (\epsilon, a), s_0') \in T_t$ *for some* ϵ *and* a*, then there exists* s_1' *such that* $(s_1, a, s_1') \in T_u$ *and* $(s_0', s_1') \in \mathcal{R}$*;*

C2: If $(s_0, s_1) \in \mathcal{R}$ *and* $(s_1, a, s_1') \in T_u$ *for some* s_1'*, then there exists some* ϵ *and* s_0' *such that* $(s_0, (\epsilon, a), s_0') \in T_t$ *and* $(s_0', s_1') \in \mathcal{R}$*;*

C3: $(init_t, init_u) \in \mathcal{R}$.

$\mathcal{T}$ time-abstract bi-simulates $\mathcal{L}$, written as $\mathcal{T} \approx \mathcal{L}$, if and only if there exists a time-abstract bi-simulation relation between them. The following result is immediate, i.e., time-abstract bi-simulation preserves un-timed runs.

PROPOSITION 2.5. $\mathcal{T} \approx \mathcal{L} \Rightarrow runs(\mathcal{T}) = runs(\mathcal{L})$. $\square$

3. SYNTAX AND OPERATIONAL SEMANTICS

In this section, we introduce Stateful Timed CSP, which is based on Timed CSP extended with data structures as well as an enriched set of timed process constructs.

3.1 Syntax and Informal Semantics

A Stateful Timed CSP model (hereafter a model) is a tuple $\mathcal{S} = (Var, init_G, P)$ where Var is a finite set of *finite-domain* global variables, $init_G$ is the initial valuation of the variables and P is a timed process. A variable can be of a pre-defined type like Boolean, integer, array of integers or any user-defined data type[2]. Process P models the control logic of the system using a rich set of process constructs. A process can be defined by the grammar presented in Figure 1. For simplicity, we assume that P is not parameterized.

Process *Stop* does nothing but idling. Process *Skip* terminates, possibly after idling for some time. Process $e \to P$ engages in event e first and then behaves as P. Note that e may serve as a synchronization barrier, if combined with parallel

[2]Refer to PAT user manual on how to define a type in C# or Java.

$$
\begin{array}{ll}
P = Stop & \text{– in-action} \\
\quad \mid Skip & \text{– termination} \\
\quad \mid e \rightarrow P & \text{– event prefixing} \\
\quad \mid a\{program\} \rightarrow P & \text{– data operation prefixing} \\
\quad \mid \textbf{if } (b)\ \{P\} \textbf{ else } \{Q\} & \text{– conditional choice} \\
\quad \mid P \mid Q & \text{– general choice} \\
\quad \mid P \setminus X & \text{– hiding} \\
\quad \mid P;\ Q & \text{– sequential composition} \\
\quad \mid P \parallel Q & \text{– parallel composition} \\
\quad \mid Wait[d] & \text{– delay*} \\
\quad \mid P\ timeout[d]\ Q & \text{– timeout*} \\
\quad \mid P\ interrupt[d]\ Q & \text{– timed interrupt*} \\
\quad \mid P\ within[d] & \text{– timed responsiveness*} \\
\quad \mid P\ deadline[d] & \text{– deadline*} \\
\quad \mid Q & \text{– process referencing}
\end{array}
$$

Fig. 1. Process constructs

composition. In order to seamlessly integrate data operations, we allow sequential programs to be attached with events. Process $a\{program\} \rightarrow P$ performs data operation a (i.e., executing the sequential *program* whilst generating event a) and then behaves as P. The *program* may be a simple procedure updating data variables (written in the form of $a\{x := 5;\ y := 3\}$) or a complicated sequential program[3]. A conditional choice is written as **if** (b) $\{P\}$ **else** $\{Q\}$. Process $P \mid Q$ offers an (unconditional) choice between P and Q[4]. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. $P \setminus X$ hides occurrences of events in X. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via event synchronization (following CSP rules [Hoare 1985]) or shared variables. Notice that if P and Q do not communicate through event synchronization, then it is written as $P \mid\mid\mid Q$, which reads as 'P interleave Q'. A process may be given a name, written as $P \mathrel{\widehat=} Q$, and then referenced through its name. Recursion is allowed by process referencing. Additional process constructs (e.g., while or periodic behaviors) can be defined using the above.

In addition, a number of timed process constructs (marked with * in Figure 1) are designed to capture common real-time system behavior patterns. Let $d \in \mathbb{R}_+$. Process $Wait[d]$ idles for exactly d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since process $P\ timeout[d]\ Q$ is activated). Otherwise, Q takes over control after exactly d time units. In process $P\ interrupt[d]\ Q$, if P terminates before d time units, $P\ interrupt[d]\ Q$ behaves exactly as P. Otherwise, $P\ interrupt[d]\ Q$ behaves as P until d time units and then Q takes over. In contrast to $P\ timeout[d]\ Q$, P may engage in multiple observable events before it is *interrupted*. Process $P\ within[d]$ must react within d time units, i.e., an observable event must be engaged by process

[3]The detailed syntax for the sequential program can be found in PAT user manual.
[4]For simplicity, we omit external and internal choices [Hoare 1985] in the discussion.

P within d time units. Urgent event prefixing [Davies 1993], written as $e \twoheadrightarrow P$, is defined as $(e \rightarrow P)$ *within*$[0]$, i.e., e must occur as soon as it is enabled. In process P *deadline*$[d]$, P must terminate within d time units, possibly after engaging in multiple observable events. Notice that a timed process construct is always associated with an integer constant d which is referred to as its parameter.

In the following, we apply Stateful Timed CSP to model two systems so as to show that it is expressive enough to capture both benchmark real-time systems and hierarchical real-time systems.

Example Let δ and ϵ be two constants such that $\delta < \epsilon$. Fischer's mutual exclusion algorithm is modeled as a model $(V, v_i, Protocol)$. V contains two variables *turn* and *counter*. The former indicates which process attempted to access the critical section most recently. The latter counts the number of processes accessing the critical section. Initial valuation v_i maps *turn* to -1 (which denotes that no process is attempting initially) and *counter* to 0 (which denotes that no process is in the critical section initially). Process *Protocol* is defined as follows.

$$
\begin{array}{lcl}
\mathit{Protocol} & \widehat{=} & \mathit{Proc}(0) \;|||\; \mathit{Proc}(1) \;|||\; \cdots \;|||\; \mathit{Proc}(n) \\
\mathit{Proc}(i) & \widehat{=} & \textbf{if}\ (\mathit{turn} = -1)\ \{\ \mathit{Active}(i)\ \}\ \textbf{else}\ \{\ \mathit{Proc}(i)\ \} \\
\mathit{Active}(i) & \widehat{=} & (\mathit{update}.i\{\mathit{turn} := i\} \rightarrow \mathit{Wait}[\epsilon])\ \mathit{within}[\delta]; \\
& & \textbf{if}\ (\mathit{turn} = i)\ \{ \\
& & \quad \mathit{cs}.i\{\mathit{counter} := \mathit{counter} + 1\} \rightarrow \\
& & \quad \mathit{exit}.i\{\mathit{counter} := \mathit{counter} - 1;\ \mathit{turn} := -1\} \rightarrow \mathit{Proc}(i) \\
& & \}\ \textbf{else}\ \{ \\
& & \quad \mathit{Proc}(i) \\
& & \}
\end{array}
$$

where n is a constant representing the number of processes. Process $Proc(i)$ models a process with a unique integer identify i. If *turn* is -1 (i.e., no other process is attempting), $Proc(i)$ behaves as specified by process $Active(i)$. In process $Active(i)$, firstly *turn* is set to be i (indicating that the i-process is now attempting) by action $update.i$. Note that $update.i$ must occur within δ time units (captured by *within*$[\delta]$). Next, the process idles for ϵ time units (captured by *Wait*$[\epsilon]$). It then checks whether *turn* is still i. If so, it enters the critical section and leaves later. Otherwise, it restarts from the beginning.

Quantitative timing plays an important role in this algorithm to guarantee mutual exclusion, i.e., mutual exclusion is not guaranteed if $\delta \geq \epsilon$. In order to verify mutual exclusion, one way is to show that $counter \leq 1$ is always true. We remark that the event names for variable updates (e.g., $update.i$ and $cs.i$ and $exit.i$) not only improves readability but also allows an alternative way of verification, i.e., through trace refinement checking. □

Example A pacemaker is an electronic implanted device which functions to regulate the heart beat by electrically stimulating the heart to contract and thus to pump blood throughout the body. Quantitative timing is crucial to pacemaker. Common pacemakers are designed to correct bradycardia, i.e., slow heart beats. A pacemaker mainly performs two functions, i.e., sensing and pacing. Sensing is to monitor the heart's natural electrical activity, helping the pacemaker to gather information on the heart beats and react accordingly. Pacing is when a pacemaker sends electrical stimuli, i.e., tiny electrical signals, to heart through a pacing lead, which starts a heartbeat. A pacemaker can operate in many different modes according to the implanted patient's heart problem. A mode of the pacemaker is typically modeled as of the following form: $Heart \parallel Sensing \parallel Pacing$ where *Heart* models normal or abnormal heart condition; *Sensing* and *Pacing* model the two functions. In the following, we present a simplified model of the simplest mode, i.e., the sense Atrial, pace Atrial, in Trigger (AAT) mode.

The model contains one variable *SA*, which is a flag indicating whether it is necessary to monitor atria (1 for necessary). Initially, *SA* is 0. The process is *AAT* which is defined as follows.

$$
\begin{array}{ll}
AAT & \widehat{=}\ Heart \parallel Sensing \parallel Pacing(LRI) \\
Sensing & \widehat{=}\ \textbf{if}\ (SA = 1)\ \{ \\
 & \qquad pulseA \rightarrow senseA \twoheadrightarrow Sensing \\
 & \ \ \} \\
 & \ \ \textbf{else}\ \{ \\
 & \qquad pulseA \rightarrow Sensing \\
 & \ \ \} \\
Pacing(X) & \widehat{=}\ (senseA \rightarrow paceA\{SA := 0\} \rightarrow Skip)\ timeout[X]\ HelpPacing; \\
 & \ \ Wait[URI]; \\
 & \ \ (enableSA\{SA := 1\} \twoheadrightarrow Pacing(LRI - URI)) \\
HelpPacing & \widehat{=}\ (stimu \rightarrow paceA\{SA := 0\} \rightarrow Skip)\ deadline[0]
\end{array}
$$

where *URI* and *LRI* are two constants representing upper and lower rate interval, i.e., the fastest and slowest a normal heart can beat. For simplicity, we skip the details of process *Heart*. Informally speaking, process *Heart* generates two events *pulseA* (i.e., atrium does a pulse) and *pluseV* (i.e., ventricle does a pulse) periodically for a normal heart or with one of them missing once a while for an abnormal heart. Process *Sensing* monitors heart pacing by synchronizing with *Heart* on *pulseA*. If *SA* is 1, it engages in event *senseA* immediately once *pulseA* occurs. Initially, process *Pacing* awaits for event *senseA*. If *senseA* occurs before X time units, action *paceA* occurs (and *SA* is set to 0 so that sensing is paused for a while). If *senseA* is missing for X time units, timeout happens and process *HelpPacing* is invoked. *HelpPacing* models the process of the pacemaker generating an electrical stimuli (captured by event *stimu*) and then performing action *paceA*. Note that *HelpPacing* must terminate before 0 time unit (captured by $deadline[0]$), which means that it must immediately perform event *stimu* and action *paceA*. Next, $Wait[URI]$ occurs and later sensing is turned on again for the next circle.

At the top level, the pacemaker model is a choice of 16 different modes. Each mode is a parallel composition of the three components. Each component may have internally hierarchies due to complicated sensing and pacing behaviors. We skip the details (refer to [Barold et al. 2004]). The complete pacemaker model can be found at [Sun et al.]. □

3.2 Formal Operational Semantics

In order to define the operational semantics of Stateful Timed CSP, we define the notion of a configuration to capture the global system state during the system execution, which is defined as a *concrete configuration*. This terminology distinguishes the notion from the state space abstraction and abstract configurations which will be introduced later. A concrete system configuration is a pair (V, P) where V is a variable valuation function and P is a process. For simplicity, an empty valuation is written as $\varnothing$. A transition of the system is in the form $(V, P) \xrightarrow{x} (V', P')$ where $x \in \Sigma_\tau \cup \mathbb{R}_+$, i.e., a transition is labeled with an event in Σ_τ or a number in $\mathbb{R}_+$.

The operational semantics is defined systematically by associating a set of firing rules with each and every process construct. The firing rules associated with the timed process constructs are presented as examples in Figure 2.

—Rules *wait*1 and *wait*2 define the semantics of *Wait*$[d]$. Rule *wait*1 states that the process may idle for an arbitrary amount of time ϵ such that $\epsilon \leq d$. Afterwards, *Wait*$[d]$ becomes *Wait*$[d-\epsilon]$ and the variable valuation is unchanged. Rule *wait*2 states that the process becomes *Skip* via a τ-transition whenever d is 0.

—Rules *to*1 to *to*4 define P *timeout*$[d]$ Q. Rule *to*1 states that if an observable event e can be engaged by P, changing (V, P) to (V', P'), then $(V, P$ *timeout*$[d]$ $Q)$ becomes (V', P') so that Q is discharged. That is, P has performed an observable event before timeout occurs. Rule *to*2 states that if d is 0, Q takes over control by a τ-transition. Rule *to*3 states that if P performs a τ-transition, then Q and *timeout* operator remain (since an observable event is yet to be performed). Rule *to*4 states that if P may idle for less than or equal to d time units, so does P *timeout*$[d]$ Q.

—Rules *ti*1 to *ti*4 define P *interrupt*$[d]$ Q. Rule *ti*1 states that if event a (which may be observable or τ, but not $\checkmark$) can be engaged by P, changing (V, P) to (V', P'), then $(V, P$ *interrupt*$[d]$ $Q)$ can perform a as well. In contrast to rule *to*1, the *interrupt* operator remains. Intuitively, it states that before P is interrupted, P behaves freely. Rule *ti*2 states that if P may idle for less than or equal to d time units, so does P *interrupt*$[d]$ Q. Rule *ti*3 states that if P terminates before being interrupted, then the whole process terminates. Rule *ti*4 states that if d is 0, Q takes over control by a τ-transition.

—Rules *wi*1 to *wi*3 define P *within*$[d]$. Rule *wi*1 states that if an observable event e occurs, then *within* is discharged, as the requirement is fulfilled. In contrast, rule *wi*2 states that if instead event τ occurs, then *within* remains. Rule *wi*3 state if P can idle for ϵ time units, so does P *within*$[d]$ as long as $\epsilon \leq d$.

—Rules *dl*1, *dl*2 and *dl*3 define P *deadline*$[d]$. P *deadline*$[d]$ requires P to terminate (marked by $\checkmark$) before d time units. Rule *dl*1 states that P can do whatever it can before the deadline is expired. Rule *dl*2 states that if P terminates, then *deadline* is discharged. Rule *dl*3 states if P can idle for ϵ time units, so does P *deadline*$[d]$ as long as $\epsilon \leq d$.

The rest of the rules are similarly defined (refer to [Sun et al. 2009]). We remark the rules are an extension of the operational semantics in [Schneider 1995]. While the rules in [Schneider 1995] are designed for Timed CSP, our rules handle data states as well as timed process constructs like *within* and *deadline*.

The following can be established immediately.

PROPOSITION 3.1. *(1) If* $(V, P) \xrightarrow{\epsilon} (V', P')$*, then* $V' = V$. *(2) If* $(V, P) \xrightarrow{\epsilon_0} (V', P')$ *and* $(V', P') \xrightarrow{\epsilon_1} (V'', P'')$*, then* $(V, P) \xrightarrow{\epsilon_0 + \epsilon_1} (V'', P'')$. □

$$\frac{\epsilon \leq d}{(V, Wait[d]) \xrightarrow{\epsilon} (V, Wait[d-\epsilon])}\ [\ wait1\ ] \qquad \frac{}{(V, Wait[0]) \xrightarrow{\tau} (V, Skip)}\ [\ wait2\ ]$$

$$\frac{(V, P) \xrightarrow{e} (V', P')}{(V, P\ timeout[d]\ Q) \xrightarrow{e} (V', P')}\ [\ to1\ ] \qquad \frac{}{(V, P\ timeout[0]\ Q) \xrightarrow{\tau} (V, Q)}\ [\ to2\ ]$$

$$\frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P\ timeout[d]\ Q) \xrightarrow{\tau} (V', P'\ timeout[d]\ Q)}\ [\ to3\ ]$$

$$\frac{(V, P) \xrightarrow{\epsilon} (V, P'), \epsilon \leq d}{(V, P\ timeout[d]\ Q) \xrightarrow{\epsilon} (V, P'\ timeout[d-\epsilon]\ Q)}\ [\ to4\ ]$$

$$\frac{(V, P) \xrightarrow{a} (V', P'), a \neq \checkmark}{(V, P\ interrupt[d]\ Q) \xrightarrow{a} (V', P'\ interrupt[d]\ Q)}\ [\ ti1\ ]$$

$$\frac{(V, P) \xrightarrow{\epsilon} (V, P'), \epsilon \leq d}{(V, P\ interrupt[d]\ Q) \xrightarrow{\epsilon} (V, P'\ interrupt[d-\epsilon]\ Q)}\ [\ ti2\ ]$$

$$\frac{(V, P) \xrightarrow{\checkmark} (V', P')}{(V, P\ interrupt[d]\ Q) \xrightarrow{\checkmark} (V', P')}\ [\ ti3\ ]$$

$$\frac{}{(V, P\ interrupt[0]\ Q) \xrightarrow{\tau} (V, Q)}\ [\ ti4\ ] \qquad \frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P\ within[d]) \xrightarrow{\tau} (V', P'\ within[d])}\ [\ wi2\ ]$$

$$\frac{(V, P) \xrightarrow{e} (V', P')}{(V, P\ within[d]) \xrightarrow{e} (V', P')}\ [\ wi1\ ] \qquad \frac{(V, P) \xrightarrow{\epsilon} (V, P'), \epsilon \leq d}{(V, P\ within[d]) \xrightarrow{\epsilon} (V, P'\ within[d-\epsilon])}\ [\ wi3\ ]$$

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

$$\frac{(V, P) \xrightarrow{\checkmark} (V', P')}{(V, P\ deadline[d]) \xrightarrow{\checkmark} (V', P')}\ [\ dl2\ ]$$

$$\frac{(V, P) \xrightarrow{\epsilon} (V, P'), \epsilon \leq d}{(V, P\ deadline[d]) \xrightarrow{\epsilon} (V, P'\ deadline[d-\epsilon])}\ [\ dl3\ ]$$

Fig. 2. Concrete firing rules where $e \in \Sigma$ and $a \in \Sigma_\tau$

Intuitively speaking, (1) states that time transitions do not modify variables and (2) states that consecutive time transitions can be accumulated.

Definition 3.2. *Let $\mathcal{S} = (Var, init_G, P)$ be a model. The concrete semantics of $\mathcal{S}$, denoted as $\mathcal{T}_\mathcal{S}$, is a TTS $(S, init, \Sigma \cup \mathbb{R}_+, T)$ such that S is a set of reachable concrete system configurations; $init = (init_G, P)$ is the initial configuration; and T satisfies $((V, P), x, (V', P')) \in T$ if and only if $(V, P) \xrightarrow{x} (V', P')$.*

4. DYNAMIC ZONE ABSTRACTION

$\mathcal{T}_\mathcal{S}$ always has infinitely many states, even when all variables have finite domains. For instance, assume $\mathcal{S} = (\varnothing, true, P)$ where there is no variable and P is defined as $P \mathrel{\widehat{=}} (a \rightarrow (P \mid c \rightarrow Skip)); (b \rightarrow Stop)$, it can be shown that the set of traces $trace(\mathcal{T}_\mathcal{S})$ constitutes an irregular language [Hoare 1985]. We thus restrict ourselves to a subset of models, which we refer as regular Stateful Timed CSP models. *A Stateful Timed CSP model $\mathcal{S}$ is regular if and only if P is a process expression constituted by finitely many process constructs*, for every reachable configuration (V, P) of $\mathcal{T}_\mathcal{S}$[5]. Given a regular model $\mathcal{S}$, there may still be infinitely many states in $\mathcal{T}_\mathcal{S}$ because parameters of timed process constructs in a process (e.g., d in $Wait[d]$) can take infinitely many different values. Intuitively, the constants capture the reading of the implicit clocks associated with the processes. In the following, we abstract the exact value of the constants by dynamic zone abstraction so as to generate a finite-state abstraction.

4.1 From Implicit Clocks to Explicit Clocks

In Stateful Timed CSP, clocks are implicitly associated with timed process constructs. A clock starts ticking once a timed process becomes activated. Before applying zone abstraction, we associate clocks with time processes explicitly so as to differentiate parameters associated with different timed process constructs. In theory, each timed process construct is associated with a unique clock. Nonetheless, multiple timed processes may be activated at the same time during system execution and therefore can be associated with the same clock. For instance, assume that a process P is defined as: $P \mathrel{\widehat{=}} (Wait[5];\ Wait[3])\ interrupt[6]\ Q$. There are three implicit clocks, one associated with $Wait[5]$ (say t_1), one with $Wait[3]$ (say t_2) and one with P (because of $interrupt[6]$, say t_3). Because $Wait[5]$ and P are activated, clock t_1 and t_3 have started. In contrast, clock t_2 starts only when $Wait[5]$ terminates. It is obvious that t_1 and t_3 always have the same reading and thus one clock is sufficient. It is known that the fewer clocks, the more efficient real-time model checking could be [Bengtsson and Yi 2003]. In order to minimize the number of clocks, clocks are introduced at runtime and are shared by as many processes as possible. In the following, we show how to systematically associate clocks with timed processes. Intuitively, a clock is introduced if and only

[5]This definition adopts the the idea of *finite-state processes* for Timed CSP as defined in [Ouaknine and Worrell 2002]. Formally, a Timed CSP process is a finite-state process if there are only finitely many states reachable via transitions labeled with events or 1 (i.e., a time transition which takes 1 time unit) [Ouaknine and Worrell 2002]. It is possible to extend their definition to the setting of Stateful Timed CSP. Nonetheless, formally establishing that a 'finite-state process' can only reach finite process expressions is tedious and not the focus of this work.

$$
\begin{array}{lll}
\mathcal{A}(Stop, t) & = Stop & \text{– A1} \\
\mathcal{A}(Skip, t) & = Skip & \text{– A2} \\
\mathcal{A}(e \rightarrow P, t) & = e \rightarrow P & \text{– A3} \\
\mathcal{A}(a\{program\} \rightarrow P, t) & = a\{program\} \rightarrow P & \text{– A4} \\
\mathcal{A}(\textbf{if } (b) \ \{P\} \textbf{ else } \{Q\}, t) & = \textbf{if } (b) \ \{P\} \textbf{ else } \{Q\} & \text{– A5} \\
\mathcal{A}(Wait[d], t) & = Wait[d]_t & \text{– A6} \\
\mathcal{A}(P \ timeout[d] \ Q, t) & = \mathcal{A}(P, t) \ timeout[d]_t \ Q & \text{– A7} \\
\mathcal{A}(P \ interrupt[d] \ Q, t) & = \mathcal{A}(P, t) \ interrupt[d]_t \ Q & \text{– A8} \\
\mathcal{A}(P \ within[d], t) & = \mathcal{A}(P, t) \ within[d]_t & \text{– A9} \\
\mathcal{A}(P \ deadline[d], t) & = \mathcal{A}(P, t) \ deadline[d]_t & \text{– A10} \\
\mathcal{A}(Wait[d]_{t'}, t) & = Wait[d]_{t'} & \text{– A11} \\
\mathcal{A}(P \ timeout[d]_{t'} \ Q, t) & = \mathcal{A}(P, t) \ timeout[d]_{t'} \ Q & \text{– A12} \\
\mathcal{A}(P \ interrupt[d]_{t'} \ Q, t) & = \mathcal{A}(P, t) \ interrupt[d]_{t'} \ Q & \text{– A13} \\
\mathcal{A}(P \ within[d]_{t'}, t) & = \mathcal{A}(P, t) \ within[d]_{t'} & \text{– A14} \\
\mathcal{A}(P \ deadline[d]_{t'}, t) & = \mathcal{A}(P, t) \ deadline[d]_{t'} & \text{– A15} \\
\mathcal{A}(P \mid Q, t) & = \mathcal{A}(P, t) \mid \mathcal{A}(Q, t) & \text{– A16} \\
\mathcal{A}(P \setminus X, t) & = \mathcal{A}(P, t) \setminus X & \text{– A17} \\
\mathcal{A}(P; \ Q, t) & = \mathcal{A}(P, t); \ Q & \text{– A18} \\
\mathcal{A}(P \parallel Q, t) & = \mathcal{A}(P, t) \parallel \mathcal{A}(Q, t) & \text{– A19} \\
\mathcal{A}(P, t) & = \mathcal{A}(Q, t) \qquad \text{if } P \mathrel{\widehat{=}} Q & \text{– A20}
\end{array}
$$

Fig. 3. Clock activation

if one or more timed processes have *just* become activated. Let $\mathcal{Q}$ denote the set of processes associated with explicit clocks. For simplicity, we write $Wait[d]_t$ (or $P \ timeout[d]_t \ Q$, $P \ interrupt[d]_t \ Q$, $P \ within[d]_t$, $P \ deadline[d]_t$) to denote that the process is associated with clock t. Given a process P and a clock t, we define function $\mathcal{A}$ to return the corresponding process in $\mathcal{Q}$. Figure 3 presents the detailed definition. Intuitively speaking, A1 to A5 state that if a process is un-timed and none of its sub-processes is activated, then it is unchanged. A6 to A10 state that if a process is timed, then it is associated with t and function $\mathcal{A}$ is applied to its activated sub-processes at the same time. Note that if a timed process has already been associated with a clock t', then it will not be associated with the new clock. This is captured by A11-A15, where $Wait[d]_{t'}$ denotes that $Wait[d]$ is associated with clock t'. If a sub-process is activated, then function $\mathcal{A}$ is applied recursively, as captured by A7-10,12-19. The last rule A20 states that if P is defined as Q, then $\mathcal{A}(P, t)$ can be obtained by applying $\mathcal{A}$ to Q.

Given a process $P \in \mathcal{Q}$, we can obtain the set of clocks associated with P or any sub-process of P. For instance, the clocks associated with $P \ timeout[d]_t Q$ contain t and the clocks associated with P. Notice that there is no clock associated with Q because it is not activated. The set is written as $cl(P)$.

4.2 Zones

The concrete firing rules presented in Figure 2 capture quantitative timing through parameters of the timed processes. Inevitably, there are infinitely many constant values. In the setting of Timed Automata, it has been shown that zone abstraction allows efficient model checking [Dill 1989; Behrmann et al. 1999; Bengtsson and Yi 2003]. Zone abstraction for Timed Automata, however, cannot be readily adopted due to the difference between Stateful Timed CSP and Timed Automata. In the following, we review necessary background on zones and zone operations before presenting how to apply zone abstraction to Stateful Timed CSP models.

A zone is the conjunction of multiple primitive constraints over a set of clocks. A primitive constraint is of the form $t \sim d$ where t is a clock, d is a constant and $\sim$ is either, $\geq$, $=$ or $\leq$. Intuitively, a zone is the maximal set of clock valuations satisfying the constraint. Given a clock valuation v, we write $v \in D$ to denote that v is in zone D. A zone is empty if and only if the constraint is unsatisfiable. We write $cl(D)$ to denote the clocks of D. A zone can be equivalently represented as a DBM (short for Difference Bound Matrices [Dill 1989; Behrmann et al. 1999]). Let $t_1, t_2, \cdots, t_n$ denote n clocks and t_0 denote 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. Entry (i, j) in the matrix, denoted by D^i_j, represents the upper bound on difference between clock t_i and t_j, i.e., $t_i - t_j \leq D^i_j$. A DBM thus represents the constraint: $t_i - t_j \leq D^i_j$ for all clock t_i and t_j such that $0 \leq i \leq n$ and $0 \leq j \leq n$. The bound on difference between t_i and t_j is captured by: $-D^j_i \leq t_i - t_j \leq D^i_j$. Because t_0 is always 0, we have $-D^0_i \leq t_i \leq D^i_0$ which is the bounds of clock t_i.

In the following, we briefly introduce the relevant zone operations/properties and its corresponding DBM implementation. Interested readers are referred to [Dill 1989; Behrmann et al. 1999; Bengtsson and Yi 2003] for details.

—*Calculate canonical form:* In theory, there are infinitely many different timing constraints representing the same zone. For instance, the clock valuations for $0 \leq t_1 \leq 3 \wedge 0 \leq t_1 - t_2 \leq 3$ and $0 \leq t_1 \leq 3 \wedge 0 \leq t_1 - t_2 \leq 3 \wedge t_2 \leq 1000$ are exactly the same and hence they represent the same zone. Zones represented as DBMs can be systematically compared if they are in their *canonical* forms. A DBM is in its canonical form if and only if every entry D^i_j is the tightest bound on difference between clock t_i and t_j. An important property of DBM is that there is a relatively efficient procedure to compute a unique canonical form. 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. Floyd-Warshall algorithm [Floyd 1962] thus can be used to compute the tightest bound on clock differences and hence the canonical form. The complexity of Floyd-Warshall algorithm is cubic in the number of clocks.

—*Check satisfiability:* It is essential to check whether a zone is empty or not. A zone is empty if and only if its DBM representation, in its canonical form, contains an entry D^i_i such that $D^i_i < 0$. Intuitively, it means that clock t_i is constrained to satisfy $t_i - t_i < 0$, which is impossible. Furthermore, it can be shown that a DBM in its canonical form represents an empty zone if and only if D^0_0 is negative.

—*Add clocks:* Clocks may be introduced during system exploration as we have shown in Section 4.1. Assume that the clock to be added is t_k and the given DBM is in its canonical form. Figure 4 shows how the DBM is updated with entries for t_k. For all i, D^i_k is set to be D^i_0 and D^k_i is set to be D^0_i. Because t_k

	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}	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^{k-1}}$
$\mathbf{t_k}$	$\mathbf{0}$	$\mathbf{d_1^0}$	$\cdots$	$\mathbf{d_i^0}$	$\cdots$	$\mathbf{d_{k-1}^0}$	$\mathbf{0}$

Fig. 4. Add clock

is a newly introduced clock, it must be equivalent to t_0. The resultant DBM is canonical if the given DBM is.

—*Prune clocks:* In our setting, clocks may be pruned. Because entries in a canonical DBM represent the tightest bounds on clock differences, pruning a clock t_i is simply to remove the i-row and i-column in the matrix. The remaining DBM is canonical, i.e., the bounds can not be possibly tightened with less constraints. Given a DBM D and a set of clocks C, we write $D[C]$ to denote the DBM obtained by pruning all clocks other than those in C. In an abuse of notation, we write $D[t]$ to denote the constraint on t.

—*Delay:* Given a zone D, $D^\uparrow$ denotes the zone obtained by delaying for an arbitrary amount of time. $D^\uparrow$ is obtained by changing D_0^i to ∞ for all i such that $i \geq 1$.

4.3 Abstraction

In the following, we present dynamic zone abstraction for Stateful Timed CSP, which was initially proposed in [Sun et al. 2009]. Firstly, we define the notion of abstract system configurations.

Definition 4.1. *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.*

In order to systematically apply zone abstraction, we define a set of *abstract* firing rules. The abstract firing rules eliminate concrete ϵ-transitions all together and use zones to ensure a process behaves correctly with respect to timing requirements. To distinguish from concrete firing rules, an abstract firing rule is written in the the form of $(V, P, D) \overset{x}{\rightsquigarrow} (V', P', D')$ where $x \in \Sigma_\tau$.

We first define a function *idle* which, given a process in $\mathcal{Q}$, returns the zone in which the process can idle. Figure 5 shows the detailed definition. Rules *idle*1 to *idle*5 state that if the process is un-timed and none of its sub-processes is activated, then function *idle* returns true, which means that the process may idle for an arbitrary amount of time. Rules *idle*6 to *idle*9 state that if sub-processes of the process are activated, then function *idle* is applied to the sub-processes. For instance, if the process is a choice (rule *idle*6) or a parallel composition (rule *idle*9) of P and Q, then the result is $idle(P) \wedge idle(Q)$. Intuitively, this means that process $P \mid Q$ (or $P \parallel Q$) may idle as long as both P and Q can idle. Rules *idle*10 to *idle*14 define the cases when the process is timed. For instance, process $Wait[d]_t$ may idle as long as t is less or equal to d. Lastly, *idle*15 defines the case for process referencing.

$$
\begin{array}{lll}
idle(Stop) & = true & \text{– rule } idle1 \\
idle(Skip) & = true & \text{– rule } idle2 \\
idle(e \rightarrow P) & = true & \text{– rule } idle3 \\
idle(a\{program\} \rightarrow P) & = true & \text{– rule } idle4 \\
idle(\textbf{if } (b)\ \{P\} \textbf{ else } \{Q\}) & = true & \text{– rule } idle5 \\
idle(P \mid Q) & = idle(P) \wedge idle(Q) & \text{– rule } idle6 \\
idle(P \setminus X) & = idle(P) & \text{– rule } idle7 \\
idle(P;\ Q) & = idle(P) & \text{– rule } idle8 \\
idle(P \parallel Q) & = idle(P) \wedge idle(Q) & \text{– rule } idle9 \\
idle(Wait[d]_t) & = t \leq d & \text{– rule } idle10 \\
idle(P\ timeout[d]_t\ Q) & = t \leq d \wedge idle(P) & \text{– rule } idle11 \\
idle(P\ interrupt[d]_t\ Q) & = t \leq d \wedge idle(P) & \text{– rule } idle12 \\
idle(P\ within[d]_t) & = t \leq d \wedge idle(P) & \text{– rule } idle13 \\
idle(P\ deadline[d]_t) & = t \leq d \wedge idle(P) & \text{– rule } idle14 \\
idle(P) & = idle(Q) \qquad \text{if } P \mathrel{\widehat{=}} Q & \text{– rule } idle15
\end{array}
$$

Fig. 5. Idling calculation

Figure 6 then exemplifies the abstract firing rules for the timed processes. The rest of the rules are similarly defined (refer to [Sun et al. 2009]).

—Rule *await* defines the abstract semantics of $Wait[d]$. In contrast to the concrete semantics, there is only one abstract rule. It states that a τ-transition occurs exactly when clock t reads d. Intuitively, $D^{\uparrow} \wedge t = d$ denotes the exact moment when t reads d. Afterwards, the process becomes *Skip*.

—Rules *ato*1, *ato*2 and *ato*3 define the abstract semantics of $P\ timeout[d]\ Q$. Rule *ato*1 states that if a τ-transition transforms (V, P, D) to (V', P', D'), then a τ-transition may occur given $(V, P\ timeout[d]_t\ Q, D)$ if zone $D^{\uparrow} \wedge D' \wedge t \leq d$ is not empty. Intuitively, this means that the τ-transition must occur before timeout occurs. Similarly, rule *ato*2 ensures that the occurrence of an observable event e from process P occurs only before timeout occurs. Rule *ato*3 states that timeout results in a τ-transition when the reading of t is d. Constraint $D^{\uparrow} \wedge t = d \wedge idle(P)$ ensures that process P may idle until timeout occurs.

—Rules *ait*1, *ait*2 and *ait*3 define the abstract semantics of $P\ interrupt[d]\ Q$. Rule *ait*1 states that a transition (other than process termination) originated from P may occur only if $t \leq d$, i.e., before interrupt occurs. Rule *ait*2 states that interrupt results in a τ-transition when the reading of t is d. Rule *ait*3 states that if P terminates before interrupt occurs, then the whole process terminates.

—Rules *awi*1 and *awi*2 define the abstract semantics of $P\ within[d]$. Rule *awi*1 states that if a τ-transition occurs within d time units, then the resultant process is of the form $P'\ within[d]$, which means that it is yet to perform some observable event before d time units. Rule *awi*2 states that once an observable event occurs, the *within* construct is removed.

—Rules *adl*1 and *adl*2 define the abstract semantics of $P\ deadline[d]$. Rule *adl*1 ensures that all transitions of P must occur within d time units. Rule *adl*2 states that if P terminates (by engaging in ✓), then *deadline* is removed.

$$\frac{}{(V, Wait[d]_t, D) \overset{\tau}{\rightsquigarrow} (V, Skip, D^{\uparrow} \wedge t = d)} \; [\,await\,]$$

$$\frac{(V, P, D) \overset{\tau}{\rightsquigarrow} (V', P', D')}{(V, P\ timeout[d]_t\ Q, D) \overset{\tau}{\rightsquigarrow} (V', P'\ timeout[d]_t\ Q, D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,ato1\,]$$

$$\frac{(V, P, D) \overset{e}{\rightsquigarrow} (V', P', D')}{(V, P\ timeout[d]_t\ Q, D) \overset{e}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,ato2\,]$$

$$\frac{}{(V, P\ timeout[d]_t\ Q, D) \overset{\tau}{\rightsquigarrow} (V, Q, D^{\uparrow} \wedge t = d \wedge idle(P))} \; [\,ato3\,]$$

$$\frac{(V, P, D) \overset{a}{\rightsquigarrow} (V', P', D'), a \neq \checkmark}{(V, P\ interrupt[d]_t\ Q, D) \overset{a}{\rightsquigarrow} (V', P'\ interrupt[d]_t\ Q, D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,ait1\,]$$

$$\frac{}{(V, P\ interrupt[d]_t\ Q, D) \overset{\tau}{\rightsquigarrow} (V, Q, D^{\uparrow} \wedge t = d \wedge idle(P))} \; [\,ait2\,]$$

$$\frac{(V, P, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D')}{(V, P\ interrupt[d]_t\ Q, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,ait3\,]$$

$$\frac{(V, P, D) \overset{\tau}{\rightsquigarrow} (V', P', D')}{(V, P\ within[d]_t, D) \overset{\tau}{\rightsquigarrow} (V', P'\ within[d]_t, D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,awi1\,]$$

$$\frac{(V, P, D) \overset{e}{\rightsquigarrow} (V', P', D')}{(V, P\ within[d]_t, D) \overset{e}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,awi2\,]$$

$$\frac{(V, P, D) \overset{a}{\rightsquigarrow} (V', P', D'), a \neq \checkmark}{(V, P\ deadline[d]_t, D) \overset{a}{\rightsquigarrow} (V', P'\ deadline[d]_t, D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,adl1\,]$$

$$\frac{(V, P, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D')}{(V, P\ deadline[d]_t, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge t \leq d)} \; [\,adl2\,]$$

Fig. 6. Abstract Firing Rules

Using the abstract firing rules, we can generate an abstract LTS which captures the abstract semantics of a model.

DEFINITION 4.2. *Let $\langle t_1, \cdots \rangle$ be a sequence of clocks. Let $\mathcal{S} = (Var, init_G, P)$ be a model. The time-abstract semantics of $\mathcal{S}$, denoted as $\mathcal{L}_{\mathcal{S}}$, is an LTS $(S, init, \Sigma_\tau, T)$ such that S is a set of valid abstract system configurations; $init = (init_G, P, true)$ is the initial abstract configuration and T is the smallest transition relation such that: for all $(V, P, D) \in S$, if t is the first clock in the sequence which is not in $cl(P)$, and if $(V, \mathcal{A}(P, t), D \wedge t = 0) \overset{a}{\rightsquigarrow} (V', P', D')$, $((V, P, D), a, (V', P', D'[cl(P')])) \in T$.*

Because of zone abstraction, $\mathcal{L}_{\mathcal{S}}$ is also referred to as a zone graph. Informally, $\mathcal{L}_{\mathcal{S}}$ is constructed as follows. Given an abstract configuration (V, P, D), firstly, a clock t

which is not currently associated with P is picked. The abstract configuration (V, P, D) is transformed to $(V, \mathcal{A}(P, t), D \wedge t = 0)$, i.e., timed processes which just become activated are associated with t and D is conjuncted with $t = 0$. Then, an abstract firing rule is applied to get a target configuration (V', P', D') such that D' must not be empty (otherwise, the transition is infeasible). Lastly, clocks which are not in $cl(P')$ are pruned from D' since those clocks are irrelevant to the behavior of P'. Note that for all $(V, P, D) \in S$, $cl(P) = cl(D)$. The construction of $\mathcal{L}_\mathcal{S}$ is illustrated in the following example.

Example Assume that a model $\mathcal{S} = (\varnothing, true, P)$ such that

$$P \mathrel{\widehat{=}} (a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]\ c \rightarrow P$$

Intuitively, event b never occurs because interrupt always occurs first. The left part of Figure 7 shows the $\mathcal{L}_\mathcal{S}$ (the right part depicts the equivalent model under the form of a Timed Automaton, which will be explained in Section 5). Notice that transitions are labeled with the clock which is associated with the just activated timed processes, an event and a set of clocks which are pruned from the zone after the transition. The initial configuration is $c_0 = (\varnothing, P, true)$.

—Starting with c_0, we apply $\mathcal{A}$ to P with t_1 to get

$$c_1 = (\varnothing, (a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow P, t_1 = 0)$$

Next, we can apply either rule $ait1$ or $ait2$. Apply rule $ait1$, we get

$$c_2 = (\varnothing, (Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow P, 0 \leq t_1 \leq 3)$$

Applying rule $ait2$ to c_1, we get $c_3 = (\varnothing, c \rightarrow P, t_1 = 3)$. Note that clock t_1 is irrelevant after the transition. After pruning t_1, we get $c_4 = (\varnothing, c \rightarrow P, true)$.

—Starting with c_2, we apply $\mathcal{A}$ to $(Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow P$ with t_2 to get

$$c_5 = (\varnothing, (Wait[5]_{t_2};\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow P, 0 \leq t_1 \leq 3 \wedge t_2 = 0)$$

Next, we can apply rule $ait1$ or $ait2$. Applying rule $ait1$ to c_5, we get zone $(0 \leq t_1 \leq 3 \wedge t_2 = 0)^{\uparrow} \wedge 0 \leq t_1 \leq 3 \wedge t_2 = 5)$. By DBM operations, this zone can be shown to be empty and therefore this transition is invalid. Intuitively, this is because $(0 \leq t_1 \leq 3 \wedge t_2 = 0)^{\uparrow}$ is equivalent to $0 \leq t_1 - t_2 \leq 3$. Apply rule $ait2$ to c_5, we get

$$c_7 = (\varnothing, c \rightarrow P, t_1 \geq 0 \wedge t_2 \geq 0 \wedge t_2 \leq 5 \wedge t_1 = 3)$$

Note that both clocks are irrelevant and therefore can be pruned. The resultant configuration is c_4.

—Starting with c_4, apply the rule for event prefixing, i.e., c may occur at any time in the future (refer to rule*aev* in [Sun et al. 2009]) to obtain the transition to c_0. *Note that t_1 is available and thus reused.* □

4.4 Stateful Timed CSP vs. Timed Automata

An obvious question is: what is the relationship between Stateful Timed CSP and Timed Automata? In the following, we show that Stateful Timed CSP is equivalent to Closed Timed Safety Automata with τ-transitions. In the original theory of

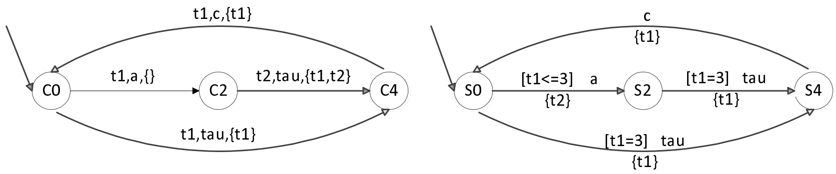

Fig. 7. An abstract LTS and its Timed Automaton equivalent

Timed Automata [Alur and Dill 1994], a Timed Automaton is a finite-state Büchi automaton extended with clocks. Büchi accepting conditions are used to enforce progress properties. Timed Safety Automata was introduced in [HNSY94] to specify progress properties using local invariant conditions instead. In the following, we focus on Timed Safety Automata and refer them simply as Timed Automata following the literature. A Timed Automaton is *closed* if it has only closed invariant and enabling clock constraints.

In [Ouaknine and Worrell 2002], it has been shown that finite-state Timed CSP processes are equivalent to closed Timed Automata with τ-transitions. Because Stateful Timed CSP is an extension of Timed CSP, it thus implies that Stateful Timed CSP is at least as expressive as closed Timed Automata. Stateful Timed CSP extends Timed CSP in two ways: shared variables and process constructs *within* and *deadline*. Firstly, it has long been known (see [Hoare 1985] and [Roscoe 2001], for example) that one can model a finite domain variable as a finite-state process parallel to the one that uses it. The user processes then read from, or write to, the variable by event synchronization. Secondly, it can be shown that *deadline* and *within* can be translated to state invariants in Timed (Safety) Automata. For instance, we have shown in [Dong et al. 2008] that *deadline* can be captured using clocks and state invariants, i.e., if a process must terminate before d, then every configuration before the process terminates is labeled with invariant $x \leq d$ where x is clock which starts when P is activated. This implies that regular Stateful Timed CSP is equivalent to closed Timed Automata with τ-transitions.

This result does not imply that Stateful Time CSP is not useful. Stateful Timed CSP has advantages over Timed CSP as it offers ease of modeling with the 'syntactic sugars'. Furthermore, there are useful properties about Stateful Timed CSP which are not satisfied by Timed Automata in general. Firstly, it can be shown that every clock is bounded from above in Stateful Timed CSP (see the definition of *idle* in Figure 5 and abstract firing rules in Figure 6), which implies that unlike Timed Automata, zone normalization [Rokichi 1993] is not essential in our setting. Secondly, the number of clocks used in our graph is often less than that of the corresponding Timed Automaton model, as shown in Section 6. Lastly, unlike Timed Automata, model checking with non-Zenoness based on the zone graphs is feasible, as we show in the following section.

5. MODEL CHECKING WITH NON-ZENONESS

In this section, we show that Stateful Timed CSP models can be model checked based on the abstract semantics. In order to apply model checking techniques, we first establish that $\mathcal{L}_{\mathcal{S}}$ is finite given any model $\mathcal{S}$.

THEOREM 5.1. *$\mathcal{L}_{\mathcal{S}}$ is finite for any regular model $\mathcal{S}$.*

Proof By definition, $\mathcal{L}_{\mathcal{S}}$ is finite if and only if there are only finitely many abstract configurations. The number of abstract configurations is bounded by $\#V \times \#P \times \#D$ where $\#V$ denotes the number of variable valuations; $\#P$ denotes the number of processes; and $\#D$ denotes the number of zones. We show all of them are finite.

—$\#V$ is finite: All variables have finite domains by assumption.

—$\#P$ is finite: Notice that P is constituted by process names, events, the associated clocks and parameters of the timed process constructs. Because processes are not parameterized[6], process names and events are finite. By assumption, every reachable process is constituted by only finitely many process constructs. Because clocks are associated with timed process constructs, it implies that for every abstract configuration (V, P, D), $cl(P)$ is finite. By reusing clocks (as in Definition 4.2), it implies that only finitely many clocks are necessary. Lastly, notice that all abstract firing rules preserve parameters of timed process constructs and therefore the possible values for parameters is finite. Finally, $\#P$ is finite.

—$\#D$ is finite: It is straightforward to show that every clock is bounded from above. It implies that every entry of D (in its canonical form) is bounded. Further, every entry of D is an integer constant and therefore $\#D$ must be finite. □

The next theorem shows that $\mathcal{L}_{\mathcal{S}}$ preserves a large class of interesting properties.

THEOREM 5.2. *$\mathcal{T}_{\mathcal{S}}$ time-abstract bi-simulates $\mathcal{L}_{\mathcal{S}}$ for any model $\mathcal{S}$.*

Proof Let $\mathcal{L}_{\mathcal{S}} = (S_a, init_a, \Sigma_\tau, T_a)$ and $\mathcal{T}_{\mathcal{S}} = (S_c, init_c, \mathbb{R}_+ \cup \Sigma_\tau, T_c)$. By definition, we need to find a time abstract bi-simulation relation $\mathcal{R}$ between S_a and S_c. We define $\mathcal{R}$ as follows. For all $(V_c, P_c) \in S_c$ and $(V_a, P_a, D) \in S_a$, $((V_c, P_c), (V_a, P_a, D)) \in \mathcal{R}$ if and only if $V_c = V_a$ and P_c is abstracted by P_a with D. P_c is abstracted by P_a with D if and only if the following two conditions are satisfied.

—P_c differs from P_a only by the parameters of the timed process constructs and the fact that P_a is associated with clocks, whereas P_c is not.

—For every timed process construct of P_c, let d be the associated parameter; let d' be the constant associated with the corresponding construct in P_a. If the construct is not associated with a clock in P, then $d = d'$. If the construct is associated with clock t in P_a, then $t = d' - d$ satisfies $D[t]$.

For instance, if $P_c = Wait[3];\ Wait[5]$ and $P = Wait[4]_t;\ Wait[5]$, then P with zone $t \leq 4$ abstracts P_c. Next, we show that $C1$, $C2$ and $C3$ of Definition 2.4 are satisfied by $\mathcal{R}$. $C3$ is proved trivially. $C1$ and $C2$ are proved by structural induction, which are exemplified using two cases where P_c is $Wait[d]$ or $P\ timeout[d]\ Q$. Notice that the first step of Definition 4.2 is to apply $\mathcal{A}$ to P_a. Let the resultant process be P_a' and the result zone be D'.

—If P_c is $Wait[d]$ and $((V_c, P_c), (V_a, P_a, D_a)) \in \mathcal{R}$, P_a' is $Wait[d']_t$ such that $t = d'$ satisfies $D_a'[t]$. We first show that $C1$ is satisfied. By rule $Wait1$ and $Wait2$,

[6]It is clear that this assumption can be relaxed to allow finite domain parameters.

$(V_c, \mathit{Wait}[d]) \stackrel{d,\tau}{\rightarrow} (V_c, \mathit{Skip})$. By rule *await*, $(V_a, P'_a, D'_a) \stackrel{\tau}{\rightsquigarrow} (V_a, \mathit{Skip}, D_a^{\uparrow} \wedge t = d)$ and thus $(V_a, P'_a, D'_a) \stackrel{\tau}{\rightsquigarrow} (V_a, \mathit{Skip}, D') \in T_a$ where $D' = \mathit{true}$ (since $cl(\mathit{Skip}) = \varnothing$). It is trivial to show $(V_c, \mathit{Skip}) \approx (V_a, \mathit{Skip}, \mathit{true})$. Further, $((V_c, \mathit{Skip}), (V_a, \mathit{Skip}, \mathit{true})) \in \mathcal{R}$. Similarly, we can show that $C2$ is satisfied.

—If P_c is P *timeout*$[d]$ Q, then P'_a is P' *timeout*$[d]_t$ Q such that P is abstracted by P' with D'. Assume $(V_c, P) \stackrel{\epsilon,e}{\rightarrow} (V_1, P_1)$ for some $\epsilon \leq d$. By rule $to4$ and $to1$, $(V_c, P_c) \stackrel{\epsilon,e}{\rightarrow} (V_1, P_1)$. By induction hypothesis, $(V_a, P', D') \stackrel{e}{\rightsquigarrow} (V_1, P'_1, D'_1)$ such that $((V_1, P_1), (V_1, P'_1, D'_1)) \in \mathcal{R}$ and $(V_1, P_1) \approx (V_1, P'_1, D'_1)$. By rule $ato2$, $(V_a, P'_a, D') \stackrel{e}{\rightsquigarrow} (V_1, P'_1, D'_1 \wedge D^{\uparrow} \wedge t \leq d)$. By assumption, $\epsilon \leq d$ and thus it is easy to show that P'_1 with $D'_1 \wedge D^{\uparrow} \wedge t \leq d$ abstracts P_1. Thus, $C1$ is satisfied. If $(V_a, P', D') \stackrel{e}{\rightsquigarrow} (V_1, P'_1, D'_1)$, $(V_a, P'_a, D') \stackrel{e}{\rightsquigarrow} (V_1, P'_1, D'_1 \wedge D^{\uparrow} \wedge t \leq d)$ by rule $ato2$. Because $D'_1 \wedge D^{\uparrow} \wedge t \leq d$ is not empty by definition, the transition $(V_a, P', D') \stackrel{e}{\rightsquigarrow} (V_1, P'_1, D'_1)$ must satisfy $t \leq d$ and therefore it must occur within $D'_1[t] - D'[t]$ time units. By induction hypothesis, there exists $\epsilon \leq D'_1[t] - D'[t]$ such that $(V_c, P_c) \stackrel{\epsilon,e}{\rightarrow} (V_1, P_1)$ and $(V_1, P'_1, D'_1) \approx (V_1, P_1)$. By rules $to4$ and $to1$, $(V_c, P_c) \stackrel{\epsilon,e}{\rightarrow} (V_1, P_1)$. It can be shown that P'_1 with $D'_1 \wedge D^{\uparrow} \wedge t \leq d$ abstracts P_1 and thus, $C2$ is satisfied in the case. Similarly, we prove the case where $(V_c, P\ \mathit{timeout}[d]\ Q) \stackrel{\epsilon,\tau}{\rightarrow} (V_1, P_1\ \mathit{timeout}[d]\ Q)$ for some $\epsilon \leq d$ or $(V_c, P\ \mathit{timeout}[d]\ Q) \stackrel{d,\tau}{\rightarrow} (V_1, Q)$.

Other cases can be proved to satisfy $C1$ and $C2$ similarly. We thus conclude that $\mathcal{R}$ is a time-abstract bi-simulation between $\mathcal{T_S}$ and $\mathcal{L_S}$ so that $\mathcal{T_S}$ and $\mathcal{L_S}$ are time-abstract bi-similar. □

By Theorem 5.2, properties which are preserved by time-abstract bi-simulation are preserved by $\mathcal{L_S}$ and therefore can be model checked based on $\mathcal{L_S}$. In the following, we take one class as an example and briefly discuss how it can be supported. Properties concerning both states and events of infinite runs can be specified in SE-LTL [Chaki et al. 2004], which is a linear temporal logic constituted by not only atomic state propositions but also events. SE-LTL is particularly interesting because Stateful Timed CSP is both state-based and event-based. SE-LTL properties can be model checked using an on-the-fly automata-based approach [Vardi and Wolper 1986]. Given an SE-LTL formula ϕ, a Büchi automaton $\mathcal{B}$ equivalent to the negation of ϕ can be built using the approach presented in [Gastin and Oddoux 2001]. The synchronous product of $\mathcal{L_S}$ and $\mathcal{B}$, which is also a Büchi automaton, is then computed. A run of the product is accepting if and only if its projection in $\mathcal{B}$ is accepting. The problem of model checking $\mathcal{S}$ against ϕ *without non-Zenoness assumption* is thus reduced to the standard emptiness problem of Büchi automata [Vardi and Wolper 1986; Holzmann 2003].

Model checking with non-Zenoness is more complicated. A Stateful Timed CSP model may contain Zeno runs. For instance, given a model $(\varnothing, \varnothing, P\ \mathit{deadline}[1])$ where $P \mathrel{\widehat=} a \rightarrow P \mid b \rightarrow \mathit{Skip}$. If property 'eventually event b occurs' is verified without non-Zenoness, then a counterexample with infinitely many a events will be generated. A close look reveals that the counterexample is Zeno since infinitely many a events must occur within 1 time unit. We thus need a method to check whether a run is Zeno or not. By Theorem 5.2, for every concrete run $\rho = \langle s_0, (\epsilon_0, a_0), s_1, (\epsilon_1, a_1), \cdots \rangle$ of $\mathcal{T}_\mathcal{S}$, there is a corresponding $\pi = \langle s_0, a_0, s_1, a_1, \cdots \rangle$ in $runs(\mathcal{L}_\mathcal{S})$. We say that ρ is an instance of π or equivalently π abstracts ρ. If π fails certain property, then ρ can be presented as a concrete counterexample. It is possible that all instances of π are Zeno so that they are not considered as realistic counterexamples. An abstract run π is Zeno if and only if all instances of π are Zeno. Otherwise, π is non-Zeno. Because Zeno runs are unrealistic, system verification must be performed with the assumption of non-Zenoness, i.e., to verify properties against only non-Zeno runs. In the setting of Timed Automata, it has been shown that it is highly nontrivial to decide if an abstract run is non-Zeno or not. The reason is that zone abstraction for Timed Automata fails *pre-stability* [Tripakis 1999]. The following shows that zone graphs in our setting satisfies pre-stability.

LEMMA 5.3. *Let $\mathcal{S}$ be a model. Let $(V, P, D) \overset{a}{\rightsquigarrow} (V', P', D')$ be a transition of $\mathcal{L}_\mathcal{S}$. For all (V, Q) such that (V, P, D) abstracts (V, Q), there is (V', Q') such that (V', P', Q') abstracts (V', Q') and $\epsilon \in \mathbb{R}_+$ such that $(V, Q) \overset{\epsilon, a}{\rightarrow} (V', Q')$.*

Proof By the proof of Theorem 5.2, $((V, Q), (V, P, D)) \in \mathcal{R}$ and therefore the lemma holds by definition. □

Remark The reason why our zone graph satisfies pre-stability is related to the characteristics of $\mathcal{L}_\mathcal{S}$, as we explain in the following. Notice that $\mathcal{L}_\mathcal{S}$ can be systematically translated into an equivalent Timed Automaton. Let $\mathcal{A}_\mathcal{S}$ denote the Timed Automaton. Every state (V, P, D) of $\mathcal{L}_\mathcal{S}$ is translated into a state of $\mathcal{A}_\mathcal{S}$. Recall that a transition from a state (V, P, D) of $\mathcal{L}_\mathcal{S}$ is generated by associating a fresh t with P; applying a firing rule so that $(V, \mathcal{A}(P, t), D \wedge t = 0) \overset{a}{\rightsquigarrow} (V', P', D')$ and lastly pruning unused clocks from D'. For each such transition, a corresponding transition is introduced in $\mathcal{A}_\mathcal{S}$ such that it is labeled with event a and clock constraint D'. Furthermore, all incoming transitions to the state (V, P, D) is labeled with a set of resetting clocks $\{t\}$. For instance, the right part of Figure 7 shows the generated Timed Automaton of the zone graph shown on the left.

The following is true about $\mathcal{A}_\mathcal{S}$ (but not Timed Automata in general): for every clock t, assume ϕ_i and ϕ_j are two constraints on t associated with two transitions along any path starting and ending with a transition resetting t, then a valuation of t which satisfies ϕ_i can always satisfy ϕ_j by letting time elapse. This can be proved by looking at the abstract firing rules and Definition 4.2. When a clock t is introduced, it is associated with a *maximum* set of timed process constructs, which results in a maximum set of constraints of the form $t \leq d$ or $t = d$. Later, when timed process constructs are discharged through transitions, there are less and less constraints on t. This justifies that a clock cannot go too far and not be able to satisfy a feasible transition later, which is why pre-stability is satisfied in our setting. Intuitively, it is because every clock acts as a count-down clock which cannot be modified or reset before it is expired. □

A transition of $\mathcal{L}_\mathcal{S}$ is sometimes written in the form $(V, P, D) \overset{t,a,X}{\rightsquigarrow} (V', P', D')$ such that t is the introduced clock; a is the event and X is the set of pruned clocks. Because a clock is introduced for every transition, through $D'[t]$, we can infer the time needed for the transition to occur. The transition is *instantaneous* if and only if $D' \Rightarrow t = 0$. Next, we establish a necessary and sufficient condition to check whether an abstract run is Zeno or not based on Lemma 5.3. Because $\mathcal{L}_\mathcal{S}$ is finite, an infinite run π of $\mathcal{L}_\mathcal{S}$ must visit a finite set abstract configurations, denoted as $inf(\pi)$, infinitely often. Let $loopCLK(\pi)$ denote the set $\{x \mid \forall (V, P, D) \in inf(\pi).\ x \in cl(D)\}$, i.e., the clocks which are present in every abstract configuration which is visited infinitely often.

THEOREM 5.4. *Let $\mathcal{S}$ be a model; π be a run of $\mathcal{L}_\mathcal{S}$. π is non-Zeno if and only if $loopCLK(\pi) = \varnothing$ and not all infinitely visited transitions are instantaneous.* □

Proof Let ρ be a run of $\mathcal{T}_\mathcal{S}$ and π be the corresponding abstract run of $\mathcal{L}_\mathcal{S}$.

$$\rho = s_0 \overset{\epsilon_0, a_0}{\rightarrow} s_1 \overset{\epsilon_1, a_1}{\rightarrow} \cdots s_i \overset{\epsilon_i, a_i}{\rightarrow} \cdots$$
$$\pi = c_0 \overset{t_0, a_0, X_0}{\rightsquigarrow} c_1 \overset{t_1, a_1, X_1}{\rightsquigarrow} \cdots c_i \overset{t_i, a_i, X_i}{\rightsquigarrow} \cdots$$

Only-if: We show that if ρ is non-Zeno, then $loopCLK(\pi) = \varnothing$. Assume that $t_i \in loopCLK(\pi)$. By definition, $\epsilon_i + \epsilon_{i+1} + \cdots$ is unbounded. Therefore, the reading of t_i becomes unbounded. Because t_i is never pruned, there must be some constraints on t in every D_m such that $m \geq i$. According to the abstract firing rules, the constraint must be of the form $t_m = n$ or $t_m \leq n$ where n is a constant. Because t_m is unbounded, we derive that $t_m > n$ and reach contradiction. Therefore, $loopCLK(\pi) = \varnothing$. Furthermore, because ρ is non-Zeno, $\epsilon_i + \epsilon_{i+1} + \cdots + \epsilon_{i+k} > 0$ and thus there must be a transition which is not instantaneous.
If: We show that if $loopCLK(\pi) = \varnothing$ and there is at least one infinitely often visited transition that is not instantaneous, then π is non-Zeno. By Lemma 5.3, strictly positive number of time units can elapse at a transition which is not instantaneous. Because $loopCLK(\pi) = \varnothing$, every clock is pruned (and re-introduced) before taking the transition again. Let ρ be a run which takes the transition repeatedly with a non-zero delay. By [Alur and Dill 1994], ρ is *progressive* as all clocks are reset (which is equivalent to *pruned and re-introduced*) infinitely often and strictly positive infinitely often. Therefore, ρ is non-Zeno. □

The next theorem follows immediately. Intuitively speaking, it allows us to solve the emptiness problem of Stateful Timed CSP using methods based on finding maximal strongly connected components (SCC). Given a set of states scc constituting an SCC, let $loopCLK(scc)$ denotes the set $\{x \mid \forall (V, P, D) \in scc.\ x \in cl(D)\}$.

THEOREM 5.5. *Let $\mathcal{S}$ be a model. $\mathcal{T}_\mathcal{S}$ is non-empty if and only if there exists a reachable maximal SCC scc in $\mathcal{L}_\mathcal{S}$ such that $loopCLK(scc) = \varnothing$ and not all transitions connecting two states in scc are instantaneous.*

Proof (**Only-if**) Let π be a non-Zeno run of $\mathcal{T}_\mathcal{S}$. Let scc be the set of states constituting the maximal SCC which contains all states and transitions visited infinitely often by π. If π is non-Zeno, $loopCLK(\pi) = \varnothing$ and therefore $loopCLK(scc) = \varnothing$. Furthermore, the transition which is not instantaneous in π is contained in scc. (**If**) Let scc be the maximal SCC which satisfies the condition. A run which traverses through every state and transition of scc is non-Zeno by Theorem 5.4. □

Given an SE-LTL formula ϕ, a Büchi automaton $\mathcal{B}$ equivalent to the negation of ϕ, model checking *with non-Zenoness assumption* is to construct the product of

$\mathcal{B}$ and $\mathcal{L}_\mathcal{S}$ and then search for an accepting run of the product whose projection on $\mathcal{L}_\mathcal{S}$ is non-Zeno. By Theorem 5.5, it is equivalent to searching for a particular maximal SCC *scc*. Therefore, the problem can be solved by an algorithm which has a complexity linear in the number of transitions in the product (e.g., based on Tarjan's algorithm for finding SCCs).

6. EVALUATION

System modeling and verification using Stateful Timed CSP have been implemented in the PAT model checker [Sun et al. 2009]. PAT is a self-contained environment for system modeling, simulation and verification. It has an extensible architecture which allows quick realization of new techniques for modeling, abstraction or verification. Interested readers are referred to [Liu et al. 2010]. The model checker for Stateful Timed CSP is built as one self-contained module in PAT, which supports SE-LTL model checking and refinement checking[7]. In the following, we evaluate Stateful Timed CSP in two aspects, system modeling and verification.

6.1 Modeling

We illustrate system modeling in Stateful Timed CSP using a multi-lift system. The system is chosen for two reasons. Firstly, the system is hierarchical, real-timed and rich in data states, which nicely demonstrates language features of Stateful Timed CSP. Secondly, the lift system is a standard case study used to demonstrate the expressive power of various specification techniques and languages. The user requirements and behaviors of the system are intuitively clear and therefore the readers can focus on the modeling. Though inspired by [Mahony and Dong 1998], our model is different from [Mahony and Dong 1998] in many aspects, e.g., our model uses shared variables, whereas [Mahony and Dong 1998] relies mostly on processes and channels communication; our model implements data operations using executable programs, whereas data operations are abstract in [Mahony and Dong 1998]; and probably most importantly, our model is model checkable whereas [Mahony and Dong 1998] is not.

The lift system consists of a building, multiple lifts and a central controller. In the following, we present the lift system model incrementally in bottom-up manner, beginning with models of the primitive components, which are then used to compose complex components. Notice that the language supported by PAT is slightly different from the previously presented notations for user's convenience. For instance, synchronous/asynchronous channels and constant definitions are supported. In the lift system model, the following constants are relevant: *NoOfFloors* (the number of floors); *NoOfLifts* (number of lifts); *Off* of value 0; *Up* of value 1; *Down* of value -1; and *Both* of value 2.

A building consists of multiple floors and each floor is equipped with one button panel on the wall so that a user can make an *external request* to traveling upwards or downwards. A button can be pushed at any time. Once pushed, the button is on until the requested service is provided. The status of the button (or equivalently the external requests) is maintained in an array *FloorButtons* of length *NoOfFloors*. Each variable in the array has four possible values: *Off* (i.e., it is not on), *Up*

[7]Readers are recommended to download PAT at [Sun et al.] and try out the RTS module.

(i.e., upward traveling has been requested), *Down* (i.e., downward traveling downward has been requested) or *Both* (i.e., both directions have been requested). The following models the building.

1. $Press(floor, direction) \mathrel{\widehat{=}} request.floor.direction\{$
2. $\quad if(FloorButtons[floor] = None)\{$
3. $\quad\quad FloorButtons[floor] := direction$
4. $\quad \}$
5. $\quad else\ if\ (FloorButtons[floor] \neq direction)\{$
6. $\quad\quad FloorButtons[floor] := Both$
7. $\quad \}$
8. $\} \rightarrow Skip$
9. $TopFloor \mathrel{\widehat{=}} Press(NoOfFloors - 1, Down);\ TopFloor$
10. $GroundFloor \mathrel{\widehat{=}} Press(0, Up);\ GroundFloor$
11. $MiddleFloor(n) \mathrel{\widehat{=}} (Press(n, Down) \mid Press(n, Up));\ MiddleFloor(n)$
12. $Building \mathrel{\widehat{=}} TopFloor \parallel GroundFloor \parallel (||_{x=1}^{NoOfFloors-2}\ MiddleFloor(x))$

Lines 1 to 8 define process *Press*(*floor*, *direction*) which models the process of pressing a floor button, where parameters *floor* and *direction* denote the requesting floor and traveling direction respectively. Notice that *direction* has two possible values: *Up* (1) or *Down* (-1). Event *request.floor.direction* is the event of a user pressing a button at the *floor* to travel in the *direction*. It is associated with a program (from line 2 to 7), which stores the request in the *FloorButtons* array. Line 9 models the top floor, where only traveling downwards is possible. Line 10 models the ground floor where only traveling upwards is possible. Line 11 models a middle floor, where traveling in both directions are possible. Lastly, line 12 models the building, which is a parallel composition of all floors. Notice that process *Building* is not real-timed since requests can arrive at any time.

Each lift consists of four components, i.e., a door for allowing access to and from the lift, a shaft for transporting the lift, an internal queue for determining the lift itinerary and a controller to coordinate the behaviors of the other components. The following is a model of the door.

$$Door \mathrel{\widehat{=}} open \rightarrow (Cycle;\ close \rightarrow Skip)\ deadline[maxTime];\ Door$$
$$Cycle \mathrel{\widehat{=}} toOpen \rightarrow opened \rightarrow conf \twoheadrightarrow Wait[minTime];\ Closing$$
$$Closing \mathrel{\widehat{=}} (closed \rightarrow Skip)\ interrupt\ (sensor \rightarrow toOpen \rightarrow opened \rightarrow Closing)$$

Process *Cycle* models the process of opening the door and later closing it. It is initiated in process *Door* by the receipt of an *open* signal from the lift controller and completed by sending a *close* signal. That is, events *open* and *close* must be synchronized by the door and lift controller. Event *conf* is a signal from the door to the lift controller to indicate that the door has been opened and thus relevant external/internal service requests can be removed. After waiting for $minTime$ time units after the door is opened, a signal *close* is sent to indicate that the door is to be closed. Process *Closing* models the process of closing the door. If an interrupt is detected through event *sensor* before the door is *closed*, the door is re-opened and later closed again. Notice that the door has many timing properties. For instance, signal *conf* must occur immediately because of $\twoheadrightarrow$ and the *deadline* in process *Door* states that a door can not remain opened forever.

$$\begin{aligned}&\mathit{Shaft}(i) \mathrel{\widehat=} \mathit{move}?[\mathit{id} = i]\mathit{id}.n.\mathit{dir} \rightarrow \mathit{Wait}[n * \mathit{movingTime} + \mathit{delayTime}];\\&\qquad\qquad \mathit{arrive} \twoheadrightarrow \mathit{Shaft}(i)\end{aligned}$$

The above models the shaft. Notice that *move* is a synchronous channel, which acts like a pair-wise synchronizing event. The question mark denotes that this is a channel input. The variables *id*, n and *dir* are place-holders for the received data. In particular, *id* indicates the intended lift; n is the number of floor to move across; and *dir* is the direction of movement. Condition $id = i$ constrains that only channel inputs satisfying the condition are received. Intuitively, it means that the shaft only picks up messages with the matching identity. The feature is adopted from the Promela language [Holzmann 2003]. It can be shown that our results in this work remain valid with channels. After receiving the input, the shaft starts moving and later signals arrival through synchronizing event *arrive*. Notice that *movingTime* is a constant denoting the time needed to travel across one floor and *delayTime* is a delay caused by the initial acceleration and final braking of the lift.

Inside each lift, there is a button panel so that a user can make an *internal request* to travel to a particular floor. The panel buttons are in one-to-one correspondence with the floor numbers. The internal requests (or equivalently the status of the internal panel buttons) are maintained in an array *IntReq*, which has dimension $\mathit{NoOfLifts} \times \mathit{NoOfFloors}$. Entry $\mathit{IntReq}[i][j] = \mathit{true}$ if and only if there is an internal request in i-lift for j-floor.

$$\begin{aligned}&\mathit{InternalQ}(i) \mathrel{\widehat=} \mathit{intReq}.0\{\mathit{IntReq}[i][0] := 1\} \rightarrow \mathit{InternalQ}(i) \mid\\&\quad \mathit{intReq}.1\{\mathit{IntReq}[i][1] := 1\} \rightarrow \mathit{InternalQ}(i) \mid \cdots \mid\\&\quad \mathit{intReq}.(\mathit{NoOfFloors} - 1)\{\mathit{IntReq}[i][\mathit{NoOfFloors} - 1] := 1\} \rightarrow \mathit{InternalQ}(i)\end{aligned}$$

The above models the internal queue of requests. The process generates all possible internal requests using choices. Note that the removal of internal requests is not modeled as a part of the above process but rather in the lift controller process, which is shown below.

$$\begin{aligned}&\mathit{LiftCtrl}(i, \mathit{fl}, \mathit{dir}) \mathrel{\widehat=} \mathit{check}!\mathit{fl}.\mathit{dir}.\mathit{call}(\mathit{GetDesInt}, \mathit{IntReq}, \mathit{fl}, \mathit{dir}, \mathit{NoOfFloors}, i)\\&\quad \rightarrow \mathit{check}?\mathit{des} \rightarrow \mathit{case}\ \{\\&\qquad \mathit{des} = \mathit{fl} : \mathit{open} \rightarrow \mathit{conf} \rightarrow \mathit{ClearReq}(i, \mathit{fl}, \mathit{dir});\\&\qquad\quad \mathit{close} \rightarrow \mathit{LiftCtrl}(i, \mathit{fl}, \mathit{dir})\\&\qquad \mathit{des} > \mathit{fl} : \mathit{move}!(\mathit{des} - \mathit{fl}).\mathit{Up} \rightarrow \mathit{arrive} \rightarrow \mathit{open} \rightarrow\\&\qquad\quad \mathit{conf} \rightarrow \mathit{ClearReq}(i, \mathit{des}, \mathit{Up});\ \mathit{close} \rightarrow \mathit{LiftCtrl}(i, \mathit{des}, \mathit{Up})\\&\qquad 0 \leq \mathit{des} < \mathit{fl} : \mathit{move}!i.(\mathit{fl} - \mathit{des}).\mathit{Down} \rightarrow \mathit{arrive} \rightarrow \mathit{open} \rightarrow\\&\qquad\quad \mathit{conf} \rightarrow \mathit{ClearReq}(i, \mathit{des}, \mathit{Down});\ \mathit{close} \rightarrow \mathit{LiftCtrl}(i, \mathit{des}, \mathit{Down})\\&\qquad \mathit{default} : \mathit{Wait}[\mathit{delayTime}];\ \mathit{LiftCtrl}(i, \mathit{fl}, \mathit{dir})\\&\quad \}\end{aligned}$$

The three parameters denote the lift *i*dentity, its current *fl*oor and *dir*ection respectively. Process *LiftCtrl* starts with sending a compound message on channel *check* to the central controller, indicating that it is ready to serve a request. The message consists of: *fl* which is the floor that the lift is at; *dir* which is the traveling direction; and the floor which the lift is traveling to. The latter is computed based on the internal requests using an externally defined function. In PAT, external C# libraries are allowed in Stateful Timed CSP models so that the models are simplified by encapsulating complicated data operations. In this example, *call* is a reserved keyword for invoking an external function; the function name is *GetDesInt*; and the rest are inputs to method *GetDesInt*. The function returns the next internally requested floor in the traveling direction if one exists, or the nearest internal request in the opposite direction, or -1 if there is no internal request[8].

The central controller is responsible for assigning external requests to specific lifts, which is modeled as follows.

$$\begin{aligned}&\mathit{Controller} \mathrel{\widehat{=}} \mathit{check}?\mathit{fl}.\mathit{dir}.\mathit{des} \rightarrow\\ &\qquad \mathit{check}!\mathit{call}(\mathit{GetDesExt}, \mathit{FloorButtons}, \mathit{fl}, \mathit{dir}, \mathit{des}, \mathit{NoOfFloors}) \rightarrow \mathit{Controller}\end{aligned}$$

Upon receiving the message from a lift, the central controller checks the pool of external requests (which are stored in *FloorButtons*) and decides whether to assign an external request to the lift. Function *GetDesExt* checks if there is an external request along the way for the lift. If there is, it sends the new destination on channel *check* to the lift. If the received *des* is -1, which means that there is no internal request for the lift, then it assigns an external request on the lift's current traveling direction. If there is no request on the current direction, then it assigns a request on the opposite direction. If there are no external requests, the message sent is -1.

Once the lift controller receives the new destination from the central controller, its behaviors diverge, which are modeled using a 'syntactic sugar'. Process $\mathit{case}\ \{c_0 : P_0 \quad c_1 : P_1 \quad \cdots\}$ is equivalent to $\mathit{if}\ (c_0)\{P_0\}\ \mathit{else}\ \{\mathit{if}\ (c_1)\ \{P_1\}\ \mathit{else}\ \{\cdots\}\}$. That is, the conditions $c_0, c_1, \cdots$ are evaluated one by one until one evaluates to true and then the corresponding process is chosen. In particular, if there is an internal request for the current floor or there is an external request from the current floor to travel in the current direction (i.e., $\mathit{des} = \mathit{fl}$), then the door is opened to serve the request. Otherwise, if the destination is above (below) the current floor, the shaft is commanded to travel upwards for $\mathit{des} - \mathit{fl}$ floors (downwards for $\mathit{fl} - \mathit{des}$ floors) and then the door is opened. Once the door is confirmed opened, by synchronizing *conf*, the requests are cleared by *ClearRequest*, which is defined as follows.

$$\begin{aligned}&\mathit{ClearReq}(i, \mathit{fl}, \mathit{dir}) \mathrel{\widehat{=}} \mathit{clearRequest}\{\\ &\qquad\qquad \mathit{IntReq}[i][\mathit{fl}] := 0;\\ &\qquad\qquad \mathit{if}(\mathit{FloorButtons}[\mathit{fl}] = \mathit{dir})\{\mathit{FloorButtons}[\mathit{fl}] := \mathit{None}\}\\ &\qquad\qquad \mathit{else}\{\mathit{FloorButtons}[\mathit{fl}] := -1 * \mathit{dir}\}\\ &\qquad \} \rightarrow \mathit{Skip}\end{aligned}$$

Afterwards, the door is closed by signal event *close* and then the lift controller restarts. If there are no internal requests or external requests (i.e., $\mathit{des} = -1$), then the lift controller simply waits for some time and then restarts. Notice that in this modeling, priority has been given to the internal requests. It is possible that a lift system is designed otherwise.

$$\begin{aligned}&\mathit{Lift}(i) \mathrel{\widehat{=}} (\mathit{Shaft}(i) \parallel \mathit{Door} \parallel \mathit{LiftCtrl}(i, 0, \mathit{Up}) \parallel \mathit{InternalQ}(i))\\ &\qquad\qquad \big\backslash\{\mathit{open}, \mathit{conf}, \mathit{close}, \mathit{arrive}\};\end{aligned}$$

A lift is then modeled as the parallel composition of the shaft, the door and the lift controller and the internal queue. Notice that the synchronizing events between the

[8]Details of the C# methods are skipped as they are less interesting.

components are hidden from the environment. Lastly, the lifts are the interleaving of all individual lifts and the lift system is composed of the interleaving of the lifts, the central controller and the building.

$$Lifts() \mathrel{\widehat{=}} |||_{i=0}^{NoOfLifts-1}\ Lift(i)$$
$$System() \mathrel{\widehat{=}} Lifts() \ ||| \ CentralController() \ ||| \ Building()$$

This model demonstrates how Stateful Timed CSP may be applied to model hierarchical real-time systems step-by-step. The rich set of process constructs not only allow us to capture real-time behaviors intuitively – without thinking about the clocks, but also to build the system model incrementally from primitive system components.

6.2 Verification

In the following, we evaluate efficiency of our method in order to show that it is practically useful. Table I shows statistics of system verification using PAT. The data are obtained with Intel® Xeon® CPU E5506 @2.13GHz and 32GB memory, on a 64-bit Windows system. '-' denotes that the experiment is aborted due to out of memory or running more than 4 hours. The verified models include the pacemaker model, the lift system, and benchmark real-time systems like Fischer's mutual exclusion algorithm, the railway control system [Yi et al. 1994], the CSMA/CD protocol [Bozga et al. 1998], and the Fiber Distributed Data Interface (FDDI) [Larsen et al. 1997]. All models with configurable parameters are available at [Sun et al.]. In the first column, the number after the model name is the number of processes. All properties are verified with or without the assumption of non-Zenoness. The verification time without non-Zenoness is shown in column Z and the time with non-Zenoness is shown in column NZ. Notice that deadlock-freeness with the assumption of non-Zenoness means that the system never reaches a state where both time-transition and event-transition are impossible, which can be checked based on the zone graphs. Column $\#St$ shows the number states in the zone graphes. Column $\#Clock$ shows the maximum number of clocks created during verification. Memory usage is skipped because PAT is based on C# with dynamic garbage collection and therefore accurate memory usage is hard to obtain.

A number of observations can be obtained from the data. Firstly, PAT currently handles in average $15K$ states per second (i.e., the total number of visited states – not new states – divided by the total number of seconds), which is reasonable compared to existing model checkers [Holzmann 2003; Roscoe et al. 1995; Larsen et al. 1997]. Secondly, model checking with non-Zenoness has little or no computational overhead. Compared to other work on model checking with non-Zenoness [Tripakis 1999; Gómez and Bowman 2007; Herbreteau et al. 2010; Herbreteau and Srivathsan 2010], this is a clear advantage. Thirdly, for some models, the number of clocks remains constant when the system size increases, e.g., the railway control system and the CSMA/CD protocol. This is because clocks are shared as much as possible in our approach.

In order to compare our method with the state-of-art real-time model checker, we conducted experiments to compare performance of PAT and Uppaal. The results are summarized in Table II, where column Uppaal(s) shows the verification time using Uppaal, with all optimization techniques. Notice that Uppaal outperforms

Model	Property	#St	#Clock	Z(s)	NZ(s)
Pacemaker	deadlock-free	463K	2	19	19
Lift (2floor; 2lift)	deadlock-free	257K	4	81	83
Lift (3floor; 1lift)	deadlock-free	14K	2	4	4
Lift (3floor; 2lift)	deadlock-free	6M	4	2788	2770
Fischer*4	LTL	2K	4	< 1	< 1
Fischer*5	LTL	15K	5	1	1
Fischer*6	LTL	108K	6	14	14
Fischer*7	LTL	857K	7	289	289
Railway*4	LTL	1K	4	< 1	< 1
Railway*5	LTL	7K	4	< 1	1
Railway*6	LTL	74K	4	5	5
Railway*7	LTL	324K	4	39	37
Railway*8	LTL	2.6M	4	845	671
CSMA*5	deadlock-free	3K	5	< 1	< 1
CSMA*6	deadlock-free	10K	5	1	1
CSMA*7	deadlock-free	30K	5	4	3
CSMA*8	deadlock-free	82K	5	11	11
CSMA*9	deadlock-free	218K	5	34	33
CSMA*10	deadlock-free	565K	5	100	99
CSMA*11	deadlock-free	1.4M	5	294	287
CSMA*12	deadlock-free	3.6M	5	848	838
CSMA*13	deadlock-free	8.7M	5	2421	2358
FDDI*3	LTL	4K	5	< 1	< 1
FDDI*4	LTL	46K	6	9	9
FDDI*5	LTL	6.4M	7	1877	1876
FDDI*3	deadlock-free	3K	5	< 1	< 1
FDDI*4	deadlock-free	28K	6	6	5
FDDI*5	deadlock-free	3.5M	7	1000	989

Table I. Experiment results

PAT in many cases[9]. There are a number of reasons. Firstly, our zone graphs are more complicated than those of Timed Automata. The nodes in our zone graphs, i.e., the abstract configurations, are more complicated than those in Uppaal as an abstract configuration consists of a process expression. The process expression can not be abstracted as an array of numbers because the system structure in Stateful Timed CSP varies through transitions. Furthermore, our zone graphes may contain more nodes due to the extra τ-transitions introduced by the compositional process constructs, e.g., the τ-transition generated by abstract firing rule *ato*3. Combined with parallel composition, these τ-transitions may result in a large number of additional states. In hand-crafted Uppaal models, the τ-transitions are often removed by carefully manipulating the clock guards or grouping clock guards and events into the same transition. Removing the extra τ-transitions is highly nontrivial. In fact, we believe that they are a price to pay in order to model hierarchical systems. Secondly, PAT is slower than Uppaal simply because some effective optimization techniques are currently missing. One particular example is *extrapolation*. The col-

[9]except the CSMA/CD protocol. The reason seemed to be that the original Uppaal model uses a global variable to check whether a station should receive the message, whereas in PAT, it is naturally modeled using a value passing channel, with guard conditions on accepting values.

Model	#Clocks		Without Non-Zenoness		
	PAT	UPPAAL	PAT(s)	UPPAAL(s)	UPPAAL+τ-o(s)
Fischer*5	5	5	1	<1	696
Fischer*6	6	6	14	<1	-
Railway*6	4	6	5	<1	-
Railway*7	4	7	39	3	-
CSMA*6	5	7	1	1	-
CSMA*7	5	8	4	6	-
CSMA*8	5	9	11	34	-
CSMA*9	5	10	34	181	-
CSMA*10	5	11	100	1048	-

Table II. Experiment results

umn UPPAAL+τ-o shows the verification time using UPPAAL without *extrapolation* (and with the same extra τ-transitions so that the models in PAT and UPPAAL have similar state spaces). The results show that PAT often outperforms UPPAAL in this setting. This suggests that PAT could be more efficient with similar optimizations in place. One last thing to notice is that in all the experiments, PAT uses less clocks than UPPAAL. It remains our future work to explore this fact and UPPAAL's powerful optimization techniques to improve PAT.

In summary, the reason why the current PAT implementation is useful is three-fold. Firstly, Stateful Timed CSP is more suited to model hierarchical real-time systems than Timed Automata. Secondly, PAT supports verification with non-Zenoness with little or no extra cost. Lastly, PAT is still reasonably efficient and supports an alternative way of specifying properties (e.g., in SE-LTL).

7. RELATED WORK

This work is related to research on real-time system modeling and verification. Compositional specification for real-time systems based on timed process algebras has been studied extensively. Examples include the algebra of timed processes named ATP [Sifakis 1999; Nicollin and Sifakis 1994], the extension of CCS with real time [Yi 1991] and Timed CSP [Reed and Roscoe 1986; Schneider 2000]. S-tateful Timed CSP is an extension of Timed CSP. Different from timed process algebras, Stateful Timed CSP integrates timed process constructs with complex data variables/operations in order to model real-world systems. There has been a related line of research on integrating timed process algebra with state-based specification languages [Mahony and Dong 2000; Butterfield et al. 2007]. One closely related language is called TCOZ [Mahony and Dong 2000], which is an integration of Timed CSP and Object-Z. In TCOZ, Object-Z is used to model data structures and operations. Different from previous work on integrated formal specification, Stateful Timed CSP is designed to be executable and model checkable. The key difference is that concrete executable programs instead of pre/post-conditions are used to specify data operations. As a modeling language for real-time systems, Stateful Timed CSP is related to Timed Automata [Alur and Dill 1994]. Remotely related modeling languages are Statecharts [Harel 1997] with clocks and timed Petri nets [Ramchandani 1974], which are capable of modeling hierarchical systems with real-time constraints.

There have been many approaches on building verification support for timed process algebras. Development of tool support for ATP was evidenced in [Nicollin et al. 1992; Closse et al. 2001]. In [Yi et al. 1994], a constraint solving based verification method was proposed to verify CCS + real time. In [Brooke 1999], a theorem proving approach for Timed CSP was discussed. In order to avoid the complexity of developing a model checker from scratch, a number of translation-based approaches have been studied. In our previous work [Dong et al. 2004; 2008], Timed CSP (as part of TCOZ models) is translated to Timed Automata so that Uppaal can be applied. In [Dong et al. 2006], Timed CSP is encoded into a constraint solver so as to verify reachability properties. These approaches share the common problems with all translation-based approaches. That is, the target tool Uppaal is not designed for Timed CSP and therefore features of Timed CSP may not be effectively encoded or efficiently verified. For instance, every timed process construct results in one fresh clock [Dong et al. 2006], which resulted in using more clocks than necessary. Furthermore, reflecting verification results back to the level of Timed CSP is not trivial. In [Ouaknine and Worrell 2002], it was proved that through digitalization, Timed CSP models can be translated into CSP models and verified by CSP model checkers like FDR [Roscoe et al. 1995]. Compared to zone abstraction adapted in this work, digitalization becomes ineffective when a model involves largely different constants associated with timed processes. There has been little verification support for integration of Timed CSP with other languages. To the best of our knowledge, the PAT model checker is the first dedicated verification tool supporting verification of hierarchical complex real-time systems with data structures/operations.

Research on verifying real-time systems have been focused on Timed Automata. Several model checkers have been developed with Timed Automata or Timed Safety Automata [Henzinger et al. 1994] being the core of their input languages [Larsen et al. 1997; Bozga et al. 1998; Tasiran et al. 1996]. Zone abstraction was originally introduced for Petri net [Berthomieu and Menasche 1983] and then adapted to the framework of Timed Automata [Dill 1989]. Our zone abstraction is based on the zone abstraction developed in [Yi et al. 1994; Dill 1989]. In contrast to approaches based on Timed Automata, our approach is capable of modeling and verifying hierarchical systems. This work is closely related to work on Hierarchical Timed Automata [Jin et al. 2007; David et al. 2001; Dong et al. 2008]. In [Jin et al. 2007], formal definitions for Hierarchical Timed Automata and their composition were defined. Furthermore, compositional verification based on Multiset-LTS are discussed. Different from [Jin et al. 2007], our work offers an alternative approach based implicit clocks.

This work is related to research on verification with non-Zenoness assumption. Syntactic conditions for Timed Automata to be free from Zeno runs have been identified in [Tripakis 1999; Gómez and Bowman 2007]. The conditions are often sufficient only [Bowman and Gómez 2006]. In the setting of Timed Automata, it has been shown that it is not possible to determine if a run can be instantiated to a non-Zeno run given only zone graphs. The solution involving adding one extra clock has been discussed in [Tripakis 1999; Tripakis et al. 2005; Tripakis 2009]. Recently, it has been shown that adding one clock may result in an exponentially larger zone graph [Herbreteau et al. 2010; Herbreteau and Srivathsan 2010]. The remedy is to transform the zone graph into a *guess zone graph* and require that all clocks which are bounded from above must be reset infinitely often during a run and the run must visit a state such that the clocks can be strictly positive [Herbreteau et al. 2010]. In this work, we show that zone graphs generated from Stateful Timed CSP models are different as our zone graphs satisfy pre-stability and all clocks are bounded from above. As a result, detecting Zeno runs based on zone graphs is straightforward. In terms of tool support for model checking with non-Zenoness, only Uppaal and KRONOS allow some form of non-Zenoness detection. Uppaal relies on test automata [Aceto et al. 2003] and leads-to properties. The problem with this approach is that it is sufficient-only. KRONOS supports an expressive language for specifying properties, which allows encoding of a sufficient and necessary condition for non-Zenoness. Checking for non-Zenoness in KRONOS is expensive. In comparison, checking non-Zeneness in our setting has a negligible computational overhead.

8. CONCLUSION

In this work, we develop a self-contained approach for model checking hierarchical complex real-time systems. In particular, we propose a modeling language named Stateful Timed CSP, which extends Timed CSP with data components as well as additional timed process constructs. We developed a fully automatic method to generate finite-state abstraction from Stateful Timed CSP models. We show that the abstraction preserves interesting properties by proving that it is time-abstract bi-similar to the original model. We then tackle the problem of non-Zenoness. We show that it is possible to check non-Zenoness based on zone graphs so that properties can be verified with the assumption of non-Zenoness. Lastly, our methods are implemented in the PAT framework.

As for future work, because verification on CSP-based models has been traditionally based on refinement checking [Roscoe 2005], we are currently investigating how to check timed refinement relationship between two Stateful Timed CSP models with the assumption of non-Zenoness. In addition, state space reduction techniques like extrapolation, symmetry reduction and partial order reduction for Stateful Timed CSP are to be studied.

REFERENCES

Aceto, L., Bouyer, P., Burgueño, A., and Larsen, K. G. 2003. The Power of Reachability Testing for Timed Automata. *Theoretical Computer Science 300,* 1-3, 411–475.

Alur, R. and Dill, D. L. 1994. A Theory of Timed Automata. *Theoretical Computer Science 126*, 183–235.

Barold, S. S., Stroopbandt, R. X., and Sinnaeve, A. F. 2004. *Cardiac Pacemakers Step by Step: an Illustrated Guide.* Blachwell Publishing.

Behrmann, G., Larsen, K. G., Pearson, J., Weise, C., and Yi, W. 1999. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *11th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science, vol. 1633. Springer, 341–353.

Bengtsson, J. and Yi, W. 2003. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets.* Lecture Notes in Computer Science, vol. 3098. Springer, 87–124.

Berthomieu, B. and Menasche, M. 1983. An Enumerative Approach for Analyzing Time Petri Nets. In *IFIP Congress*. 41–46.

Bowman, H. and Gómez, R. 2006. How to Stop Time Stopping. *Formal Aspects of Computing 18,* 4, 459–493.

Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., and Yovine, S. 1998. Kronos: A Model-Checking Tool for Real-Time Systems. In *10th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1427. Springer, 546–550.

Brooke, P. 1999. A Timed Semantics for a Hierarchical Design Notation. Ph.D. thesis, University of York.

Butterfield, A., Sherif, A., and Woodcock, J. 2007. Slotted-Circus. In *6th International Conference on Integrated Formal Methods (IFM)*. Lecture Notes in Computer Science, vol. 4591. Springer, 75–97.

Chaki, S., Clarke, E. M., Ouaknine, J., Sharygina, N., and Sinha, N. 2004. State/Event-Based Software Model Checking. In *4th International Conference on Integrated Formal Methods (IFM)*. Lecture Notes in Computer Science, vol. 2999. Springer, 128–147.

Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., and Yovine, S. 2001. TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems. In *13th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2102. Springer, 391–395.

David, A., David, R., and Möller, M. O. 2001. From HUPPAAL to UPPAAL - A Translation from Hierarchical Timed Automata to Flat Timed Automata.

Davies, J. 1993. *Specification and Proof in Real-Time CSP*. Cambridge University Press.

Dill, D. L. 1989. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science, vol. 407. Springer, 197–212.

Dong, J. S., Hao, P., Qin, S. C., Sun, J., and Yi, W. 2004. Timed Patterns: TCOZ to Timed Automata. In *3rd International Conference on Formal Engineering Methods (ICFEM)*. Lecture Notes in Computer Science, vol. 3308. Springer, 483–498.

Dong, J. S., Hao, P., Qin, S. C., Sun, J., and Yi, W. 2008. Timed Automata Patterns. *IEEE Transactions on Software Engineering 34,* 6, 844–859.

Dong, J. S., Hao, P., Sun, J., and Zhang, X. 2006. A Reasoning Method for Timed CSP Based on Constraint Solving. In *8th International Conference on Formal Engineering Methods (ICFEM)*. Lecture Notes in Computer Science, vol. 4260. Springer, 342–359.

Dong, J. S., Mahony, B. P., and Fulton, N. 1999. Modeling Aircraft Mission Computer Task Rates. In *Formal Method (FM)*. Lecture Notes in Computer Science, vol. 1708. Springer, 1855.

Floyd, R. W. 1962. Algorithm 97: Shortest Path. *Communications of ACM 5,* 6, 345.

Gastin, P. and Oddoux, D. 2001. Fast LTL to Büchi Automata Translation. In *14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2102. Springer, 53–65.

Gómez, R. and Bowman, H. 2007. Efficient Detection of Zeno Runs in Timed Automata. In *5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*. Lecture Notes in Computer Science, vol. 4763. Springer, 195–210.

Harel, D. 1997. Some Thoughts on Statecharts, 13 Years Later. In *9th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1254. Springer, 226–231.

Harel, D. and Gery, E. 1997. Executable Object Modeling with Statecharts. *IEEE Computer 30,* 7, 31–42.

Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S. 1994. Symbolic Model Checking for Real-Time Systems. *Information and Computation 111,* 2, 193–244.

Herbreteau, F. and Srivathsan, B. 2010. Efficient On-The-Fly Emptiness Check for Timed Büchi Automata. In *8th International Symposium on Automated Technology for Verification and Analysis (AVTA)*. Lecture Notes in Computer Science. Springer.

Herbreteau, F., Srivathsan, B., and Walukiewicz, I. 2010. Efficient Emptiness Check for Timed Büchi Automata. In *22nd International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 6174. Springer, 148–161.

Hoare, C. A. R. 1985. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall.

Holzmann, G. J. 2003. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley.

Jin, X., Ma, H., and Gu, Z. 2007. Real-Time Component Composition Using Hierarchical Timed Automata. In *Seventh International Conference on Quality Software (QSIC)*. IEEE Computer Society, 90–99.

Lai, L. M. and Watson, P. 1997. A Case Study in Timed CSP: The Railroad Crossing Problem. In *International Workshop on Hybrid and Real-Time Systems (HART)*. Lecture Notes in Computer Science, vol. 1201. Springer, 69–74.

Larsen, K. G., Pettersson, P., and Wang, Y. 1997. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer 1,* 1-2, 134–152.

Lindahl, M., Pettersson, P., and Wang, Y. 2001. Formal Design and Analysis of a Gearbox Controller. *International Journal on Software Tools for Technology Transfer 3,* 3, 353–368.

Liu, Y., Sun, J., and Dong, J. S. 2010. Developing Model Checkers Using PAT. In *8th International Symposium on Automated Technology for Verification and Analysis (AVTA)*. Lecture Notes in Computer Science. Springer.

Lynch, N. A. and Vaandrager, F. W. 1996. Action Transducers and Timed Automata. *Formal Aspects of Computing 8,* 5, 499–538.

Mahony, B. P. and Dong, J. S. 1998. Network Topology and a Case Study in TCOZ. In *11th International Conference of Z Users (ZUM)*. Lecture Notes in Computer Science, vol. 1493. Springer, 308–327.

Mahony, B. P. and Dong, J. S. 2000. Timed Communicating Object Z. *IEEE Transactions on Software Engineering 26,* 2, 150–177.

Nicollin, X. and Sifakis, J. 1994. The Algebra of Timed Processes, ATP: Theory and Application. *Information and Computation 114,* 1, 131–178.

Nicollin, X., Sifakis, J., and Yovine, S. 1992. Compiling Real-Time Specifications into Extended Automata. *IEEE Transactions on Software Engineering 18,* 9, 794–804.

Ouaknine, J. and Worrell, J. 2002. Timed CSP = Closed Timed Safety Automata. *Electrical Notes Theoretical Computer Science 68,* 2.

Ramchandani, C. 1974. Analysis of Asynchronous Concurrent Systems by Timed Petri Nets. Ph.D. thesis, Massachusetts Institute of Technology.

Reed, G. M. and Roscoe, A. W. 1986. A Timed Model for Communicating Sequential Processes. In *13th International Colloquium on Automata, Languages and Programming (ICALP)*. Lecture Notes in Computer Science, vol. 226. Springer, 314–323.

Rokichi, T. G. 1993. Representing and Modeling Digital Circuits. Ph.D. thesis.

Roscoe, A. W. 2001. Compiling Shared Variable Programs into CSP. In *Proceedings of PROGRESS workshop 2001*.

Roscoe, A. W. 2005. On the Expressive Power of CSP Refinement. *Formal Aspects of Computing 17,* 2, 93–112.

Roscoe, A. W., Gardiner, P. H. B., Goldsmith, M., Hulance, J. R., Jackson, D. M., and Scattergood, J. B. 1995. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 1019. Springer, 133–152.

Schneider, S. 1995. An Operational Semantics for Timed CSP. *Information and Computation 116,* 2, 193–213.

Schneider, S. 2000. *Concurrent and Real-time Systems*. John Wiley and Sons.

SIFAKIS, J. 1999. The Compositional Specification of Timed Systems - A Tutorial. In *11th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1633. Springer, 2–7.

SUN, J., LIU, Y., DONG, J. S., LIU, Y., AND SHI, L. Stateful Timed CSP: Models and Experiments. http://www.comp.nus.edu.sg/~pat/rts.

SUN, J., LIU, Y., DONG, J. S., AND PANG, J. 2009. PAT: Towards Flexible Verification under Fairness. In *20th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 5643. Springer.

SUN, J., LIU, Y., DONG, J. S., AND ZHANG, X. 2009. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *11th IEEEInternational Conference on Formal Engineering Methods (ICFEM)*. Lecture Notes in Computer Science, vol. 5885. Springer, 581–600.

TASIRAN, S., ALUR, R., KURSHAN, R. P., AND BRAYTON, R. K. 1996. Verifying Abstractions of Timed Systems. In *7th International Conference on Concurrency Theory (CONCUR)*. Lecture Notes in Computer Science, vol. 1119. Springer, 546–562.

TRIPAKIS, S. 1999. Verifying Progress in Timed Systems. In *5th International AMAST Workshop ARTS on Formal Methods for Real-Time and Probabilistic Systems*. Lecture Notes in Computer Science, vol. 1601. Springer, 299–314.

TRIPAKIS, S. 2009. Checking Timed Büchi Automata Emptiness on Simulation Graphs. *ACM Transactions on Computational Logic 10,* 3, 1–19.

TRIPAKIS, S., YOVINE, S., AND BOUAJJANI, A. 2005. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design 26,* 3, 267–292.

VARDI, M. Y. AND WOLPER, P. 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 332–344.

YI, W. 1991. CCS + Time = An Interleaving Model for Real Time Systems. In *18th International Colloquium on Automata, Languages and Programming (ICALP)*. Lecture Notes in Computer Science, vol. 510. Springer, 217–228.

YI, W., PETTERSSON, P., AND DANIELS, M. 1994. Automatic Verification of Real-time Communicating Systems by Constraint-Solving. In *7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE)*. IFIP Conference Proceedings, vol. 6. Chapman & Hall, 243–258.