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

Jun Sun
Singapore University of Technology and Design
Yang Liu, 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 graphs. 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 CSP [Schneider 2000], etc. The most popular one is Timed Automata [Alur and Dill 1994]. This 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 models and verification case studies. Most importantly, we solve the problem of verification with non-Zenoness assumption.
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. 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 in order to support real-world applications easily. 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 \( P \text{ deadline}[d] \). 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 assump-
A labeled transition system (LTS) is a tuple $L = (S, \text{init}, \Sigma, T)$ where $S$ is a set of states; init $\in S$ is an initial state and $T : S \times \Sigma \times S$ 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 $s \xrightarrow{a} s'$ to denote $(s, a, s') \in T$ 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 $L$ is a sequence of alternating states/events $(s_0, a_0, s_1, a_1, \cdots)$ such that $s_0 = \text{init}$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $i$. The set of runs of $L$ is written as $\text{runs}(L)$.

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

There are two kinds of transitions in $T$, i.e., event transitions $s \xrightarrow{\Sigma} s'$ and time transitions $s \xrightarrow{\tau} s'$. 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 do not exist $\epsilon, a$ and $s'$ such that $s \xrightarrow{\epsilon,a} s'$. A run of $T$ is a sequence $\rho$ of the form $(s_0, (\epsilon_0, a_0), s_1, (\epsilon_1, a_1), \cdots)$ such that $s_0 = \text{init}$ and $s_i \xrightarrow{\epsilon_i,a_i} s_{i+1}$ for all $i$. Given $\rho$, we say that $(s_0, a_0, s_1, a_1, \cdots)$ is an un-timed run of $T$. The set of un-timed runs of $T$ is written as $\text{runs}(T)$.
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 $\rho$ is infinite and $(\epsilon_i + \epsilon_{i+1} + \cdots)$ for all $i$ is unbounded.

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. 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 $T = (S_t, init_t, \Sigma_t \times \mathbb{R}_+, T_t)$ and an LTS $L = (S_u, init_u, \Sigma_u, T_u)$ is a relation $R \subseteq S_t \times S_u$ satisfying the following condition.

$C1$: If $(s_0, s_1) \in R$ and $(s_0, (\epsilon, a), s'_0) \in T_t$ for some $\epsilon$ and $a$, then there exists $s'_1$ such that $(s_1, a, s'_1) \in T_u$ and $(s'_0, s'_1) \in R$;

$C2$: If $(s_0, s_1) \in R$ and $(s_1, a, s'_1) \in T_u$ for some $s'_1$, then there exists some $\epsilon$ and $s'_0$ such that $(s_0, (\epsilon, a), s'_0) \in T_t$ and $(s'_0, s'_1) \in R$;

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

$T$ time-abstract bi-simulates $L$, written as $T \approx L$, 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. $T \approx L \Rightarrow \text{runs}(T) = \text{runs}(L)$. \qed

3. SYNTAX AND OPERATIONAL SEMANTICS

In this section, we introduce Stateful Timed CSP, which is an extension of Timed CSP proposed in [Sun et al. 2009].

3.1 Syntax and Informal Semantics

A Stateful Timed CSP model (hereafter a model) is a tuple $S = (\text{Var}, init_G, P)$ where $\text{Var}$ 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 $\text{Stop}$ does nothing but idling. Process $\text{Skip}$ terminates, possibly after idling for some time. Process $e \rightarrow P$ engages in event $e$ first and then behaves as $P$. Note that $e$ may serve as a synchronization barrier, if combined with parallel composition. In order to seamlessly integrate data operations, we allow sequential programs to be attached with events. Process $a\{\text{program}\} \rightarrow P$ 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 $\text{if} \ (b) \ \{P\} \ \text{else} \ \{Q\}$. Process $P \ | \ Q$ offers an entry point for $P$.

---

\(^2\)Refer to PAT user manual on how to define a type in C# or Java.

\(^3\)The detailed syntax for the sequential program can be found in PAT user manual.
\[ P = \text{Stop} \]  
\[ \text{Skip} \]  
\[ e \to P \]  
\[ a\{\text{program}\} \to P \]  
\[ \text{if } \{b\} \{P\} \text{ else } \{Q\} \]  
\[ P \mid Q \]  
\[ P \setminus X \]  
\[ \text{Wait}[d] \]  
\[ P \text{ timeout}[d] Q \]  
\[ P \text{ interrupt}[d] Q \]  
\[ P \text{ within}[d] \]  
\[ P \text{ deadline}[d] \]  
\[ Q \]

- in-action
- termination
- event prefixing
- data operation prefixing
- conditional choice
- general choice
- hiding
- sequential composition
- parallel composition
- delay*
- timeout*
- timed interrupt*
- timed responsiveness*
- deadline*
- process referencing

Fig. 1. Process constructs

(conditional) choice between \( P \) and \( Q \). 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 \( P \parallel\parallel Q \), which reads as ‘\( P \) interleave \( Q \)’. A process may be given a name, written as \( P \equiv Q \), 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 \( \text{Wait}[d] \) idles for exactly \( d \) time units. In process \( P \text{ timeout}[d] Q \), the first observable event of \( P \) shall occur before \( d \) time units elapse (since process \( P \text{ timeout}[d] Q \) is activated). Otherwise, \( Q \) takes over control after exactly \( d \) time units. In process \( P \text{ interrupt}[d] Q \), if \( P \) terminates before \( d \) time units, \( P \text{ interrupt}[d] Q \) behaves exactly as \( P \). Otherwise, \( P \text{ interrupt}[d] Q \) behaves as \( P \) until \( d \) time units and then \( Q \) takes over. In contrast to \( P \text{ timeout}[d] Q \), \( P \) may engage in multiple observable events before it is interrupted. Process \( P \text{ within}[d] \) must react within \( d \) time units, i.e., an observable event must be engaged by process \( P \) within \( d \) time units. Urgent event prefixing [Davies 1993], written as \( e \to P \), is defined as \( (e \to P) \text{ within}[0] \), i.e., \( e \) must occur as soon as it is enabled. In process \( P \text{ deadline}[d] \), \( P \) 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 real-time systems.

\footnote{For simplicity, we omit external and internal choices [Hoare 1985] in the discussion.}
Example  Let $\delta$ and $\epsilon$ be two constants such that $\delta < \epsilon$. Fischer’s mutual exclusion algorithm is modeled as a model $(V, v, \text{Protocol})$. $V$ contains two variables $\text{turn}$ and $\text{counter}$. 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 $\text{turn}$ to -1 (which denotes that no process is attempting initially) and $\text{counter}$ to 0 (which denotes that no process is in the critical section initially). Process $\text{Protocol}$ is defined as follows.

$$\text{Protocol} \triangleq \text{Proc}(0) || \text{Proc}(1) || \cdots || \text{Proc}(n)$$

$$\text{Proc}(i) \triangleq \text{if}(turn = -1) \{ \text{Active}(i) \} \text{ else } \{ \text{Proc}(i) \}$$

$$\text{Active}(i) \triangleq (\text{update}.i\{turn := i\} \rightarrow \text{Wait}[\epsilon]) \text{ within}[^{\delta}];$$

$$\text{if}(turn = i) \{$$

$$\text{cs}.i\{counter := counter + 1\} \rightarrow$$

$$\text{exit}.i\{counter := counter - 1; turn := -1\} \rightarrow \text{Proc}(i)$$

$$\} \text{ else } \{$$

$$\text{Proc}(i) \}$$

where $n$ is a constant representing the number of processes. Process $\text{Proc}(i)$ models a process with a unique integer identify $i$. If $\text{turn}$ is -1 (i.e., no other process is attempting), $\text{Proc}(i)$ behaves as specified by process $\text{Active}(i)$. In process $\text{Active}(i)$, firstly $\text{turn}$ is set to be $i$ (indicating that the $i$-process is now attempting) by action $\text{update}.i$. Note that $\text{update}.i$ must occur within $\delta$ time units (captured by $\text{within}[\delta]$). Next, the process idles for $\epsilon$ time units (captured by $\text{Wait}[\epsilon]$). It then checks whether $\text{turn}$ 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 $\delta \geq \epsilon$. One way to verify mutual exclusion is to show that $\text{counter} \leq 1$ is always true. We remark that the event names for variable updates (e.g., $\text{update}.i$ and $\text{cs}.i$ and $\text{exit}.i$) 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: $\text{Heart} || \text{Sensing} || \text{Pacing}$ where $\text{Heart}$ models normal or abnormal heart condition; $\text{Sensing}$ and $\text{Pacing}$ 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, \( SA = 0 \). The process is \( AAT \) which is defined as follows.

\[
\begin{align*}
AAT & \equiv \text{Heart} \ || \ \text{Sensing} \ || \ \text{Pacing}(\text{LRI}) \\
\text{Sensing} & \equiv \begin{cases} 
\text{pulseA} \rightarrow \text{senseA} \rightarrow \text{Sensing} & (\text{SA} = 1) \\
\text{else} & \\
\text{pulseA} \rightarrow \text{Sensing} 
\end{cases} \\
\text{Pacing}(X) & \equiv \begin{cases} 
\text{senseA} \rightarrow \text{paceA}\{\text{SA} := 0\} \rightarrow \text{Skip} \ \text{timeout}[X] \ \text{HelpPacing}; \\
\text{Wait}[\text{URI}]; \\
(\text{enableSA}\{\text{SA} := 1\} \rightarrow \text{Pacing}(\text{LRI} - \text{URI})) 
\end{cases} \\
\text{HelpPacing} & \equiv \begin{cases} 
\text{stimu} \rightarrow \text{paceA}\{\text{SA} := 0\} \rightarrow \text{Skip} \ \text{deadline}[0] 
\end{cases}
\end{align*}
\]

where \( \text{URI} \) and \( \text{LRI} \) 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 \( \text{pulseA} \) (i.e., atrium does a pulse) and \( \text{pulseV} \) (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 \( \text{pulseA} \). If \( \text{SA} = 1 \), it engages in event \( \text{senseA} \) immediately once \( \text{pulseA} \) occurs. Initially, process Pacing awaits for event \( \text{senseA} \). If \( \text{senseA} \) occurs before \( X \) time units, action \( \text{paceA} \) occurs (and \( \text{SA} \) is set to 0 so that sensing is paused for a while). If \( \text{senseA} \) 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 \( \text{stimu} \)) and then performing action \( \text{paceA} \). Note that HelpPacing must terminate before 0 time unit (captured by \( \text{deadline}[0] \)), which means that it must immediately perform event \( \text{stimu} \) and action \( \text{paceA} \). Next, \( \text{Wait}[\text{URI}] \) 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 referred as a \emph{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 \( \Sigma_{\tau} \) 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.
\[ \epsilon \leq d \]

\[
(V, \text{Wait}[d]) \xrightarrow{\text{[wait1]}} (V, \text{Wait}[d - \epsilon])
\]

\[
(V, P) \xrightarrow{\text{[to1]}} (V', P')
\]

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

\[
(V, P) \xrightarrow{\text{[to3]}} (V', P')
\]

\[
(V, P \text{ timeout}[d] Q) \xrightarrow{\text{[to4]}} (V', P')
\]

\[
(V, P) \xrightarrow{\text{[ti1]}} (V', P'), \epsilon \leq d
\]

\[
(V, P \text{ interrupt}[d] Q) \xrightarrow{\text{[ti2]}} (V', P')
\]

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

\[
(V, P \text{ interrupt}[0] Q) \xrightarrow{\text{[ti4]}} (V', P')
\]

\[
(V, P) \xrightarrow{\text{[ul2]}} (V', P')
\]

\[
(V, P \text{ within}[d]) \xrightarrow{\text{[wl1]}} (V', P')
\]

\[
(V, P) \xrightarrow{\text{[wl3]}} (V', P')
\]

\[
(V, P \text{ within}[d]) \xrightarrow{\text{[dl1]}} (V', P')
\]

\[
(V, P \text{ deadline}[d]) \xrightarrow{\text{[dl2]}} (V', P')
\]

\[
(V, P) \xrightarrow{\text{[dl3]}} (V', P')
\]

Fig. 2. Concrete firing rules where \( \epsilon \in \Sigma \) and \( a \in \Sigma_r \)
— Rules wait1 and wait2 define the semantics of Wait[d]. Rule wait1 states that the process may idle for an arbitrary amount of time \( \epsilon \) such that \( \epsilon \leq d \). Afterwards, Wait[d] becomes Wait[d−\( \epsilon \)] and the variable valuation is unchanged. Rule wait2 states that the process becomes Skip via a \( \tau \)-transition whenever \( d \) is 0.

— Rules to1 to to4 define \( P \) timeout[d] \( Q \). Rule to1 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 to2 states that if \( d \) is 0, \( Q \) takes over control by a \( \tau \)-transition. Rule to3 states that if \( P \) performs a \( \tau \)-transition, then \( Q \) and timeout operator remain (since an observable event is yet to be performed). Rule to4 states that if \( P \) may idle for less than or equal to \( d \) time units, so does \( P \) timeout[d] \( Q \).

— Rules ti1 to ti4 define \( P \) interrupt[d] \( Q \). Rule ti1 states that if event \( a \) (which may be observable or \( \tau \), 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 to1, the interrupt operator remains. Intuitively, it states that before \( P \) is interrupted, \( P \) behaves freely. Rule ti2 states that if \( P \) may idle for less than or equal to \( d \) time units, so does \( P \) interrupt[d] \( Q \). Rule ti3 states that if \( P \) terminates before being interrupted, then the whole process terminates. Rule ti4 states that if \( d \) is 0, \( Q \) takes over control by a \( \tau \)-transition.

— Rules wi1 to wi3 define \( P \) within[d]. Rule wi1 states that if an observable event \( e \) occurs, then within is discharged, as the requirement is fulfilled. In contrast, rule wi2 states that if instead event \( \tau \) occurs, then within remains. Rule wi3 states that if \( P \) can idle for \( \epsilon \) time units, so does \( P \) within[d] as long as \( \epsilon \leq d \).

— Rules dl1, dl2 and dl3 define \( P \) deadline[d]. P deadline[d] requires \( P \) to terminate (marked by \( \checkmark \) ) before \( d \) time units. Rule dl1 states that \( P \) can do whatever it can before the deadline is expired. Rule dl2 states that if \( P \) terminates, then deadline is discharged. Rule dl3 states if \( P \) can idle for \( \epsilon \) 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]. In particular, we handle data states and extend [Schneider 1995] with rules for timed process constructs like within and deadline.

The following can be established immediately.

**Proposition 3.1.** (1) If \((V, P) \overset{\epsilon}{\rightarrow} (V′, P′)\), then \( V′ = V \). (2) If \((V, P) \overset{\checkmark}{\rightarrow} (V′, P′)\) and \((V′, P′) \overset{\epsilon}{\rightarrow} (V″, P″)\), then \((V, P) \overset{\epsilon + \epsilon}{\rightarrow} (V″, P″)\).

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 \( S = (\text{Var}, \text{init}_G, P) \) be a model. The concrete semantics of \( S \), denoted as \( \mathcal{T}_S \), is a TTS \((S, \text{init}, \Sigma \cup \mathbb{R}^+, T)\) such that \( S \) is a set of reachable concrete system configurations; \( \text{init} = (\text{init}_G, P) \) is the initial configuration; and \( T \) satisfies \(((V, P), x, (V′, P′)) \in T \) if and only if \((V, P) \overset{\epsilon}{\rightarrow} (V′, P′)\).
4. DYNAMIC ZONE ABSTRACTION

$\mathcal{T}_S$ always has infinitely many states, even when all variables have finite domains. For instance, assume $S = (\emptyset, true, P)$ where there is no variable and $P$ is defined as $P \equiv (a \rightarrow (P | c \rightarrow \text{Skip})); \ (b \rightarrow \text{Stop})$, it can be shown that the set of runs of $S$ 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 $S$ is regular if and only if $P$ is a process expression constituted by finitely many process constructs, for every reachable configuration $(V, P)$ of $\mathcal{T}_S$.

Given a regular model $S$, there may still be infinitely many states in $\mathcal{T}_S$ because parameters of timed process constructs in a process (e.g., $d$ in $\text{Wait}[d]$) can take infinitely many different values. Intuitively, the constants capture the reading of the implicit clocks associated with the processes. In this following, we abstract the exact value of the constants by dynamic zone abstraction (initially proposed in [Sun et al. 2009]) so as to generate a finite-state abstraction. We extend [Sun et al. 2009] with a discussion on how Stateful Timed CSP is different from Timed Automata semantically in the end of this section.

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 one 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: $P \equiv (\text{Wait}[5]; \text{Wait}[3]) \text{ Interrupt}[6] Q$. There are at least three implicit clocks: one associated with $\text{Wait}[5]$ (say $t_1$); one with $\text{Wait}[3]$ (say $t_2$); and one with $P$ (because of $\text{Interrupt}[6]$, say $t_3$). Because $\text{Wait}[5]$ and $P$ are activated, clock $t_1$ and $t_3$ have started. In contrast, clock $t_2$ starts only when $\text{Wait}[5]$ 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 if one or more timed processes have just become activated. For simplicity, we write $\text{Wait}[d]_t$, $\text{P timeout}[d]_t$, $\text{P interrupt}[d]_t$, $\text{P within}[d]_t$, $\text{P deadline}[d]_t$ to denote that the process is associated with clock $t$. Given a process $P$ and a clock $t$, we define function $A$ to associate $t$ with $P$. Figure 3 presents the detailed definition. Intuitively speaking, A1 to A5 state that if a process is un-timed and

\textit{This definition adopts 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). 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.}

ACM Journal Name, Vol. V, No. N, Month 20YY.
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 \( A \) 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 \( \text{Wait}[d] \) denotes that \( \text{Wait}[d] \) is associated with clock \( t' \). If a sub-process is activated, then function \( A \) is applied recursively, as captured by A7–10,12–19. The last rule A20 states that if \( P \) is defined as \( Q \), then \( A(P, t) \) can be obtained by applying \( A \) to \( Q \).

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

\[
\begin{align*}
A(\text{Stop}, t) &= \text{Stop} & - A1 \\
A(\text{Skip}, t) &= \text{Skip} & - A2 \\
A(e \rightarrow P, t) &= e \rightarrow P & - A3 \\
A(a\{\text{program}\} \rightarrow P, t) &= a\{\text{program}\} \rightarrow P & - A4 \\
A(\text{if} (b) \{P\} \text{ else } \{Q\}, t) &= \text{if} (b) \{P\} \text{ else } \{Q\} & - A5 \\
A(\text{Wait}[d], t) &= \text{Wait}[d], t & - A6 \\
A(P \text{ timeout}[d], t) &= A(P, t) \text{ timeout}[d], t & - A7 \\
A(P \text{ interrupt}[d], t) &= A(P, t) \text{ interrupt}[d], t & - A8 \\
A(P \text{ within}[d], t) &= A(P, t) \text{ within}[d], t & - A9 \\
A(P \text{ deadline}[d], t) &= A(P, t) \text{ deadline}[d], t & - A10 \\
A(\text{Wait}[d], t') &= \text{Wait}[d], t' & - A11 \\
A(P \text{ timeout}[d], t') &= A(P, t) \text{ timeout}[d], t' & - A12 \\
A(P \text{ interrupt}[d], t') &= A(P, t) \text{ interrupt}[d], t' & - A13 \\
A(P \text{ within}[d], t') &= A(P, t) \text{ within}[d], t' & - A14 \\
A(P \text{ deadline}[d], t') &= A(P, t) \text{ deadline}[d], t' & - A15 \\
A(P \mid Q, t) &= A(P, t) \mid A(Q, t) & - A16 \\
A(P \setminus X, t) &= A(P, t) \setminus X & - A17 \\
A(P; Q, t) &= A(P, t); Q & - A18 \\
A(P \parallel Q, t) &= A(P, t) \parallel A(Q, t) & - A19 \\
A(P, t) &= A(Q, t) \text{ if } P \equiv Q & - A20
\end{align*}
\]
primitive constraint is of the form \( t \sim d \) or \( t_1 - t_2 \sim d \) where \( t, t_1, t_2 \) are clocks; \( d \) is a constant; and \( \sim \) is either, \( \geq, = \) or \( \leq \). 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 \( t_1, t_2, \ldots, t_n \) 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: \( -D^i_j \leq t_i - t_j \leq D^i_j \). Because \( t_0 \) is always 0, we have \(-D^i_0 \leq t_i \leq D^i_0\), 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 \( 0 \leq t_1 \leq 3 \land 0 \leq t_2 \leq 3 \) and \( 0 \leq t_3 \leq 3 \land 0 \leq t_1 - t_2 \leq 3 \land t_2 \leq 1000 \) 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 \( D^i_j \) such that \( D^i_j < 0 \). Intuitively, it means that clock \( t_i \) is constrained to satisfy \( t_i - t_j < 0 \), which is impossible. Furthermore, it can be shown that a DBM in its canonical form represents an empty zone if and only if \( D^i_0 \) 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 \( t_k \) 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
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 $\infty$ for all $i$ such that $i \geq 1$.

4.3 Abstraction

In the following, we present dynamic zone abstraction for Stateful Timed CSP. 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 $\epsilon$-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 form of $(V, P, D)_x (V', P', D')$ where $x \in \Sigma_T$.

We first define a function $idle$ which, given a process (which has been associated with clocks), returns the zone in which the process can idle. Figure 5 shows the detailed definition. Rules $idle1$ to $idle5$ 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 $idle6$ to $idle9$ state that if sub-processes of the process are activated, then function $idle$ is applied to the sub-processes. For instance, process $Wait[d]$ may idle as long as $t$ is less or equal to $d$. Lastly, $idle15$ defines the case for process referencing.

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 $\tau$-transition occurs exactly when $t$ reads $d$. Intuitively, $D^\uparrow \land t = d$ denotes the exact moment when $t$ reads $d$. Afterwards, the process becomes $Skip$. 

Fig. 4. Add clock
Let $\langle t_1, \ldots \rangle$ be a sequence of clocks. Let $S = (\text{Var}, \text{init}_G, P)$ be a model. The time-abstract semantics of $S$, denoted as $\mathcal{L}_S$, is an LTS $(S, \text{init}, \Sigma_{\tau}, T)$ such that $S$ is a set of valid abstract system configurations; $\text{init} = (\text{init}_G, P, \text{true})$ is the initial abstract configuration and $T$ is the smallest transition relation such that:

\begin{align*}
\text{idle}(\text{Stop}) &= \text{true} & \text{-- rule idle1} \\
\text{idle}(	ext{Skip}) &= \text{true} & \text{-- rule idle2} \\
\text{idle}(e \rightarrow P) &= \text{true} & \text{-- rule idle3} \\
\text{idle}(a\{\text{program}\} \rightarrow P) &= \text{true} & \text{-- rule idle4} \\
\text{idle}(\text{if}\ (b)\ \{P\}\ \text{else}\ \{Q\}) &= \text{true} & \text{-- rule idle5} \\
\text{idle}(P \mid Q) &= \text{idle}(P) \land \text{idle}(Q) & \text{-- rule idle6} \\
\text{idle}(P \setminus X) &= \text{idle}(P) & \text{-- rule idle7} \\
\text{idle}(P;\ Q) &= \text{idle}(P) & \text{-- rule idle8} \\
\text{idle}(P || Q) &= \text{idle}(P) \land \text{idle}(Q) & \text{-- rule idle9} \\
\text{idle}(\text{Wait}[d]\_i) &= t \leq d & \text{-- rule idle10} \\
\text{idle}(P \text{ timeout}[d]\_i,\ Q) &= t \leq d \land \text{idle}(P) & \text{-- rule idle11} \\
\text{idle}(P \text{ interrupt}[d]\_i,\ Q) &= t \leq d \land \text{idle}(P) & \text{-- rule idle12} \\
\text{idle}(P \text{ within}[d]\_i) &= t \leq d \land \text{idle}(P) & \text{-- rule idle13} \\
\text{idle}(P \text{ deadline}[d]\_i) &= t \leq d \land \text{idle}(P) & \text{-- rule idle14} \\
\text{idle}(P) &= \text{idle}(Q) & \text{if } P \sqsupseteq Q & \text{-- rule idle15}
\end{align*}

Fig. 5. Idling calculation

---

Rules $\text{ato1}$, $\text{ato2}$ and $\text{ato3}$ define the abstract semantics of $P \text{ timeout}[d]\_i Q$. Rule $\text{ato1}$ states that if a $\tau$-transition transforms $(V, P, D)$ to $(V', P', D')$, then a $\tau$-transition may occur given $(V, P \text{ timeout}[d]\_i, Q, D)$ if zone $D_1 \land D' \land t \leq d$ is not empty. Intuitively, this means that the $\tau$-transition must occur before timeout occurs. Similarly, rule $\text{ato2}$ ensures that the occurrence of an observable event $e$ from process $P$ occurs only before timeout occurs. Rule $\text{ato3}$ states that timeout results in a $\tau$-transition when the reading of $t$ is $d$. Constraint $D_1 \land t = d \land \text{idle}(P)$ ensures that process $P$ may idle until timeout occurs.

Rules $\text{ait1}$, $\text{ait2}$ and $\text{ait3}$ define the abstract semantics of $P \text{ interrupt}[d]\_i Q$. Rule $\text{ait1}$ states that a transition (other than process termination) originated from $P$ may occur only if $t \leq d$, i.e., before interrupt occurs. Rule $\text{ait2}$ states that interrupt results in a $\tau$-transition when the reading of $t$ is $d$. Rule $\text{ait3}$ states that if $P$ terminates before interrupt occurs, then the whole process terminates.

Rules $\text{awi1}$ and $\text{awi2}$ define the abstract semantics of $P \text{ within}[d]\_i$. Rule $\text{awi1}$ states that if a $\tau$-transition occurs within $d$ time units, then the resultant process is of the form $P' \text{ within}[d]\_i$, which means that it is yet to perform some observable event before $d$ time units. Rule $\text{awi2}$ states that once an observable event occurs, the within construct is removed.

Rules $\text{adl1}$ and $\text{adl2}$ define the abstract semantics of $P \text{ deadline}[d]\_i$. Rule $\text{adl1}$ ensures that all transitions of $P$ must occur within $d$ time units. Rule $\text{adl2}$ states that if $P$ terminates (by engaging in $\checkmark$), then $\text{deadline}$ is removed.

Using the abstract firing rules, we can generate an abstract LTS which captures the abstract semantics of a model.
An abstract firing rule is applied to get a target configuration \((V, \text{Skip}, D^\top \land t = d)\)

\[(V, P, D) \xrightarrow{\text{ato1}} (V', P', D')\]

\[(V, P \text{ timeout}[d], Q, D) \xrightarrow{\text{ato2}} (V', P', D')\]

\[(V, P \text{ timeout}[d], Q, D) \xrightarrow{\text{ato3}} (V', P', D^\top \land D' \land t \leq d)\]

\[(V, P \text{ timeout}[d], Q, D) \xrightarrow{\text{ato3}} (V', Q, D = d \land \text{idle}(P))\]

\[(V, P, D) \xrightarrow{\text{ait1}} (V', P', D')\]

\[(V, P \text{ interrupt}[d], Q, D) \xrightarrow{\text{ait2}} (V', P', D^\top \land D' \land t \leq d)\]

\[(V, P \text{ interrupt}[d], Q, D) \xrightarrow{\text{ait3}} (V', Q, D^\top \land t = d \land \text{idle}(P))\]

\[(V, P, D) \xrightarrow{\text{awi1}} (V', P', D')\]

\[(V, P \text{ within}[d], D) \xrightarrow{\text{awi2}} (V', P' \text{ within}[d], D^\top \land D' \land t \leq d)\]

\[(V, P, D) \xrightarrow{\text{adl1}} (V', P', D')\]

\[(V, P \text{ deadline}[d], D) \xrightarrow{\text{adl2}} (V', P' \text{ deadline}[d], D^\top \land D' \land t \leq d)\]

\[(V, P \text{ deadline}[d], D) \xrightarrow{\text{adl2}} (V', P', D^\top \land D' \land t \leq d)\]

Fig. 6. Abstract Firing Rules

For all \((V, P, D) \in S\), if \(t\) is the first clock in the sequence which is not in \(cl(P)\), and if \((V, A(P, t), D \land t = 0) \xrightarrow{a} (V', P', D')\), then \(((V, P, D), a, (V', P', D'[cl(P')])) \in T\).

Because of zone abstraction, \(L_S\) is also referred to as a zone graph. Informally, \(L_S\) 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 \((V, A(P, t), D \land t = 0)\), 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 $L_S$ is illustrated in the following example.

**Example** Assume that a model $S = (\emptyset, true, P)$ such that

$$P \triangleq (a \rightarrow Wait[5]; b \rightarrow Stop) \ \text{interrupt}[3] \ c \rightarrow P$$

Intuitively, event $b$ never occurs because interrupt always occurs first. The left part of Figure 7 shows the $L_S$ (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 $c_0 = (\emptyset, P, true)$.

—Starting with $c_0$, we apply $A$ to $P$ with $t_1$ to get

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

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

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

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

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

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

Next, we can apply rule $ait1$ or $ait2$. Applying rule $ait1$ to $c_5$, we get zone

$(0 \leq t_1 \leq 3 \land t_2 = 0) \lor (0 \leq t_1 \leq 3 \land t_2 = 5)$. By DBM operations, this zone can be shown to be empty and therefore this transition is invalid. Intuitively, this is because $(0 \leq t_1 \leq 3 \land t_2 = 0)$ is equivalent to $0 \leq t_1 - t_2 \leq 3$. Apply rule $ait2$ to $c_5$, we get

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

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. $\Box$

### 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 regular Stateful Timed CSP is equivalent to Closed Timed Safety Automata with $\tau$-transitions. In the original theory of 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 [Henzinger et al. 1994] to specify progress properties using local invariant conditions instead.

ACM Journal Name, Vol. V, No. N, Month 20YY.
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 \( \tau \)-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 \( \text{within} \) and \( \text{deadline} \). 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 \( \text{deadline} \) and \( \text{within} \) can be translated to state invariants in Timed (Safety) Automata. For instance, we have shown in [Dong et al. 2008] that \( \text{deadline} \) 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 \( \tau \)-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 \( \text{idle} \) 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. Compared to [Sun et al. 2009], we show in addition that model checking non-Zenoness is feasible.

In order to apply model checking techniques, we first establish that \( \mathcal{L}_S \) is finite given any model \( S \).

**Theorem 5.1.** \( \mathcal{L}_S \) is finite for any regular model \( S \).
Proof By definition, $\mathcal{L}_S$ 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 $\mathcal{L}_S$ preserves a large class of interesting properties.

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

**Proof** Let $\mathcal{L}_S = (S_a, init_a, \Sigma_T, T_a)$ and $\mathcal{T}_S = (S_c, init_c, \mathbb{R} \cup \Sigma_T, T_c)$. 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 $(V_c, P_c) \in S_c$; $(V_a, P_a, D) \in S_a$, $((V_c, P_c), (V_a, P_a, D)) \in \mathcal{R}$ 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 abstracted 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 $P_a = \text{Wait}[3]$; $\text{Wait}[5]$ and $P = \text{Wait}[4]$; $\text{Wait}[5]$, 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 $\text{Wait}[d]$ or $P$ $\text{timeout}[d]$ $Q$.

Notice that the first step of Definition 4.2 is to apply $A$ to $P_a$. Let the resultant process be $P'_a$ and the resultant zone be $D'$.

—If $P_c$ is $\text{Wait}[d]$ and $((V_c, P_c), (V_a, P_a, D_a)) \in \mathcal{R}$, $P'_a$ is $\text{Wait}[d']$, such that $t = d'$ satisfies $D'_{a}[t]$. We first show that $C1$ is satisfied. By rule $\text{Wait}1$ and $\text{Wait}2$, $(V_c, \text{Wait}[d]) \xrightarrow{d} (V_c, \text{Skip})$. By rule $\text{await}$, $(V_a, P'_a, D') \xrightarrow{c} (V_a, \text{Skip}, D'_a \land t = d)$ and thus $(V_a, P'_a, D'_a) \xrightarrow{\tau} (V_a, \text{Skip}, D') \in T_a$ where $D' = \text{true}$ (since

\(^6\)It is clear that this assumption can be relaxed to allow finite domain parameters.

ACM Journal Name, Vol. V, No. N, Month 20YY.
\( \text{cl}(\operatorname{Skip}) = \emptyset \). It is trivial to show \((V_c, \operatorname{Skip}) \approx (V_a, \operatorname{Skip}, \text{true})\). Further, \(((V_c, \operatorname{Skip}), (V_a, \operatorname{Skip}, \text{true})) \in \mathcal{R} \). Similarly, we can show that \(C2\) is satisfied.

If \(P_e\) is \(P\ \text{timeout}[d] Q\), then \(P'_e\) is \(P' \text{timeout}[d] Q\) such that \(P\) is abstracted by \(P'\) with \(D'\). Assume \((V_c, P) \overset{\epsilon}{\sim} (V_1, P_1)\) for some \(\epsilon \leq d\). By rule \(\text{to4}\) and \(\text{to1}\), \((V_c, P_e) \overset{\epsilon}{\sim} (V_1, P_1)\). By induction hypothesis, \((V_a, P', D') \overset{\epsilon}{\sim} (V_1, P'_1, D'_1)\) 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 \(\text{ato2}\), \((V_c, P'_e, D') \overset{\epsilon}{\sim} (V_1, P'_1, D'_1 \wedge D^1 \wedge t \leq d)\). By assumption, \(\epsilon \leq d\) and thus it is easy to show that \(P'_1\) with \(D'_1 \wedge D^1 \wedge t \leq d\) abstracts \(P_1\). Thus, \(C1\) is satisfied. If \((V_a, P', D') \overset{\epsilon}{\sim} (V_1, P'_1, D'_1)\), \((V_a, P'_a, D') \overset{\epsilon}{\sim} (V_1, P'_1, D'_1 \wedge D^1 \wedge t \leq d)\) by rule \(\text{ato2}\). Because \(D'_1 \wedge D^1 \wedge t \leq d\) is not empty by definition, the transition \((V_a, P', D') \overset{\epsilon}{\sim} (V_1, P'_1, D'_1)\) 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 \((V_c, P_c) \overset{\epsilon}{\sim} (V_1, P_1)\) and \((V_1, P'_1, D'_1) \approx (V_1, P_1)\). By rules \(\text{to4}\) and \(\text{to1}\), \((V_c, P_e) \overset{\epsilon}{\sim} (V_1, P_1)\). It can be shown that \(P'_1\) with \(D'_1 \wedge D^1 \wedge t \leq d\) abstracts \(P_1\) and thus, \(C2\) is satisfied in the case. Similarly, we prove the case where \((V_c, P \text{timeout}[d] Q) \overset{\epsilon}{\sim} (V_1, P_1 \text{timeout}[d] Q)\) for some \(\epsilon \leq d\) or \((V_c, P \text{timeout}[d] Q) \overset{d, \tau}{\rightarrow} (V_1, Q)\).

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. \(\square\)

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 \(\phi\), a Büchi automaton \(\mathcal{B}\) equivalent to the negation of \(\phi\) 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 \(\phi\) \textit{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 \(\langle \emptyset, \emptyset, P \mid a \rightarrow \text{timeout}[1]\rangle\) where \(P \equiv a \rightarrow P \mid b \rightarrow \text{Skip}\). 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 \(\rho = \langle s_0, (s_0, a_0), s_1, (s_1, a_1), \cdots \rangle\) of \(\mathcal{T}_S\), there is a corresponding \(\pi = \langle s_0, a_0, s_1, a_1, \cdots \rangle\) in \(\text{runs}(\mathcal{L}_S)\). We say that \(\rho\) is an instance of \(\pi\) or equivalently \(\pi\) abstracts \(\rho\).
Let $t$ constraints on $A$ cannot be modified or reset before it is expired.

ACM Journal Name, Vol. V, No. N, Month 20YY.

The following is true about $A$: for every clock $t$, assume $\phi_i$ and $\phi_j$ 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 $\phi_i$ can always satisfy $\phi_j$ 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 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 $t$. In other words, every clock acts as a count-down clock which cannot be modified or reset before it is expired.

Next, we establish a necessary and sufficient condition to check whether an abstract run is Zeno or not. Because $L_\mathcal{S}$ is finite, an infinite run $\pi$ of $L_\mathcal{S}$ must visit a finite set abstract configurations, denoted as $\text{inf}(\pi)$, infinitely often. Let $\text{loopCLK}(\pi)$ be the set $\{x | \forall (V, P, D) \in \text{inf}(\pi), x \in \text{cl}(D)\}$, i.e., the clocks which are present in every configuration which is visited infinitely often. A transition of $L_\mathcal{S}$ is sometimes written in the form $(V, P, D) \overset{a, X}{\longrightarrow} (V', P', D')$ 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 $D' \Rightarrow t = 0$. Notice that a transition which is not instantaneous might in fact be forced to occur immediately with any delay. For instance, let $P \equiv (a \rightarrow \text{Wait}[5]) \text{ deadline}[5]; P$. The transition labeled with $a$ is not instantaneous by definition. It is nonetheless constrained to occur immediately if we consider the loop generated by $P$, i.e., the run contains infinitely many event $a$, $\tau$ and $\checkmark$ (which are generated by $\text{Wait}[5]$).

**Theorem 5.3.** Let $S$ be a model; $\pi$ be a run of $L_\mathcal{S}$. $\pi$ is non-Zeno if and only if $\text{loopCLK}(\pi) = \emptyset$ and not all infinitely visited transitions are instantaneous.
Proof. Let ρ be a run of T_S and π be the corresponding abstract run of L_S.

\[ \rho = s_0 \xrightarrow{c_0,a_0} s_1 \xrightarrow{c_1,a_1} \ldots \xrightarrow{c_i,a_i} \ldots \]
\[ \pi = t_0 \xrightarrow{t_0,a_0} \{X_0\} t_1 \xrightarrow{t_1,a_1} \{X_1\} \ldots \xrightarrow{t_i,a_i} \{X_i\} \ldots \]

Only-if: We show that if ρ is non-Zeno, then loopCLK(π) = ∅. Assume that \( t_i \in \text{loopCLK}(\pi) \). 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(π) = ∅. 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(π) = ∅ and there is at least one infinitely often visited transition, say \( (V, P, D) \xrightarrow{t,a,X} (V', P', D') \), that is not instantaneous, then π is non-Zeno. Because \( D' \) does not imply \( t = 0 \), all clocks \( t' \) in \( D' \) must satisfy \( t' \leq d \) for some positive integer \( d \). Because the constraint on \( t' \) can only get weakened (as we argue above), strictly positive number of time units must be able to elapse at some transition. Because loopCLK(π) = ∅, 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 \( \{ x \mid \forall(V, P, D) \in \text{scc}. x \in \text{cl}(D) \} \).

**Theorem 5.4.** Let S be a model. T_S is non-empty if and only if there exists a reachable maximal SCC scc in L_S such that loopCLK(scc) = ∅ and not all transitions connecting two states in scc are instantaneous.

**Proof** (Only-if) Let π be a non-Zeno run of T_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(π) = ∅ and therefore loopCLK(scc) = ∅. 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 5.3. □

Given an SE-LTL formula φ, a Büchi automaton B equivalent to the negation of φ, model checking with non-Zenoness assumption is to construct the product of B and L_S and then search for an accepting run of the product whose projection on L_S is non-Zeno. By Theorem 5.4, 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; Liu et al. 2011]. 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; 2011]. The model checker for Stateful Timed CSP is built as one plug-in 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-time 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:\(^\text{NoOfFloors}\) (the number of floors); \(^\text{NoOfLifts}\) (number of lifts); \(^\text{Off}\) of value 0; \(^\text{Up}\) of value 1; \(^\text{Down}\) of value -1; and \(^\text{Both}\) 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 \(^\text{FloorButtons}\) of length \(^\text{NoOfFloors}\). Each variable in the array has four possible values: \(^\text{Off}\) (i.e., it is not on), \(^\text{Up}\) (i.e., upward traveling has been requested), \(^\text{Down}\) (i.e., downward traveling has been requested) or \(^\text{Both}\) (i.e., both directions have been requested). The following models the building.

\(^\text{Readers are recommended to download PAT at http://www.patroot.com and try out the RTS module.}\)

ACM Journal Name, Vol. V, No. N, Month 20YY.
1. Press(floor, direction) ≡ request.floor.direction{
   2.     if (FloorButtons[floor] = None){
   3.         FloorButtons[floor] := direction
   4.     }
   5.   else if (FloorButtons[floor] ≠ direction){
   6.         FloorButtons[floor] := Both
   7.     }
   8. } → Skip
   9. TopFloor ≡ Press(NoOfFloors − 1, Down); TopFloor
10. GroundFloor ≡ Press(0, Up); GroundFloor
11. MiddleFloor(n) ≡ (Press(n, Down) | Press(n, Up)); MiddleFloor(n)
12. Building ≡ TopFloor || GroundFloor || (||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 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 ≡ open → (Cycle; close → Skip) deadline[maxTime]; Door
Cycle ≡ toOpen → opened → conf → Wait[minTime]; Closing
Closing ≡ (closed → Skip) interrupt (sensor → toOpen → opened → 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 min.Time 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 →. Furthermore, the deadline in process Door is used to ensure that a door cannot remain opened forever. Notice that this is not exactly the case in reality. Nonetheless, this simplifying assumption allows us to ignore unlikely (not still possible) scenarios (e.g., an obstruction keeps the door open forever) and thus be able to verify liveness properties (e.g., always
eventually a lift will serve some request.

A shaft takes input from the controller and transports a lift from one floor to another. The time required for transit depends on the distance to be traveled. It is modeled as follows.

\[
Shaft(i) \equiv move?[id = i]id.n.dir \rightarrow Wait[n * movingTime + delayTime];
\]

\[
arrive \rightarrow Shaft(i)
\]

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. The constant 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. Notice that in this model, we assume that once a lift is assigned with a destination, it will not be interrupted to serve a newly arrived request on its way. This is a simplifying assumption made to keep the model small enough for presentation.

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 \(NoOfLifts \times NoOfFloors\). Entry \(IntReq[i][j] = true\) if and only if there is an internal request in \(i\)-lift for \(j\)-floor.

\[
\begin{align*}
InternalQ(i) \equiv & \quad intReq.0\{IntReq[i][0] := 1\} \rightarrow InternalQ(i) | \\
& \quad intReq.1\{IntReq[i][1] := 1\} \rightarrow InternalQ(i) | \cdots | \\
& \quad intReq.(NoOfFloors - 1)\{IntReq[i][NoOfFloors - 1] := 1\} \rightarrow InternalQ(i)
\end{align*}
\]

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.

The following models a lift controller. The three parameters denote the lift identity, its current floor and direction 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...

\footnote{Refer to a different model of multi-lifts in PAT which models interruptible lifts.}
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 request9.

\[
\text{LiftCtrl}(i, \text{fl}, \text{dir}) \doteq \text{check!fl.dir.call(GetDesInt, IntReq, fl, dir, NoOfFloors, i)} \\
\rightarrow \text{check?des} \rightarrow \text{case} \{ \\
\text{des = fl : open} \rightarrow \text{conf} \rightarrow \text{ClearReq}(i, \text{fl}, \text{dir}); \\
\text{close} \rightarrow \text{LiftCtrl}(i, \text{fl}, \text{dir}) \\
\text{des > fl : move!((des - fl).Up) \rightarrow arrive \rightarrow open} \\
\rightarrow \text{conf} \rightarrow \text{ClearReq}(i, \text{des}, \text{Up}); \text{close} \rightarrow \text{LiftCtrl}(i, \text{des}, \text{Up}) \\
\text{0} \leq \text{des} < \text{fl} : \text{move!}((\text{fl} - \text{des}).\text{Down}) \rightarrow \text{arrive} \rightarrow \text{open} \\
\rightarrow \text{conf} \rightarrow \text{ClearReq}(i, \text{des}, \text{Down}); \text{close} \rightarrow \text{LiftCtrl}(i, \text{des}, \text{Down}) \\
\text{default : Wait[delayTime]; LiftCtrl(i, fl, dir)} \\
\}
\]

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

\[
\text{Controller} \doteq \text{check?fl.dir.des} \rightarrow \\
\text{check!call(GetDesExt, FloorButtons, fl, dir, des, NoOfFloors)} \rightarrow \text{Controller}
\]

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 case \( \{ c_0 : P_0 \quad c_1 : P_1 \quad \cdots \} \) is equivalent to if \( (c_0)\{P_0\} \) else \{if \( (c_1) \{P_1\} \) 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., \( \text{des} = \text{fl} \)), 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 \( \text{des} - \text{fl} \) floors (downwards for \( \text{fl} - \text{des} \) 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.

\[
\text{ClearReq}(i, \text{fl}, \text{dir}) \doteq \text{clearRequest}\{ \\
\text{IntReq[i][fl]} := 0; \\
\text{if(FloorButtons[fl] = dir)}\{\text{FloorButtons[fl]} := \text{None}\} \\
\text{else if(FloorButtons[fl]! = None)}\{\text{FloorButtons[fl]} := -1 * \text{dir}\}\} \\
\rightarrow \text{Skip}
\]

9Details of the C# methods are skipped as they are less interesting.

ACM Journal Name, Vol. V, No. N, Month 20YY.
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., \( des = -1 \)), 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.

\[
\text{Lift}(i) \triangleq (\text{Shaft}(i) \parallel \text{Door} \parallel \text{LiftCtrl}(i, 0, \text{Up}) \parallel \text{InternalQ}(i))
\not\{\text{open}, \text{conf}, \text{close}, \text{arrive}\};
\]

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
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.

\[
\text{Lifts()} \equiv \big|\big| \big|_{i=0}^{N_{\	ext{NoOfLifts}}-1} \text{Lift}(i)
\]

\[
\text{System()} \equiv \text{Lifts()} \parallel \text{CentralController()} \parallel \text{Building()}
\]

This model demonstrates how Stateful Timed CSP may be applied to model hi-
erarchical 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 Fis-
cher’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 num-
er of processes. LTL 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 \( NZ \). Column \( \#St \) shows the
number states in the zone graphs. 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 cur-
currently handles in average about 20K 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
computational overhead and may be even more efficient. Compared to other work
on model checking with non-Zenoness [Tripakis 1999; Gómez and Bowman 2007;
<table>
<thead>
<tr>
<th>Model</th>
<th>Property</th>
<th>#St</th>
<th>#Clock</th>
<th>Z(s)</th>
<th>NZ(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacemaker</td>
<td>reachability</td>
<td>450K</td>
<td>2</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Lift (2floor; 2lift)</td>
<td>reachability</td>
<td>197K</td>
<td>4</td>
<td>54</td>
<td>53</td>
</tr>
<tr>
<td>Lift (3floor; 2lift)</td>
<td>reachability</td>
<td>943K</td>
<td>4</td>
<td>310</td>
<td>305</td>
</tr>
<tr>
<td>Lift (2floor; 2lift)</td>
<td>LTL</td>
<td>748K</td>
<td>4</td>
<td>346</td>
<td>305</td>
</tr>
<tr>
<td>Lift (3floor; 2lift)</td>
<td>LTL</td>
<td>4.3M</td>
<td>4</td>
<td>6102</td>
<td>3948</td>
</tr>
<tr>
<td>Fischer*5</td>
<td>LTL</td>
<td>15K</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fischer*7</td>
<td>LTL</td>
<td>857K</td>
<td>7</td>
<td>229</td>
<td>105</td>
</tr>
<tr>
<td>Railway*4</td>
<td>LTL</td>
<td>2K</td>
<td>4</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Railway*6</td>
<td>LTL</td>
<td>74K</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Railway*8</td>
<td>LTL</td>
<td>4.3M</td>
<td>4</td>
<td>1536</td>
<td>1200</td>
</tr>
<tr>
<td>CSMA*6</td>
<td>LTL</td>
<td>14K</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CSMA*9</td>
<td>LTL</td>
<td>295K</td>
<td>5</td>
<td>75</td>
<td>71</td>
</tr>
<tr>
<td>CSMA*12</td>
<td>LTL</td>
<td>4.7M</td>
<td>5</td>
<td>11475</td>
<td>9303</td>
</tr>
<tr>
<td>FDDI*4</td>
<td>LTL</td>
<td>46K</td>
<td>6</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>FDDI*5</td>
<td>LTL</td>
<td>6.4M</td>
<td>7</td>
<td>1461</td>
<td>1438</td>
</tr>
</tbody>
</table>

Table I. Experiment results

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 (version 3.3) and Uppaal (version 4.1). For simplicity, all properties are reachability (without non-Zenoness). 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 PAT in many cases. 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 cannot be abstracted as an array of numbers because the system structure in Stateful Timed CSP varies through transitions. Furthermore, our zone graphs may contain more nodes due to the extra $\tau$-transitions introduced by the compositional process constructs, e.g., the $\tau$-transition generated by abstract firing rule $\text{ato}3$. Combined with parallel composition, these $\tau$-transitions may result in a large number of additional states. In hand-crafted Uppaal models, the $\tau$-transitions are often removed by carefully manipulating the clock guards or grouping clock guards and events into the same transition. Removing the extra $\tau$-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 column Uppaal+$\tau$-o shows the verification time using Uppaal without extrapolation (and with the same extra $\tau$-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
Table II. Experiment results

<table>
<thead>
<tr>
<th>Model</th>
<th>#Clocks</th>
<th>Without Non-Zenoness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer*5</td>
<td>5</td>
<td>PAT(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPPAAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAT(s)</td>
</tr>
<tr>
<td>Railway*6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPPAAL</td>
</tr>
<tr>
<td>CSMA*6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>CSMA*8</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>CSMA*10</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

experiments, PAT uses less clocks than UPPAAL. It remains as 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 reasonably efficient and supports alternative ways of specifying properties (e.g., in SE-LTL, trace-refinement).

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]. Stateful 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 that Uppaal can be applied. 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 Srivatsan 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 all clocks are bounded from above and cannot be reset arbitrarily.
As a result, detecting Zeno runs based on zone graphs is feasible. 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-Zenoness 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


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.


