Parameter Synthesis for Hierarchical Concurrent Real-Time Systems (Report Version)

Étienne André[1], *Yang Liu*[2], *Jun Sun*[3] *and Jin-Song Dong*[4]
[1] *LIPN, CNRS UMR 7030, Université Paris 13, France*
[2] *Temasek Laboratories, National University of Singapore*
[3] *Singapore University of Technology and Design*
[4] *School of Computing, National University of Singapore*

Abstract

Modeling and verifying complex real-time systems, involving timing delays, are notoriously difficult problems. Checking the correctness of a system for one particular value for each delay does not give any information for other values. It is hence interesting to reason parametrically, by considering that the delays are parameters (unknown constants) and synthesize a constraint guaranteeing a correct behavior. We present here Parametric Stateful Timed CSP, viz., a parameterization of Stateful Timed CSP, a language capable of specifying hierarchical real-time systems with complex data structures. Although we prove that the synthesis is undecidable in general, we present an algorithm for efficient parameter synthesis that behaves well in practice.

Keywords: CSP, parametric timed verification, control, robustness, refinement.

1 Introduction

The specification and verification of real-time systems, involving complex data structures and timing delays, are notoriously difficult problems. The correctness of such real-time systems usually depends on the values of these timing delays. One can check the correctness for one particular value for each delay, using classical techniques of timed model checking, but this does not guarantee the correctness for other values. Actually, checking the correctness for all possible delays, even in a bounded interval, would require an infinite number of calls to the model checker, because those delays can have real (or rational) values. It is therefore interesting to reason *parametrically*, by considering that these delays are unknown constants, or *parameters*, and try to synthesize a constraint (i.e., a conjunction of linear inequalities) on these parameters guaranteeing a correct behavior.

Motivation We are interested here in the *good parameters problem* for real-time systems: "find a set of parameter valuations for which the system is correct". This problem stands between verification and control, in the sense that we actually change (the timing part of) the system in order to guarantee some property. Furthermore, we aim at defining a formalism that is intuitive, powerful (with use of external variables, structures and user defined functions), and allowing efficient parameter synthesis and verification.

Parameter Synthesis for Timed Concurrent Systems Timed Automata (TAs) are finite control automata equipped with *clocks*, that are compared with *timing delays* in guards and invariants [2]. TAs have been efficiently used over the last decade to verify timed systems, in particular using the UPPAAL model checker [26]. The parametric extension of TAs (viz., *parametric timed automata*, or *PTAs*) allows the use of parameters within guards and invariants [3].

The parameter design problem for PTAs was formulated in [21], where a straightforward solution is given, based on the generation of the whole state space – which is unfortunately unrealistic in most cases. The HYTECH model checker, one of the first for parametric timed (actually hybrid) automata, allowed to solve several case studies; Unfortunately, it can hardly verify even medium sized examples due to exact arithmetics with limited precision and static composition of automata, quickly leading to memory overflows. The parameter synthesis problem has then been applied in particular to communication protocols (e.g., Bounded Retransmission protocol [16] or Root Contention protocol [15] using TREX [7]) and asynchronous circuits (e.g., [38, 14]). Although drastic optimizations were developed for Timed Automata, in particular using DBMs, most of them do not apply to the parametric framework, or to only partially parameterized systems (e.g., [10], where a non-parametric model is verified against a parameterized formula). In [5], the *inverse method* synthesizes constraints for fully parameterized systems modeled using PTAs. Different from CEGAR-based methods, this original semi-algorithm is based on a "good" parameter valuation π, and synthesizes a constraint guaranteeing the same time abstract behavior as for π, thus providing the system with a criterion of robustness. As an interesting consequence, the preservation of the time-abstract behavior guarantees the preservation of linear time properties.

The authors of [24] synthesize a set of parameter valuations under which a given property specified in the existential part of CTL without the next operator (viz., the ECTL$_{-X}$ logic) holds in a system modeled by PTAs. This is done by using bounded model checking techniques applied to PTAs.

In [25], parametric analysis of scheduling problems is performed, based on the process algebra ACSR-VP. Constraints are synthesized using symbolic bisimulation methods, guaranteeing the feasibility of a scheduling problem. This work is closer to our approach, in the sense that it synthesizes timing parameters in a process algebra; however, it is dedicated to scheduling problems only, whereas our approach is general.

Semi-algorithms (i.e., if the algorithm terminates, then the result is correct) have been proposed in [36] for synthesizing parameters for Time Petri Nets with stopwatches. Different from our setting, the constraint satisfies a formula expressed using a non-recursive subset of parametric TCTL; furthermore, their implementation does not allow the use of elaborated data structures.

Stateful Timed CSP CSP (Communicating sequential processes) [22] is a powerful event based formalism for describing patterns of interaction in concurrent systems. Timed CSP (see, e.g., [32]) extends CSP with timed constructs for reasoning about real-time systems. Stateful Timed CSP (STCSP) extends Timed CSP with more timed constructs and shared variables in order to specify hierarchical complex real-time systems [35]. Through dynamic zone abstraction, *finite-state* zone graphs can be generated automatically from STCSP models, which are subject to model checking. Like TAs, STCSP models suffer from Zeno runs, i.e., runs which take infinitely many steps within finite time. Unlike TAs, model checking with non-Zenoness in STCSP can be easily achieved, based on the zone graph.An advantage of Timed CSP over TAs is the lower number of clocks necessary to verify the systems. Indeed, unlike TAs, clocks are *implicit* in STCSP, and are only activated when necessary.

Contribution We present here Parametric Stateful Timed CSP (PSTCSP). First, this parameterization of STCSP is a powerful language capable of specifying hierarchical real-time systems with shared variables and complex, user-defined data structures, in an intuitive manner.

Second, although we show that the emptiness problem is undecidable for PSTCSP, we develop and compare two semi-algorithms for parameter synthesis. The first one, computing all reachable states, allows the application of finite state timed model checking techniques defined in [35], but does not often terminate. We also extend the inverse method [5] to PSTCSP, and give a sufficient termination condition; this algorithm behaves very well in practice, allowing efficient parameter synthesis even for fully parameterized systems, i.e., where all timing delays are parametric.

Third, the implementation of PSTCSP within the PAT model checker offers both an intuitive modeling facility using a graphical interface, and efficient algorithms for verification and parameter synthesis.

PSTCSP shares similar design principles with integrated specification languages like TCOZ [27] and CSP-OZ-DC [23]. The main idea is to treat sequential terminating programs (rather than Z or Object-Z), which may indeed be C# programs, as internal events. The result is a highly expressive modeling language that can be automatically analyzed by tools.

Plan of the Paper We recall preliminary notions in Section 2. We introduce PSTCSP in Section 3 and study its expressiveness and decidability questions in

Section 4. We introduce algorithms for parameter synthesis in Section 5, and apply them to case studies. We conclude in Section 6.

2 Preliminaries

Finite-Domain Variables We assume a finite set $\mathcal{V}ar$ of finite-domain *variables*. Given $Var \subset \mathcal{V}ar$, a *variable valuation* is a function assigning to each variable a value in its domain. We denote by $\mathcal{V}(Var)$ the set of all variable valuations.

Constraints on Clocks and Parameters Let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers. We assume a set $\mathcal{X}$ of *clocks*, disjoint with $\mathcal{V}ar$. A clock is a variable with value in $\mathbb{R}_{\geq 0}$. All clocks evolve linearly at the same rate. Given a finite set $X = \{x_1, \dots, x_H\} \subset \mathcal{X}$, a *clock valuation* is a function $w : X \rightarrow \mathbb{R}_{\geq 0}$ assigning a non-negative real value to each clock. We will often identify a valuation w with the point $(w(x_1), \dots, w(x_H))$. Given $d \in \mathbb{R}_{\geq 0}$, we use $X + d$ to denote $\{x_1 + d, \dots, x_H + d\}$.

We also assume a fixed set $\mathcal{U}$ of *parameters*, i.e., unknown constants, disjoint with $\mathcal{V}ar$ and $\mathcal{X}$. Given a finite set $U = \{u_1, \dots, u_M\} \subset \mathcal{U}$, a *parameter valuation* is a function $\pi : U \rightarrow \mathbb{R}_{\geq 0}$ assigning a non-negative real value to each parameter. There is a one-to-one correspondence between valuations and points in $(\mathbb{R}_{\geq 0})^M$. We will often identify a valuation π with the point $(\pi(u_1), \dots, \pi(u_M))$.

Given $X \subset \mathcal{X}$ and $U \subset \mathcal{U}$, an inequality over X and U is $e \prec e'$, where $\prec \in \{<, \leq\}$, and e, e' are two linear terms of the form $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ with $z_i \in X \cup U$, $\alpha_i \in \mathbb{R}_{\geq 0}$ for $1 \leq i \leq N$, and $d \in \mathbb{R}_{\geq 0}$. We define similarly inequalities over X (resp. U). A constraint is a conjunction of inequalities. We denote by $\mathcal{K}_{X \cup U}$ the set of all constraints over X and U, and similarly for $\mathcal{K}_X$ and $\mathcal{K}_U$. In the sequel, we use the following conventions: w (resp. π) denotes a clock (resp. parameter) valuation; J denotes an inequality over U; $D \in \mathcal{K}_X$; $K \in \mathcal{K}_U$; and $C \in \mathcal{K}_{X \cup U}$.

We denote by $D[w]$ the expression obtained by replacing in D each clock x with $w(x)$. If $D[w]$ evaluates to true, we say that w *satisfies* D (denoted by $w \models D$). We denote by $C[\pi]$ the constraint over X obtained by replacing in C each $u \in U$ with $\pi(u)$. Likewise, we denote by $C[\pi][w]$ the expression obtained by replacing each clock x in $C[\pi]$ with $w(x)$. If $C[\pi][w]$ evaluates to true, we write $<w, \pi> \models C$. If the set of clock valuations that satisfy $C[\pi]$ is nonempty, i.e., $\exists w : <w, \pi> \models C$, then π *satisfies* C, denoted by $\pi \models C$. Given $C_1, C_2 \in \mathcal{K}_{X \cup U}$, we write $C_1 = C_2$ if $\forall w, \pi : <w, \pi> \models C_1 \Leftrightarrow <w, \pi> \models C_2$.

Similarly to the semantics of constraints over X and U, we say that a parameter valuation π *satisfies* K, denoted by $\pi \models K$, if the expression obtained by replacing in K each $u \in U$ with $\pi(u)$ evaluates to true.

Given C and a set $X' \subseteq X$ of clocks, we denote by $\exists X' : C$ the constraint over X and U obtained from C after elimination[1] of the clocks of X'. Similarly, we denote by $\exists X : C$ the constraint over U obtained from C after elimination

[1] Using variable elimination techniques such as Fourier-Motzkin elimination [33].

of all clocks. We denote by $C_{/X'}$ the constraint over X and U obtained from C after projection onto the clocks of X', i.e., $\exists (X \setminus X') : C$.

We define the *time elapsing* of C, denoted by $C^{\uparrow}$, as the constraint over X and U obtained from C by delaying an arbitrary amount of time, i.e., by renaming X' with X in the following expression: $(\exists X, d : C \wedge X' = X + d)$, where d is a new parameter with values in $\mathbb{R}_{\geq 0}$, and X' is a fresh set of clocks.

Events In the following, τ denotes an unobservable event; $\checkmark$ denotes the special event of process termination; Σ denotes the set of observable events such that $\tau \notin \Sigma$ and $\checkmark \in \Sigma$; $\Sigma_\tau = \Sigma \cup \{\tau\}$. Furthermore, the following event naming conversion is adapted: $e \in \Sigma$ denotes an observable event; $a \in \Sigma_\tau$ denotes an observable event or τ; $E \subseteq \Sigma$ denotes a set of observable events.

Labeled Transition Systems Labeled transition systems will be used later on to represent the semantics of PSTCSP.

Definition 2.1: A *labeled transition system (LTS)* is a tuple $\mathcal{L} = (S, s_0, \Sigma_\tau, \Rightarrow)$ where S is a set of *states*, $s_0 \in S$ is the *initial state*, Σ_τ is a *set of symbols*, and $\Rightarrow : S \times \Sigma_\tau \times S$ is a *labeled transition relation*. We write $s \overset{a}{\Rightarrow} s'$ for $(s, a, s') \in \Rightarrow$. A *run* of $\mathcal{L}$ is an alternating sequence of states $s_i \in S$ and symbols $a_i \in \Sigma_\tau$ of the form $\langle s_0, a_0, s_1, a_1, \cdots \rangle$ such that $s_i \overset{a_i}{\Rightarrow} s_{i+1}$ for all i. A state s_i is *reachable* if it belongs to some run r. We denote by $\mathit{Runs}(\mathcal{L})$ the set of runs of $\mathcal{L}$.

3 Syntax and Semantics of PSTCSP

In this section, we introduce the syntax and the semantics of PSTCSP. Due to the strong similarity between the syntax of STCSP and PSTCSP, we do not recall here the syntax of the former (refer to [35] for details).

3.1 Syntax

A process models the control logic of the system using a rich set of process constructs. A process P is defined by the grammar in Figure 1, where $u \in U$.[2] Processes marked with * allow the use of parameters instead of timing constants in STCSP. $\mathcal{P}$ denotes the set of all possible processes.

Definition 3.1: A Parametric Stateful Timed CSP (or PSTCSP) model is a tuple $\mathsf{M} = (\mathit{Var}, U, V_0, P, K_0)$ where $\mathit{Var} \subset \mathcal{V}ar$, $U \subset \mathcal{U}$, V_0 is the initial variable valuation, $P \in \mathcal{P}$ is a process, and $K_0 \in \mathcal{K}_U$ is an initial constraint.

The initial constraint K_0 allows one to define constrained models, where some parameters are already related. For example, in a timed model with two parameters min and max, one may want to constrain min to be always smaller or equal to max, i.e., $K_0 = \{\mathit{min} \leq \mathit{max}\}$.

[2] Actually, $u \in (U \cup \mathbb{Q}_{\geq 0})$ would be possible too, but having $u \in U$ simplifies the reasoning and proofs.

$P \doteq \texttt{Stop}$	inaction
$\mid \texttt{Skip}$	termination
$\mid e \rightarrow P$	event prefixing
$\mid a\{\mathit{program}\} \rightarrow P$	data operation
$\mid \texttt{if}\ (b)\ \{P\}\ \texttt{else}\ \{Q\}$	conditional choice
$\mid P \vert Q$	general choice
$\mid P \setminus E$	hiding
$\mid P; Q$	sequential composition
$\mid P \parallel Q$	parallel composition
$\mid \texttt{Wait}[u]$	delay*
$\mid P\ \texttt{timeout}[u]\ Q$	timeout*
$\mid P\ \texttt{interrupt}[u]\ Q$	timed interrupt*
$\mid P\ \texttt{within}[u]$	timed responsiveness*
$\mid P\ \texttt{deadline}[u]$	deadline*
$\mid Q$	process referencing

Fig. 1: Syntax of PSTCSP processes

Hierarchy comes from the nested definition of processes. Each component may have internal hierarchies, and allow abstraction and refinement, in the sense that a subprocess may be replaced by another equivalent one in some cases. Also, this offers a readable syntax, starting from the top level of the system, and being more precisely defined when one goes to lower hierarchical levels.

Instantiation of a Model Given a PSTCSP model $\mathsf{M} = (\mathit{Var}, U, V_0, P, K_0)$ and a parameter valuation $\pi = (\pi_1, \dots, \pi_M)$, $\mathsf{M}[\pi]$ denotes the *instantiation* of M with π, viz., the model $(\mathit{Var}, U, V_0, P, K)$, where K is $K_0 \wedge \bigwedge_{i=1}^{M}(u_i = \pi_i)$. This corresponds to the PSTCSP model obtained from M by substituting every occurrence of a parameter u_i by constant π_i in the timed constructs. Note that $\mathsf{M}[\pi]$ is a non-parametric STCSP model.

3.2 Informal Semantics

Untimed constructs We first briefly describe the untimed constructs, which are identical to the ones in STCSP, and very close to the ones of CSP. Process `Stop` does nothing but idling. Process `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, sequential programs may be attached with events. Process $a\{\mathit{program}\} \rightarrow P$ 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 (e.g., $a\{v_1 := 5; v_2 := 3\}$, where $v_1, v_2 \in \mathit{Var}$) or a more complicated sequential program. A conditional choice is written as `if` (b) $\{P\}$ `else` $\{Q\}$. If b is true, then it behaves as P; otherwise it behaves as Q. Process $P|Q$ offers an unconditional choice[3] between P and Q. Process $P; Q$ behaves as P until P terminates and then behaves as Q immediately. $P \setminus E$ hides occurrences of events in E. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization (following CSP rules [22]) or shared variables.

Timed Constructs We now explain the parametric timed constructs.

- Given a parameter u, process `Wait`$[u]$ idles for an unknown (constant) number of u time units.
- In process P `timeout`$[u]$ Q, the *first* observable event of P shall occur before u time units elapse. Otherwise, Q takes over control after exactly u time units.
- Process P `interrupt`$[u]$ Q behaves exactly as P until u time units, and then Q takes over. In contrast to P `timeout`$[u]$ Q, P may engage in *multiple* observable events before it is interrupted. Also note that Q will be executed in any case, whereas in P `timeout`$[u]$ Q, process Q will only be executed if no observable event occurs before u time units.
- Process P `within`$[u]$ must react within an unknown number of u time units, i.e., an observable event must be engaged by process P within u time units.
- Process P `deadline`$[u]$ constrains P to terminate, possibly after engaging in multiple observable events, before u time units.

Discussion on deadline The deadline timed construct intuitively means that a process must terminate within a certain amount of time. Different definitions of deadline actually appear in the literature. In [19], a definition of the deadline command is given, and an instantiation as an extension to the high-integrity SPARK programming language is proposed. In this case, a static analysis is performed during the compiling process and, in the case where an inability to meet the timing constraints occurs, then an appropriate error feedback is sent to the programmer. As a consequence, the deadline construction *guarantees* that the constrained process will terminate before the specified deadline.

In [31], the authors use Unifying Theory of Programming in order to formalize the semantics of Timed Communicating Object Z (TCOZ). As in [19], they consider that the deadline imposes a *timing constraint* on P, which thus requires the computation of P to be finished within the time mentioned in the deadline.

[3] For simplicity, in the discussion, we leave out external and internal choices from the classic CSP [22]. Nevertheless, both constructions are defined in PSTCSP, implemented in PAT, and used in our case studies.

Different from [31, 19], we here choose to stick to the semantics of STCSP [35] and consider a deadline semantics as an *attempt* to terminate a process before a certain time. If the process does not terminate before the deadline, it is just stopped[4].

Syntactic sugar Urgent event prefixing [17], written as $e \twoheadrightarrow P$, is defined as $(e \rightarrow P)$ `within`$[0]$, i.e., e must occur as soon as it is enabled. Furthermore, we sometimes use e for $e \rightarrow$ `Skip` when clear from the context (in particular, in the proof of Theorem 4.5).

Also note that some timed constructs can be defined using other timed constructs. For instance, `within` can be defined using `deadline` (see proof of Proposition4.1 p. 13).

3.3 Example: Fischer Mutual Exclusion

We introduce an example[5] to show that PSTCSP is expressive enough to capture hierarchical concurrent real-time systems.

Fischer's mutual exclusion algorithm is modeled as $(\mathit{Var}, U, v_i, \mathit{FME}, \mathit{True})$, where $U = \{\delta, \gamma\}$, and $\mathit{Var} = \{\mathit{turn}, \mathit{cnt}\}$. The *turn* variable indicates which process attempted to access the critical section most recently. The *cnt* variable counts the number of processes accessing the critical section. Initial valuation v_i maps *turn* to -1 (which denotes that no process is attempting initially) and *cnt* to 0 (which denotes that no process is in the critical section initially). Process *FME* is defined as follows[6].

$$
\begin{aligned}
\mathit{FME} &\doteq \mathit{proc}(1) \parallel \mathit{proc}(2) \parallel \cdots \parallel \mathit{proc}(n)\\
\mathit{proc}(i) &\doteq \texttt{if}\ (\mathit{turn} = -1)\ \{\mathit{Active}(i)\}\ \texttt{else}\ \{\mathit{proc}(i)\}\\
\mathit{Active}(i) &\doteq (\mathit{update}.i\{\mathit{turn} := i\} \rightarrow \texttt{Wait}[\gamma])\ \texttt{within}[\delta];\\
&\quad \texttt{if}\ (\mathit{turn} = i)\\
&\quad\quad \mathit{cs}.i\{\mathit{cnt} := \mathit{cnt} + 1\} \rightarrow\\
&\quad\quad \mathit{exit}.i\{\mathit{cnt} := \mathit{cnt} - 1; \mathit{turn} := -1\} \rightarrow \mathit{proc}(i)\\
&\quad \texttt{else}\\
&\quad\quad \mathit{proc}(i)
\end{aligned}
$$

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

[4] Remark that, in that case, time elapsing may be stopped too.
[5] This example is a parametrization of Example 1 from [35].
[6] Note that this is not the real ASCII-based PAT syntax.

A classical parameter synthesis problem is to find values of δ and γ for which mutual exclusion is guaranteed. One way to verify mutual exclusion is to show that $cnt \leq 1$ is always true. A solution to this problem will be given in Section 5.4.2. □

3.4 Clock Activation

The semantics uses parameters and clocks. Like in STCSP, clocks in PSTCSP are *implicitly* associated with timed processes – which is different from PTAs. For instance, given a process P `timeout`$[u]$ Q, an implicit clock should start whenever this process is activated. A clock starts ticking once the process becomes activated. Before defining the semantics, we need to associate clocks with time processes explicitly. In theory, each timed process construct is associated with a unique clock. Nonetheless, as in STCSP, multiple timed processes can be activated at the same time during system execution and, therefore, the associated clocks always have the same value. Consider the following process:

$$P \doteq (\texttt{Wait}[u_1]; \texttt{Wait}[u_2]) \ \texttt{interrupt}[u_3] \ Q.$$

There are three implicit clocks, one associated with `Wait`$[u_1]$ (say x_1), one with `Wait`$[u_2]$ (say x_2) and one with P (because of `interrupt`$[u_3]$, say x_3). Clocks x_1 and x_3 are starting at the same time because the execution of `interrupt` is linked with `Wait`$[u_1]$. In contrast, clock x_2 starts only when `Wait`$[u_1]$ terminates. It can be shown that x_1 and x_3 always have the same value and thus one clock is sufficient. In order to minimize the number of clocks, we introduce clocks at runtime so that timed processes which are activated at the same time share the same clock. Intuitively, a clock is introduced if and only if one or more timed processes have just become activated.

We recall from [35] how to systematically associate clocks with timed processes. To distinguish from ordinary PSTCSP processes, let $\mathcal{P}_{act}$ denote the set of processes associated with explicit clocks. We write `Wait`$[u]_x$ (and, similarly, P `timeout`$[u]_x$ Q, P `interrupt`$[u]_x$ Q, P `within`$[u]_x$, P `deadline`$[u]_x$) to denote that the process is associated with clock x. Given a process P and a clock x, we use function $Act(P, x)$ to define the corresponding process in $\mathcal{P}_{act}$.

Figure 2 presents the detailed definition. Rules A1 to A5 state that if a process is untimed and none of its subprocesses are activated, then it is unchanged. Rules A6 to A10 state that if the process is timed, then it is associated with x. Note that if a timed process has already been associated with a clock x', then it will not be associated with the new clock. This is captured by rules A11 to A15, where `Wait`$[u]_{x'}$ denotes that `Wait`$[u]$ is associated with clock x'. If a subprocess is activated, then function Act is applied recursively, as captured by rules A7 to A10 and A12 to A19. Rule A20 states that if P is defined as Q, then $Act(P, x)$ can be obtained by applying Act to Q.

We denote by $cl(P)$ the set of *active clocks* associated with P or any subprocess of P. For instance, the set of clocks associated with P `timeout`$[u]_x$ Q contains x and the clocks associated with P. Notice that there is no clock associated with Q because it is not activated yet.

$Act(\texttt{Stop}, x)$	$= \texttt{Stop}$	A1
$Act(\texttt{Skip}, x)$	$= \texttt{Skip}$	A2
$Act(e \rightarrow P, x)$	$= e \rightarrow P$	A3
$Act(a\{program\} \rightarrow P, x)$	$= a\{program\} \rightarrow P$	A4
$Act(\texttt{if } (b)\ \{P\} \texttt{ else } \{Q\}, x)$	$= \texttt{if } (b)\ \{P\} \texttt{ else } \{Q\}$	A5
$Act(\texttt{Wait}[u], x)$	$= \texttt{Wait}[u]_x$	A6
$Act(P \texttt{ timeout}[u]\ Q, x)$	$= Act(P, x) \texttt{ timeout}[u]_x\ Q$	A7
$Act(P \texttt{ interrupt}[u]\ Q, x)$	$= Act(P, x) \texttt{ interrupt}[u]_x\ Q$	A8
$Act(P \texttt{ within}[u], x)$	$= Act(P, x) \texttt{ within}[u]_x$	A9
$Act(P \texttt{ deadline}[u], x)$	$= Act(P, x) \texttt{ deadline}[u]_x$	A10
$Act(\texttt{Wait}[u]_{x'}, x)$	$= \texttt{Wait}[u]_{x'}$	A11
$Act(P \texttt{ timeout}[u]_{x'}\ Q, x)$	$= Act(P, x) \texttt{ timeout}[u]_{x'}\ Q$	A12
$Act(P \texttt{ interrupt}[u]_{x'}\ Q, x)$	$= Act(P, x) \texttt{ interrupt}[u]_{x'}\ Q$	A13
$Act(P \texttt{ within}[u]_{x'}, x)$	$= Act(P, x) \texttt{ within}[u]_{x'}$	A14
$Act(P \texttt{ deadline}[u]_{x'}, x)$	$= Act(P, x) \texttt{ deadline}[u]_{x'}$	A15
$Act(P \vert Q, x)$	$= Act(P, x) \vert Act(Q, x)$	A16
$Act(P \setminus E, x)$	$= Act(P, x) \setminus E$	A17
$Act(P; Q, x)$	$= Act(P, x); Q$	A18
$Act(P \parallel Q, x)$	$= Act(P, x) \parallel Act(Q, x)$	A19
$Act(P, x)$	$= Act(Q, x) \quad \text{if } P \doteq Q$	A20

Fig. 2: Clock activation function

In the Fischer's mutual exclusion example, assume there are three processes. The first and second processes have evaluated the condition $\texttt{if}\ (turn = -1)$ and become $Active(0)$ and $Active(1)$, whereas the third process has not made any move. So the current process is $Active(1) \parallel Active(2) \parallel proc(3)$. Assume that x is a fresh clock. Then applying function Act with x returns

$$\begin{array}{l}(update.1\{turn := 1\} \rightarrow \texttt{Wait}[\gamma]) \texttt{ within}_x[\delta]; \cdots \\ \parallel (update.2\{turn := 2\} \rightarrow \texttt{Wait}[\gamma]) \texttt{ within}_x[\delta]; \cdots \\ \parallel \texttt{if } (turn = -1)\ \{\ Active(3)\ \} \texttt{ else } \{\ proc(3)\ \}\end{array}$$

Clock x is associated with the first process and the second process, but not the third process. Note that $\texttt{Wait}[\gamma]$ has not been activated yet. □

3.5 Semantics

In STCSP, a state is a triple (V, P, D), where V is variable valuation, $P \in \mathcal{P}$ and $D \in \mathcal{K}_X$. This abstraction of the timing constants by dynamic zone abstraction implies a finite number of states. This is implemented in [35] using Difference Bound Matrices (DBMs), following works for TAs [18, 9, 11].

In the following, we introduce the semantics for PSTCSP in terms of states containing constraints over X and U. We define below the notion of state.

Definition 3.2: Let $\mathsf{M} = (Var, U, V_0, P, K_0)$ be a PSTCSP model. Then a *(symbolic) state* s of M is a triple (V, P, C) where V is a variable valuation, $P \in \mathcal{P}$

is a process, and $C \in \mathcal{K}_{X \cup U}$.

The semantics of a PSTCSP model can then be understood intuitively as the union of the semantics of the instantiated non-parametric STCSP models, for all possible parameter valuations. For each parameter valuation π, we may view a state $s = (V, P, C)$ as the set of triples (V, P, w) where w is a clock valuation such that $<w, \pi> \models C$.

3.5.1 Idling Function

We adapt in the following the function *idle*, defined in [35], which, given a process in $\mathcal{P}_{act}$, calculates a constraint expressing how long the process can idle. The result is in the form of a constraint over the clocks and the parameters. Figure 3 shows the detailed definition. Rules *idle*1 to *idle*5 state that if the process is untimed and none of its subprocesses is activated, then the function returns true. Intuitively, it means that the process may idle for arbitrary amount of time. Rules *idle*6 to *idle*9 state that if subprocesses of the process are activated, then function *idle* is applied to the subprocesses. For instance, if the process is a choice (rule *idle*6) or a parallel composition (rule *idle*9) of P and Q, then the result is $idle(P) \wedge idle(Q)$. Intuitively, this means that process $P|Q$ (or $P \parallel Q$) may idle as long as both P and Q can idle. Rules *idle*10 to *idle*14 define the cases when the process is timed. For instance, process $\texttt{Wait}[u]_x$ may idle as long as x is less than or equal to u.

$$\begin{array}{lll}
idle(\texttt{Stop}) & = \mathit{True} & idle1 \\
idle(\texttt{Skip}) & = \mathit{True} & idle2 \\
idle(e \rightarrow P) & = \mathit{True} & idle3 \\
idle(a\{program\} \rightarrow P) & = \mathit{True} & idle4 \\
idle(\texttt{if}\ (b)\ \{P\}\ \texttt{else}\ \{Q\}) & = \mathit{True} & idle5 \\
idle(P|Q) & = idle(P) \wedge idle(Q) & idle6 \\
idle(P \setminus E) & = idle(P) & idle7 \\
idle(P; Q) & = idle(P) & idle8 \\
idle(P \parallel Q) & = idle(P) \wedge idle(Q) & idle9 \\
idle(\texttt{Wait}[u]_x) & = x \leq u & idle10 \\
idle(P\ \texttt{timeout}[u]_x\ Q) & = x \leq u \wedge idle(P) & idle11 \\
idle(P\ \texttt{interrupt}[u]_x\ Q) & = x \leq u \wedge idle(P) & idle12 \\
idle(P\ \texttt{within}[u]_x) & = x \leq u \wedge idle(P) & idle13 \\
idle(P\ \texttt{deadline}[u]_x) & = x \leq u \wedge idle(P) & idle14 \\
idle(P) & = idle(Q) \quad \text{if } P \doteq Q & idle15
\end{array}$$

Fig. 3: Idling calculation

3.5.2 Semantics

We now define the semantics of PSTCSP under the form of an LTS. Let $Y = \langle x_0, x_1, \cdots \rangle$ be a sequence of clocks.

Definition 3.3: Let $\mathsf{M} = (Var, U, V_0, P, K_0)$ be a PSTCSP model. The *semantics of* M, denoted by $\mathcal{L}_{\mathsf{M}}$, is an LTS $(S, s_0, \Rightarrow, \Sigma_\tau)$ where

$$\begin{aligned} S &= \{(V, P, C) \in \mathcal{V}(Var) \times \mathcal{P} \times \mathcal{K}_{X \cup U}\}, \\ s_0 &= (V_0, P, K_0) \end{aligned}$$

and the transition relation $\Rightarrow$ is the smallest transition relation satisfying the following. For all $(V, P, C) \in S$, if x is the first clock in the sequence Y which is not in $cl(P)$, and $(V, Act(P, x), C \wedge x = 0) \overset{a}{\rightsquigarrow} (V', P', C')$ then we have: $((V, P, C), a, (V', P', C'_{/cl(P')})) \in \Rightarrow$.

The transition relation $\rightsquigarrow$ is specified by a set of rules, given in Appendix A. We explain below some of the rules defining the transition relation $\rightsquigarrow$. Other rules can be explained similarly, following the way of [35].

- Rule *await* defines the semantics of $\texttt{Wait}[u]$.

$$\frac{}{(V, \texttt{Wait}[u]_x, C) \overset{\tau}{\rightsquigarrow} (V, \texttt{Skip}, C^{\uparrow} \wedge x = u)} \quad (await)$$

  It states that a τ-transition occurs exactly when clock x is equal to u. Intuitively, $C^{\uparrow} \wedge x = u$ denotes the time when u time units elapsed since x has started. Afterwards, the process becomes $\texttt{Skip}$.

- Rules *ato*1, *ato*2 and *ato*3 define the semantics of $P \ \texttt{timeout}[u] \ Q$. Rule *ato*1 states that if a τ-transition transforms (V, P, C) to (V', P', C'), then a τ-transition may occur given $(V, P \ \texttt{timeout}[u]_x \ Q, C)$ if constraint $C' \wedge x \leq u$ is satisfiable. Intuitively, this means that the τ-transition must occur before u time units since x has started.

$$\frac{(V, P, C) \overset{\tau}{\rightsquigarrow} (V', P', C')}{(V, P \ \texttt{timeout}[u]_x \ Q, C) \overset{\tau}{\rightsquigarrow} (V', P' \ \texttt{timeout}[u]_x \ Q, C' \wedge x \leq u)} \quad (ato1)$$

  Similarly, rule *ato*2 ensures that the occurrence of an observable event e from process P may occur only if $x \leq u$, i.e., before timeout occurs.

$$\frac{(V, P, C) \overset{e}{\rightsquigarrow} (V', P', C')}{(V, P \ \texttt{timeout}[u]_x \ Q, C) \overset{e}{\rightsquigarrow} (V', P', C' \wedge x \leq u)} \quad (ato2)$$

  Rule *ato*3 states that timeout results in a τ-transition when the reading of x is exactly u. The constraint $x = u \wedge idle(P)$ ensures that process P may idle all the way until timeout occurs.

$$\frac{}{(V, P \ \texttt{timeout}[u]_x \ Q, C) \overset{\tau}{\rightsquigarrow} (V, Q, C^{\uparrow} \wedge x = u \wedge idle(P))} \quad (ato3)$$

Let us explain further Definition 3.3. Given a state (V, P, C), a clock x which is not currently associated with P is picked. The state (V, P, C) is transformed into $(V, Act(P, x), C \wedge x = 0)$, i.e., timed processes which just become activated are associated with x and C is conjuncted with $x = 0$. Then, a firing rule is applied to get a target state (V', P', C') such that C' be satisfiable (otherwise, the transition is infeasible). Lastly, clocks which are not in $cl(P')$ are pruned from C'. Observe that one clock may be introduced and zero or more clocks may be pruned during a transition.

Consider some state $s_1 = (V, \texttt{Wait}[u_1]\texttt{interrupt}[u_2]\texttt{Skip}, u_2 < u_1)$. Activation with x_1 gives $(V, \texttt{Wait}[u_1]_{x_1}\texttt{interrupt}[u_2]_{x_1}\texttt{Skip}, u_2 < u_1 \wedge x_1 = 0)$. Applying firing rule *ait2* gives state $(V, \texttt{Skip}, C)$ with $C = \{(u_2 < u_1 \wedge x_1 = 0)^{\uparrow} \wedge x_1 = u_2 \wedge idle(\texttt{Wait}[u_1]_{x_1})\}$, viz., $u_2 < u_1 \wedge x_1 \geq 0 \wedge x_1 = u_2 \wedge x_1 \leq u_1$. Then, we remove x_1 from C because it does not appear within `Skip`; this gives new state $s_2 = (V, \texttt{Skip}, u_2 < u_1)$.

We can also apply firing rule *ait1* (and hence *await*) to s_1, which gives $(V, \texttt{Skip interrupt}[u_2]_{x_1}, C')$ with $C' = u_2 < u_1 \wedge x_1 = u_1 \wedge x_1 \leq u_2$. This constraint is unsatisfiable, hence this state is discarded. □

4 Expressiveness and Undecidability

4.1 Expressiveness

We first state that STCSP is equivalent to Closed Timed ϵ-Automata [29], i.e., timed safety automata [20] with ϵ-transitions [1, 12] and exclusively closed guards and invariants (i.e., whose inequalities are of the form $e \leq e'$, with e, e' linear terms). It is usually considered that this restriction is benign in practice, due to the fact that any timed automaton can be infinitesimally approximated by one with closed constraints [28, 29, 8].

Lemma 4.1: Stateful Timed CSP is as expressive as Closed Timed ϵ-Automata.

Proof We first show that STCSP without the `deadline` and the `within` constructs is equivalent to Timed CSP. It is known that all Timed CSP constructs, including `timeout` and `interrupt` can be derived from `Wait`$[d]$ and CSP constructs [17]. It has been shown that the expressive power of Timed CSP is equal to Closed Timed ϵ-Automata [29]. As a consequence, STCSP without the `deadline` and the `within` constructs is equivalent to Timed CSP.

Furthermore, the `within` construct can be defined using the `deadline` construct: considering P `within`$[d]$, this can be achieved by executing P in parallel with Q `deadline`$[d]; R$, with Q a process synchronizing on any observable event with P, and R a process synchronizing, possibly several times, on any observable event with P. Finally, the `deadline`$[d]$ construct can be easily translated into a Closed Timed ϵ-Automata by adding a location with an invariant $x \leq d$, for some additional clock x set to 0 when the process `deadline`$[d]$ is activated. Which gives the result. ■

We define Parametric Closed Timed ϵ-Automata as a parametric extension of Closed Timed ϵ-Automata, following the parameterization of TAs into PTAs [3], i.e., by using within guards and invariants parameters (unknown constants). It follows from Lemma 4.1 that PSTCSP is equivalent to Parametric Closed Timed ϵ-Automata.

Proposition 4.2: Parametric Stateful Timed CSP is as expressive as Parametric Closed Timed ϵ-Automata.

Since Closed Timed ϵ-Automata are a subclass of ϵ-TAs [4], then Parametric Closed Timed ϵ-Automata are a subclass of ϵ-PTAs. By corollary of Proposition 4.2, PSTCSP is less expressive than ϵ-PTAs, but incomparable with standard PTAs.

We believe that PSTCSP is an interesting formalism because one can make use of complex data structures and the τ-transitions are used in PSTCSP for compositionality of the sub-component, which is missing in PTAs. Furthermore, high level real-time system requirements often state the system timing constraints in terms of deadline, timeout or wait, which can be regarded as common timing patterns. For example, "task P must complete within u units of time" is a typical one (`deadline`$[u]$). PSTCSP is better suited for specifying the requirements of complex real-time systems because it has the exact language constructs that can directly capture those common timing patterns. On the other hand, if PTAs are considered to be used to capture high level real-time requirements, then one often needs to manually cast those timing patterns into a set of clock variables explicitly and carefully design the constraints. Also, although tools exist for specifying hierarchy or some data structures for (non-parametric) TAs, such as UPPAAL, PSTCSP is, as far as we know, the first fully parametric formalism allowing to combine hierarchical aspects, shared variables and complex data structures in a single and readable formalism.

4.2 Membership and Emptiness

We consider here the questions of membership ("is a parameter valuation consistent with a model?") and emptiness ("given a model M, does there exist a parameter valuation consistent with M?"). Both questions refer to the notion of *consistency*. For PTAs, consistency is defined as the acceptance of at least one timed word. This notion of acceptance of words relies on the existence of accepting locations: a timed word is accepted by a PTA A if A ends up in an accepting location after reading it. However, CSP (and its timed, parametric extensions) does not feature the notion of "accepting" processes. We consider instead the reachability problem: can the initial state of the model reach another state through some run? Or, equivalently, does an execution starting from a process P_0 lead to a given process P?

Definition 4.3 (Consistency): Given a PSTCSP model M, given a process P, a parameter valuation π is *consistent* with π if there exists a run such that the initial state (V_0, P_0, C_0) of M derives to a state (V, P, C), for some V and C.

Formally, given a PSTCSP model M of initial state (V_0, P_0, C_0), given $P \in \mathcal{P}$, we denote by $\Pi(\mathsf{M})$ the set of parameter valuations consistent with M, i.e., $\{\pi \in U \mid \exists V, C : (V_0, P_0, C_0) \rightsquigarrow (V, P, C) \in \mathit{Runs}(\mathsf{M}[\pi])\}$.

Membership The membership question consists of deciding whether a given parameter valuation π is consistent with a PSTCSP model M. The membership problem is decidable for PSTCSP: it suffices to consider the non-parametric STCSP model $\mathsf{M}[\pi]$ and solve this problem using techniques developed in [35], e.g., by building the set of all reachable states.

Proposition 4.4 (Decidability of membership): Let M a PSTCSP model, and π a parameter valuation. The problem of deciding if π is consistent with $\Pi(\mathsf{M})$ is decidable.

Emptiness We now show that the emptiness problem for PSTCSP is undecidable.

Theorem 4.5 (Undecidability of emptiness): Let M be a PSTCSP model, and P a process. The problem of deciding if $\Pi(\mathsf{M})$ is empty is undecidable.

Proof We reduce the halting problem for 2-counter machines to the problem of testing if there exists a parameter valuation consistent with a PSTCSP model. We follow the reduction used in [3], and adapt it to PSTCSP.

As in [3], we consider a 2-counter machine CM with two counters C_1 and C_2. The control variable l of CM ranges over the set $\{l_1, \dots, l_n\}$. Each instruction of CM can either increment or decrement one of the counters, or test if one of the counters is equal to 0, and change the location of control. A configuration of CM is a triple (l_i, C_1, C_2), specifying the values of l, C_1 and C_2, respectively. The initial configuration of CM is $(l_0, 0, 0)$. The halting problem consists of deciding if CM can reach a given configuration (l_i, C_1, C_2). We construct in the following a PSTCSP model M_{CM} such that $\Pi(\mathsf{M}_{CM})$ is nonempty iff M_{CM} halts. In order to simplify the proof, we consider that no instruction corresponds to control variable l_n (i.e., if the machine reaches l_n, it will halt).

We set $\mathsf{M}_{CM} = (\mathit{Var}, U, V_0, P_{CM}, K_0)$, with

- $\mathit{Var} = \emptyset$;
- $U = \{a, a_{-1}, a_{+1}, b, b_{-1}, b_{+1}\}$;
- $V_0 = \emptyset$;
- $K_0 = \mathit{True}$; and
- P_{CM} is explained in the following.

Recall that in [3], a configuration is encoded using the triple $(l_i, b - y, b - a - z)$, where y and z are two of the three clocks used in the construction. Our encoding will be relatively similar, with the exception that clocks are implicit.

Also recall that an instruction of the form “if $l = l_i$ then $C_1 := C_1 + 1$ and $l := l_{i'}$” is modeled in [3] by adding a path between some appropriate states of the PTA modeling CM, using the scheme recalled in Figure 4.

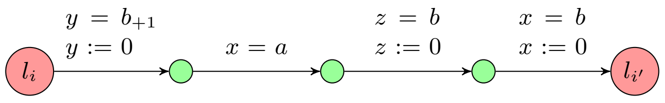

Fig. 4: Undecidability proof of [3]

The main difficulty when adapting the proof of [3] to PSTCSP is the fact that clocks are now implicit. As a consequence, it is more difficult to constrain the parameters than in a PTA. We will use 3 processes X, Y and Z running in parallel in order to model the 3 clocks used in [3], plus an additional process W in order to synchronize on events.

For each control variable l_i of CM, consider the set of instructions starting in this control variable (i.e., of the form “if $l = l_i$ then ...”). For each instruction I_{ij}, i.e., the jth instruction starting in the control variable i, we will define 4 processes.

Consider an instruction of the form “if $l = l_i$ then $C_1 := C_1 + 1$ and $l := l_{i'}$”. The 4 processes defined for this instruction are as follows:

$$\begin{aligned}
X_{ij} &\doteq \texttt{Wait}[b-a]; e^4_{ij} \twoheadrightarrow (e^1_{ij}\texttt{within}[a]); e^2_{ij} \twoheadrightarrow X_{i'}\\
Y_{ij} &\doteq \texttt{Wait}[b_{+1}]; e^1_{ij} \twoheadrightarrow Y_{i'}\\
Z_{ij} &\doteq \texttt{Wait}[b]; e^3_{ij} \twoheadrightarrow Z_{i'}\\
W_{ij} &\doteq e^4_{ij}; e^1_{ij}; e^2_{ij}; e^3_{ij}; W_{i'}
\end{aligned}$$

The three processes X, Y, Z correspond to the three clocks x, y, z, respectively, of Figure 4. They synchronize on a set of events, and the order between the events, which is crucial in order to constrain the parameters, is achieved by process W. We name those events e^1_{ij} to e^4_{ij}, where e^k_{ij} corresponds to the kth transition of the construction recalled in Figure 4. Figure 5 gives the idea of the construction, mentioning in particular the duration between any two events. The `within` construct in process X is used in order to let event e^1_{ij} occur anytime between e^4_{ij} and e^2_{ij}. However, for an instruction of the form “if $l = l_i$ and $C_1 = 0$ then $l := l_{i'}$” (see below), it will be constrained to happen immediately after e^4_{ij}.

For an instruction of the form “if $l = l_i$ then $C_1 := C_1 - 1$ and $l := l_{i'}$”, we define the four processes in the same way, except Y_{ij} where $\texttt{Wait}[b_{+1}]$ should be replaced with $\texttt{Wait}[b_{-1}]$.

For an instruction of the form “if $l = l_i$ and $C_1 = 0$ then $l := l_{i'}$”, we define the four processes in the same way, except Y_{ij} where $\texttt{Wait}[b_{+1}]$ should be replaced with $\texttt{Wait}[b]$, and X_{ij} is defined as follows:

$$X_{ij} \doteq \texttt{Wait}[b-a]; e^4_{ij} \twoheadrightarrow e^1_{ij} \twoheadrightarrow \texttt{Wait}[a]; X_{i'}$$

We also define four sets of processes, for $i = 1, \ldots, n-1$, as follows:

$$X_i \doteq \bigcup X_{ij} \quad , \quad Y_i \doteq \bigcup Y_{ij} \quad , \quad Z_i \doteq \bigcup Z_{ij} \quad , \quad W_i \doteq \bigcup W_{ij}$$

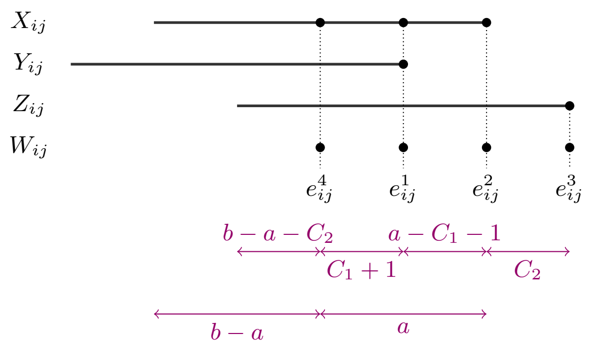

Fig. 5: Proof of undecidability: synchronization between processes

where $\bigcup X_{ij}$ denotes a general choice between the m processes starting in control variable i, i.e., $X_{i1} \mid \cdots \mid X_{im}$.

The final processes Y_n, Z_n and W_n are all equal to each other and defined as $Y_n \doteq e_n \to \texttt{Skip}$ (and similarly for Z_n and W_n). As for X_n, we define it as $X_n \doteq e_n \to P$. This gives the final synchronization allowing the global process to derive to P.

As in [3], we must also ensure that we initially have the following relationship between the parameters:

$$a = a_{+1} - 1 = a_{-1} + 1 \quad \wedge \quad b = b_{+1} - 1 = b_{-1} + 1$$

This can be easily ensured by the following initialization process:

$$\begin{aligned} P_0 \doteq \quad & \texttt{Wait}[a]; e^0 \twoheadrightarrow \texttt{Wait}[b]; e^0 \twoheadrightarrow \texttt{Skip} \\ \| \; & \texttt{Wait}[a_{-1}+1]; e^0 \twoheadrightarrow \texttt{Wait}[b_{-1}+1]; e^0 \twoheadrightarrow \texttt{Skip} \\ \| \; & \texttt{Wait}[a_{+1}-1]; e^0 \twoheadrightarrow \texttt{Wait}[b_{+1}-1]; e^0 \twoheadrightarrow \texttt{Skip} \end{aligned}$$

Or, alternatively, we can simply set the constraint K_0 to the desired constraint, instead of *True*.

The global process encoding our construction scheme is given by:

$$P_{CM} \doteq P_0; ((\texttt{Skip}; X_1) \parallel Y_1 \parallel (\texttt{Skip}; Z_1) \parallel (\texttt{Skip}; W_1))$$

The `Skip` construction prefixing each process but Y_1 allows these processes to idle for some time before starting, as the four processes are dephased (see Figure 5).

Then, as in [3], if CM does not halt, then there is no way to reduce P_{CM} to P, and $\Pi(\mathsf{M}) = \emptyset$. If CM does halt, and suppose the value of C_1 (resp. C_2) never exceeds c_1 (resp. c_2), then for a parameter valuation π, $\pi \in \Pi(\mathsf{M})$ iff $a = a_{+1} - 1 = a_{-1} + 1$, and $b = b_{+1} - 1 = b_{-1} + 1$, and $a \geq c_1$ and $b - a \geq c_2$. ■

Alternative Proof of Undecidability Actually, the results of expressiveness given in Section 4.1 gives another way to prove undecidability. Indeed, the proof of undecidability of the emptiness problems for PTAs relies on the reduction of the halting problem for 2-counter machines to the problem of testing if there exists a consistent parameter valuation [3]. The construction uses a translation from 2-counter machines to a PTA using 3 clocks. This PTA actually belongs to the class of parametric closed timed automata, itself a subclass of parametric closed timed ϵ-automata, which has been shown in Section 4.1 to be equivalent to PSTCSP.

An immediate corollary of Theorem 4.5 is that parameter synthesis is undecidable in general.

5 Parameter Synthesis

5.1 An Example of PSTCSP Model

We present here an example of PSTCSP model[7] that will be used to show the application of our algorithms introduced in this section.

$$\mathsf{M}_{ex}^{np} = \{\emptyset, \emptyset, P^{np}, \mathit{True}\}$$

This model is actually non-parametric (np stands for non-parametric), has no variables, and process P^{np} is defined as follows.

$$P^{np} \doteq (a \rightarrow \texttt{Wait}[2]; b \rightarrow \texttt{Stop}) \ \texttt{interrupt}[1] \ c \rightarrow P_{np}$$

Intuitively, event b never occurs because `interrupt` occurs before `Wait`[2] can be achieved. We give in Figure 6 the set of reachable states given under the form of an LTS.

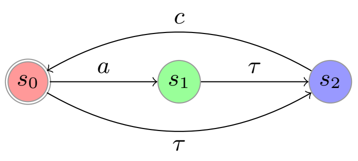

Fig. 6: Reachable states of process P_{np}

Now consider the following parametrized version of M_{ex}^{np}.

$$\mathsf{M}_{ex} = \{\emptyset, \{u_1, u_2\}, P, \mathit{True}\}$$

Process P, still containing no variable, is defined as follows.

$$P \doteq (a \rightarrow \texttt{Wait}[u_2]; b \rightarrow \texttt{Stop}) \ \texttt{interrupt}[u_1] \ c \rightarrow P$$

[7] This example is inspired by Example 2 from [35]

5.2 State Space Exploration

We first define a semi-algorithm to explore the state space until a fixpoint reached, i.e., until no new state can be computed, or all new states have been encountered before. Recall from Definition 2.1 that a state s is reachable in one step from another state s' if s is the successor of s' in a run. This definition extends to sets of states: Given a PSTCSP model M, one defines $Post_{\mathsf{M}}(S)$ (resp. $Post^i_{\mathsf{M}}(S)$) as the set of states reachable from a set S of states in one step (resp. i steps). Formally, $Post_{\mathsf{M}}(S) = \{s' | \exists s \in S, \exists a \in \Sigma_\tau : s \overset{a}{\Rightarrow} s'\}$. And $Post^*_{\mathsf{M}}(S)$ is defined as the set of all states reachable from S in M (i.e., $Post^*_{\mathsf{M}}(S) = \bigcup_{i \geq 0} Post^i_{\mathsf{M}}(S)$). We give in Algorithm 1 a semi-algorithm for computing the set of all reachable states. The inclusion test (used in $Post_{\mathsf{M}}(S) \subseteq S$) denotes the classical set inclusion, i.e.: $S \subseteq S'$ iff $\forall s \in S, \exists s' \in S' : s' = s$. Note that this algorithm does not strictly speaking return the LTS, because transitions are not stored. It would be straightforward to modify this algorithm so that it outputs the whole LTS, including transitions.

```
Algorithm 1: Algorithm reachAll(M)
input  : A PSTCSP model M of initial state $s_0$
output: Set of reachable states
1 $S \leftarrow \{s_0\}$
2 while True do
3     if $Post_{\mathsf{M}}(S) \subseteq S$ then return $S$
4     $S \leftarrow S \cup Post_{\mathsf{M}}(S)$
```

Application to the Example Let us apply *reachAll* to M_{ex}. Since we have no variable, we denote for the sake of conciseness the states by the pair (P, C), where P is the current process, and C the current constraint over X and U. The initial state is $s_0 = (P, \mathit{True})$. Let $\langle x_1, x_2, \cdots \rangle$ be a sequence of clocks.

Starting with s_0, we pick the first unused clock (x_1) and apply Act to P with x_1 to get:

$$s'_0 = (a \to \texttt{Wait}[u_2]; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \quad , \quad x_1 = 0$$

Next, we can apply either rule $ait1$ or $ait2$. Apply rule $ait1$ (with $ase1$, aev), we get:

$$s_1 = (\texttt{Wait}[u_2]; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \quad , \quad 0 \leq x_1 \leq u_1$$

Apply rule $ait2$ to s_0, we get $s_2 = (c \to P, x_1 \geq 0 \land x_1 = u_1)$. Note that clock x_1 becomes irrelevant after the transition. After pruning x_1, we get $s'_2 = (c \to P, \mathit{True})$.

Now consider state s_1. We pick the first unused clock (x_2) and apply Act with x_2 to get:

$$s'_1 = (\texttt{Wait}[u_2]_{x_2}; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \quad , \quad 0 \leq x_1 \leq u_1 \land x_2 = 0$$

One can first apply rule $ait1$ (with $ase1$, $await$) to s'_1, and get (after pruning of x_2):

$$s_3 = (\texttt{Skip}; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \quad , \quad u_2 \leq x_1 \leq u_1$$

One can also apply rule $ait2$ (and $idle8$, $idle10$) to s'_1, and get:

$$c \to P \quad , \quad 0 \leq x_1 - x_2 \leq u_1 \wedge x_1 = u_1 \wedge x_2 \leq u_2$$

After pruning of both x_1 and x_2, we get ($c \to P$, $True$), which is equal to s'_2.

One can apply rule aev to s'_2 to get (P, $True$), which is equal to s_0.

Now consider state s_3. One can first apply rule $ait1$ (with $ase2$, aki) to get:

$$s_4 = (b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \quad , \quad u_2 \leq x_1 \leq u_1$$

One can also apply rule $ait2$ (with $idle8$, $idle2$) to s_3 to get:

$$s_5 = c \to P \quad , \quad u_2 \leq x_1 \wedge x_1 = u_1$$

Which gives after pruning of x_1:

$$s'_5 = c \to P \quad , \quad u_2 \leq u_1$$

Note that s'_5 is not equal to s_2, because the associated constraint is different.

Now consider state s_4. One can first apply rule $ait1$ (with aev) to get:

$$s_6 = \texttt{Stop}\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \quad , \quad u_2 \leq x_1 \leq u_1$$

One can also apply rule $ait2$ (with $idle8$, $idle2$) to s_4, which gives s_5.

From s_6, one can only apply rule $ait2$ (with $idle1$), which also gives s_5.

From state s'_5, one can apply rule aev and get:

$$s_7 = P \quad , \quad u_2 \leq u_1$$

which is almost equivalent to s'_0 after application of Act with x_1, but with the addition of the constraint $u_2 \leq u_1$.

From s_7, one can apply rule $ait1$ (with $ase1$, aev) and get, after application of Act with x_2:

$$s_8 = (\texttt{Wait}[u_2]_{x_2}; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P \ , \ 0 \leq x_1 \leq u_1 \wedge x_2 = 0 \wedge u_2 \leq u_1$$

From s_7, one can also apply rule $ait2$ (with $idle8$, $idle3$), which gives s_5.

Then, from s_8, one can either apply $ait1$ (with $ase1$, $await$), which gives s_4, or apply $ait2$ (with $idle8$, $idle10$), which gives s_5.

We finally reach the fixpoint, and $reachAll$ terminates. The set of reachable states is now stable, and is depicted in Figure 7 under the form of an LTS, viz., a directed graph whose edges are labeled with actions.

The interpretation of the graph is as follows: the projection onto U of the constraint associated with states s'_0, s'_1 and s'_2 is $True$. Hence, these states can be reached for any valuation of u_1 and u_2. However, the projection onto U of the constraint associated with the other states is $u_2 \leq u_1$. Hence, these states can only be reached for parameter valuations satisfying this inequality. Observe that the non-parametric model P^{np} can only reach states (equivalent to) s'_0, s'_1 and s'_2. Indeed, we have that $\mathsf{M}^{np}_{ex} = \mathsf{M}_{ex}[\pi]$, with π is such that $u_1 = 3$ and $u_2 = 5$, hence $u_1 < u_2$. □

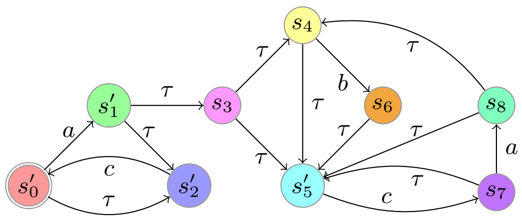

Fig. 7: States reachable in model M_{ex}

Non-termination We show below that *reachAll* does not terminate in the general case.

Proposition 5.1 (Non-termination): Let M be a PSTCSP model. Then Algorithm *reachAll*(M) does not terminate in the general case.

Proof See counterexample in Example 5.2. ∎

We introduce here an example of PSTCSP model for which Algorithm *reachAll* does not terminate. Recall that e stands for $e \rightarrow \texttt{Skip}$.

Consider the PSTCSP model $\mathsf{M} = (\emptyset, \{u_1, u_2\}, \emptyset, P, \mathit{True})$ where P is defined as follows:

$$\begin{aligned} P &\doteq Q \ \texttt{interrupt}[u_1] \ b \\ Q &\doteq a \rightarrow \texttt{Wait}[u_2]; Q \end{aligned}$$

In the following, for the sake of readability, we present states (V, P, C) under the form (P, C), considering the set of variables is empty. The initial state is the following:

$$Q \ \texttt{interrupt}[u_1] \ b \quad , \quad \mathit{True}$$

Instead of applying the whole *Post* operation, which would be time consuming, we only compute a single state successor at a time, and show that we can find an infinite (non-converging) run.

Let us first apply *Act* with fresh clock x_1 (rules A8, A18, A3):

$$Q \ \texttt{interrupt}[u_1]_{x_1} \ b \quad , \quad x_1 = 0$$

By expanding Q, we get:

$$(a \rightarrow \texttt{Wait}[u_2]; Q) \ \texttt{interrupt}[u_1]_{x_1} \ b \quad , \quad x_1 = 0$$

Let us then apply rules ait1, ase1, aev:

$$(\texttt{Wait}[u_2]; Q) \ \texttt{interrupt}[u_1]_{x_1} \ b \quad , \quad x_1 \leq u_1$$

Let us then apply *Act* with fresh clock x_2 (rules A13, A18, A6):

$$(\texttt{Wait}[u_2]_{x_2}; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad x_1 \leq u_1 \wedge x_2 = 0$$

Let us then apply rules ait1, ase1, await:

$$(\texttt{Skip}; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad 0 \leq x_1 - x_2 \leq u_1 \wedge x_2 = u_2 \wedge x_1 \leq u_1$$

Let us now remove clock x_2:

$$(\texttt{Skip}; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad u_2 \leq x_1 \leq u_1$$

Let us then apply rules ait1, ase2, aki:

$$Q\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad u_2 \leq x_1 \leq u_1$$

By expanding Q, we get:

$$(a \to \texttt{Wait}[u_2]; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad u_2 \leq x_1 \leq u_1$$

Let us then apply rules ait1, ase1, aev:

$$(\texttt{Wait}[u_2]; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad u_2 \leq x_1 \leq u_1$$

Let us then apply *Act* with fresh clock x_2 (rules A13, A18, A6):

$$(\texttt{Wait}[u_2]_{x_2}; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad u_2 \leq x_1 \leq u_1 \wedge x_2 = 0$$

Let us then apply rules ait1, ase1, await:

$$(\texttt{Skip}; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad u_2 \leq x_1 - x_2 \leq u_1 \wedge x_2 = u_2 \wedge x_1 \leq u_1$$

Let us now remove clock x_2:

$$(\texttt{Skip}; Q)\ \texttt{interrupt}[u_1]_{x_1}\ b \quad , \quad 2u_2 \leq x_1 \leq u_1$$

So it is now easy to see that the algorithm will go into an infinite loop with constraints of the form $i * u_2 \leq x_1 \leq u_1$, with i infinitely growing. $\Box$

Model checking When the set of reachable states is finite, i.e., when *reachAll* terminates, one can apply to the reachability graph finite-state model checking techniques, such as most techniques defined in [35] for STCSP, e.g., model checking with and without non-Zenoness assumption, and refinement checking.

Parametric model-checking One can also extend the techniques defined in [35] to perform parameter synthesis using parametric model checking. Instead of replying "yes" or "no" to a request (reachability analysis, refinement checking, etc.), one can output a constraint such that the request is valid or violated.

Unfortunately, in most cases, the set of reachable states in PSTCSP (as in other parametric timed formalisms) is infinite[8]. Hence the techniques (even on-the-fly) defined in the non-parametric framework do not apply anymore.

5.3 Parameter Synthesis Using the Inverse Method

The state space is often infinite, and classical techniques (even using on-the-fly algorithms) may not terminate. We show here how to adapt to PSTCSP the inverse method *IM* initially proposed in [5] for PTAs. Given a PTA A and a reference parameter valuation π, *IM* synthesizes a constraint K on the parameters such that, for all $\pi' \models K$, the time abstract behavior, i.e., the sequences of locations and actions, of A instantiated with π and A instantiated with π' are the same. This algorithm consists in generating runs starting from the initial state, and removing states incompatible with the reference valuation by appropriately refining K. The generation procedure is then restarted until no incompatible state is generated. This method guarantees the time-abstract equivalence of the behaviors. Hence, all linear time properties valid in A instantiated with π are also valid in A instantiated with π', and vice versa.

In order to adapt *IM* to the framework of PSTCSP, we need to check whether the constraint associated with a state is satisfied by a given parameter valuation. This refers to the following notion of π-compatibility.

Definition 5.2 (π-compatibility): Let M be a PSTCSP model, and $s = (P, V, C)$ be a state of M. The state s is said to be *π-compatible* if $\pi \models C$, and π-incompatible otherwise.

In order to characterize the properties of *IM*, we define the notion of trace as an alternating sequence of processes and actions.

Definition 5.3 (Trace): Given a PSTCSP model M and a run r of M of the form $(P_0, V_0, C_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (P_m, V_m, C_m)$, the *trace associated with* r is the alternating sequence of processes and actions $P_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} P_m$. The *trace set* of M is the set of all traces associated with the runs of M.

We give in Algorithm 2 the adaptation of $IM(\mathsf{M}, \pi)$ to PSTCSP. We consider in the following the model $\mathsf{M} = (Var, U, V_0, P, K_0)$. Starting with a constraint over the parameters $K = K_0$, we iteratively compute a growing set of reachable states. When a *π-incompatible* state (V, P, C) is encountered (i.e., when $\pi \not\models C$), K is refined as follows: a π-incompatible inequality J (i.e., such that $\pi \not\models J$) is selected within the projection of C onto the parameters U and the negation $\neg J$

[8] For timed systems, the state space is always infinite because of dense time. Here, we mean that the number of (symbolic) states (V, P, C) is infinite too.

of J is added to K. The procedure is then started again with this new K, and so on, until fixpoint is reached (i.e., all new states have been met before, or no new state is reachable). We finally return the intersection of the projection onto the parameters U of the constraints associated with all reachable states.

Algorithm 2: Algorithm $IM(\mathsf{M}, \pi)$

```
input  : PSTCSP model $\mathsf{M} = (Var, U, V_0, P, K_0)$
input  : Parameter valuation $\pi$
output: Constraint $K$ over the parameters
1  $i \leftarrow 0$;  $K \leftarrow K_0$;  $S \leftarrow \{(V_0, P, K)\}$
2  while True do
3      while there are $\pi$-incompatible states in $S$ do
4          Select a $\pi$-incompatible state $(V, P, C)$ of $S$ (i.e., s.t. $\pi \not\models C$) ;
5          Select a $\pi$-incompatible $J$ in $C_{/U}$ (i.e., s.t. $\pi \not\models J$) ;
6          $K \leftarrow K \wedge \neg J$ ;
7          $S \leftarrow \bigcup_{j=0}^{i} Post_{\mathsf{M}}^{j}(\{(V_0, P, K)\})$ ;
8      if $Post_{\mathsf{M}}(S) \subseteq S$ then
9          return $\bigcap_{(V,P,C)\in S} C_{/U}$ ;
10     $i \leftarrow i + 1$ ;
11     $S \leftarrow S \cup Post_{\mathsf{M}}(S)$ ;          /* $S = \bigcup_{j=0}^{i} Post_{\mathsf{M}}^{j}(\{(V_0, P, K)\})$ */
```

Actually, the two major steps of the algorithm are the following ones:

1. the iterative negation of the π-incompatible states (by negating a π-incompatible inequality J) prevents for any $\pi' \models K$ the behavior different from π;

2. the intersection of the constraints associated with all the reachable states guarantees that all the behaviors under π are allowed for all $\pi' \models K$.

Properties Most properties of *IM* for PTAs and its variants (see [5, 6]) also apply to our framework. In particular, *IM* preserves the equality of trace sets, as defined below.

Proposition 5.4: Let M be a PSTCSP model, and π a parameter valuation. Let $K = IM(\mathsf{M}, \pi)$. Then: (1) $\pi \models K$, and (2) for all $\pi' \in K$, the trace sets of $\mathsf{M}[\pi]$ and $\mathsf{M}[\pi']$ are the same.

Proof Using a reasoning similar to [5]. ■

As a consequence, all linear-time properties valid for $\mathsf{M}[\pi]$ are preserved in $\mathsf{M}[\pi']$, for all $\pi' \in K$. This is the case of properties expressed using the Linear Time Logics (LTL) [30], but also using the SE-LTL logics [13], which is a linear temporal logic constituted by both atomic state propositions and events.

Advantages The efficiency of *IM* in practice comes from the fact that the exploration of the state space is very partial; branches are cut as soon as they differ from π. Furthermore, in contrast to classical model checking techniques, transitions are not stored in memory; only states are needed (see Algorithm 2). Although *IM* is not guaranteed to output the weakest constraint (i.e., the largest set of parameters), it often does (see Section 5.4.2); and it is always guaranteed to output a dense set of parameter valuations in $|U|$ dimensions, both non-null and non-reduced to a point.

Termination of *IM* is not guaranteed in the general case; however, it terminates for all our case studies. For instance, the application of *IM* to Example 5.2 terminates for any non-null parameter valuation, although Algorithm *reachAll* does not terminate. It has been shown that termination is guaranteed for PTAs whose associated graph is acyclic. This can be extended to PSTCSP, if a process has no recursion (i.e., no cyclic dependencies between subprocesses).

Proposition 5.5: $\mathit{IM}(\mathsf{M}, \pi)$ terminates if M has no recursion.

Actually, whereas it is possible to find counterexamples for *IM* in the setting of PTAs, we were not able to exhibit any example in PSTCSP (with non-null parameter valuations) such that *IM* does not terminate. For instance, *IM* terminates for Example 5.2, although it contains a recursive definition (because process P is defined using Q, and Q itself defined using Q). This is not trivial, since a standard reachability analysis would go into an infinite loop, precisely because the recursion is under the parameterized `interrupt` construct, where u_1 can be arbitrarily big when compared to u_2. This result is of particular interest since parameter synthesis is undecidable for PSTCSP.

Furthermore, *IM* gives a criterion of *robustness*: it guarantees that, if the system is correct for π, it will also be correct for valuations *around* π (viz., for all valuations satisfying $\mathit{IM}(\mathsf{M}, \pi)$). This gives a quantitative measure of the *implementability* of a timed system.

Application to the Example Let us apply *IM* to M_{ex} and the following reference parameter valuation π: $u_1 = 1 \ \wedge \ u_2 = 2$. Again, since $\mathit{Var} = \emptyset$, we denote the states by (P, C), where P is the current process, and C the current constraint on X and U. Recall that $K_0 = \mathit{True}$.

We start with $i = 0$, $K = \mathit{True}$ and $S = \{s'_0\}$, with

$$s'_0 = ((a \rightarrow \texttt{Wait}[u_2]; b \rightarrow \texttt{Stop}) \ \texttt{interrupt}[u_1]_{x_1} \ c \rightarrow P, x_1 = 0).$$

The projection of $x_1 = 0$ onto the parameters gives *True*; hence, s'_0 is π-compatible and we perform $i \leftarrow i + 1$ and $S \leftarrow S \cup \mathit{Post}_{\mathsf{M}}(S)$.

Now, we have $i = 1$ and $S = \{s'_0, s'_1, s'_2\}$, with

$$s'_1 = ((\texttt{Wait}[u_2]_{x_2}; b \rightarrow \texttt{Stop}) \ \texttt{interrupt}[u_1]_{x_1} \ c \rightarrow P, 0 \leq x_1 \leq u_1 \wedge x_2 = 0)$$

and

$$s'_2 = (c \rightarrow P, \mathit{True}).$$

The projection onto the parameters of the constraint associated with both s'_1 and s'_2 gives *True*; hence, S is π-compatible and we perform again $i \leftarrow i+1$ and $S \leftarrow S \cup Post_{\mathsf{M}}(S)$.

Now, we have $i = 2$ and $S = \{s'_0, s'_1, s'_2, s_3\}$, with

$$s_3 = ((\texttt{Skip}; b \to \texttt{Stop}) \ \texttt{interrupt}[u_1]_{x_1} \ c \to P, u_2 \leq x_1 \leq u_1).$$

The projection onto the parameters of the constraint associated with s_3 gives $u_2 \leq u_1$, which is obviously π-incompatible. As a consequence, we negate this inequality, and add it to K, which gives $K = u_2 > u_1$. Afterwards, we perform $\bigcup_{j=0}^{i} Post_{\mathsf{M}}^{j}(\{(V_0, P, K)\})$; this gives a set of states similar to the last S computed above, except that s_3 is now absent from S, and all three states s'_0, s'_1, s'_2 contain the inequality $u_2 > u_1$ in their constraint. The fixpoint is reached, and the intersection of the constraints on the parameters is returned (viz., $u_2 > u_1$). □

By Proposition 5.4, for all $\pi' \models u_2 > u_1$, the trace set of $\mathsf{M}_{ex}[\pi']$ is the same as for $\mathsf{M}_{ex}[\pi]$. Note that this trace set is actually the one depicted in Figure 6 page 18.

It can also be shown that the application of *IM* to M_{ex} and a reference parameter valuation such that $u_2 \leq u_1$ (e.g., $u_1 = 2$ and $u_2 = 1$) leads to the result $u_2 \leq u_1$.

5.4 Implementation and Experiments

This work has been implemented within PAT [34, 37], a self-contained framework implemented in C# and able to support composing, simulating and automatic verification of concurrent, real-time systems and other domains. It comes with user friendly interfaces, featured model editor and animated simulator. Most importantly, PAT implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. To achieve good performance, advanced optimization techniques are implemented in PAT, e.g. partial order reduction, symmetry reduction, process counter abstraction, parallel model checking.

The implementation of PSTCSP within PAT allows in particular the use (within the process definitions) of complex data structures, such as counters, lists, sets, and more generally any structure and function defined by the user in C#.

One of the major issues in the synthesis of timing parameters is the handling of constraints on both clocks and parameters. Operations on such constraints (intersection, variable elimination, satisfiability, etc.) are by far more complex than equivalent operations on constraints on clocks, because the latter benefit from the efficient representation using DBMs. Unfortunately, most optimizations defined for DBMs do not apply to parametric timed constraints. In our setting, each state is implemented under the form of a pair (process id, constraint id), both under the form of a string. Although some processing is needed each time a new state is computed, an advantage is that the constraint equality test (when checking whether this new state has been met before) reduces to (trivial) string equality.

We present in the remainder of this section an optimization for state space reduction, as well as a set of case studies.

5.4.1 State Space Reduction

In PSTCSP, some states considered as different are actually equivalent. Consider the following two states:

$$s_1 = (\emptyset, \texttt{Wait}[u_1]_{x_1}\texttt{deadline}[u_2]_{x_2}, x_1 \leq x_2 \leq u_2)$$

$$s_2 = (\emptyset, \texttt{Wait}[u_1]_{x_2}\texttt{deadline}[u_2]_{x_1}, x_2 \leq x_1 \leq u_2)$$

It is obvious that $s_1 = s_2$, except the *names* of the clocks. Merging these states may lead to an exponential diminution of the number of states. Hence, we implemented a technique of *state normalization*: First, the clocks in the process are renamed so that the first one (from left to right) is named x_1, the second x_2, and so on. Second, the variables in the constraint are swapped accordingly. This technique solves this problem at the cost of several nontrivial operations (lists and strings sorting). We denote by *reachAll+* (resp. *IM+*) the version of *reachAll* (resp. *IM*) using this technique.

5.4.2 Experiments

We give in Table 1 the example name, the number $|U|$ of parameters and, for each algorithm, the number $|S|$ (resp. $|T|$) of states (resp. transitions)[9], the maximum number $|X|$ of clocks, and the computation time t on a Windows XP desktop computer with an Intel Quad Core 2.4 GHz processor with 4 GiB memory.[10]

Bridge is a bridge crossing problem for 4 persons within 17 min. Fischer$_i$ is the mutual exclusion protocol for i protocols. Jobshop is a scheduling problem. TrAHV is the train example from [3]. RCS$_i$ is a railway control system with i trains. When *reachAll* (resp. *reachAll+*) terminates, one can apply classical model checking techniques: for instance, we checked that all models are deadlock-free (except Jobshop which is precisely finite-state). When *reachAll* does not terminate (Bridge, Fischer), *IM* is interesting because it synthesizes constraints even for infinite symbolic state space case studies; and when *reachAll* terminates slowly (TrAHV), *IM* may synthesize constraints quickly. The reference valuation used for *IM* either is the standard valuation for the considered problem (Bridge, Jobshop, RCS$_i$, TrAHV) or has been computed in order to satisfy a well-known constraint of good behavior (Fischer$_i$).

[9] Recall that *IM* does *not* maintain transitions. Hence, the transition number for *IM* and *IM+* is only an integer maintained within the program for statistics purpose.

[10] Binaries, models and results are available on `www.comp.nus.edu.sg/~pat/par/`.

Case		*reachAll*				*reachAll+*				*IM*				*IM+*			
study	$\lvert U\rvert$	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert X\rvert$	t	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert X\rvert$	t	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert X\rvert$	t	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert X\rvert$	t
M_{ex}	2	8	14	2	0.008	8	14	2	0.006	3	5	2	0.004	3	5	2	0.005
M'_{ex}	2	8	14	2	0.008	8	14	2	0.006	8	14	2	0.016	8	14	2	0.008
Bridge	4	-	-	-	M.O.	-	-	-	M.O.	2.8k	5.5k	2	253	2.8k	5.5k	2	455
Fischer$_2$	2	-	-	-	M.O.	-	-	-	M.O.	44	75	2	0.086	45	77	2	0.103
Fischer$_3$	2	-	-	-	M.O.	-	-	-	M.O.	870	2004	3	3.38	313	730	3	0.723
Fischer$_4$	2	-	-	-	M.O.	-	-	-	M.O.	11k	31k	4	41.9	2k	5.8k	4	8.65
Fischer$_5$	2	-	-	-	M.O.	-	-	-	M.O.	133k	447k	5	1176	13k	44k	5	84.5
Fischer$_6$	2	-	-	-	M.O.	-	-	-	M.O.	-	-	-	M.O.	86k	342k	6	1144
Jobshop	8	14k	20k	2	21.0	12k	17k	2	18.1	1112	1902	2	17.1	877	1497	2	22.8
RCS$_2$	4	52	64	4	0.038	52	64	4	0.059	52	64	4	0.091	52	64	4	0.147
RCS$_3$	4	233	296	4	0.186	233	296	4	0.300	233	296	4	0.310	233	296	4	0.513
RCS$_4$	4	1070	1374	4	1.74	1070	1374	4	1.58	1070	1374	4	1.40	1070	1374	4	2.38
RCS$_5$	4	5.6k	7.2k	4	10.5	5.6k	7.2k	4	9.54	5.6k	7.2k	4	7.83	5.6k	7.2k	4	16.7
RCS$_6$	4	34k	43k	4	91.7	34k	43k	4	54.5	34k	43k	4	60.4	34k	43k	4	91.3
TrAHV	6	7.2k	13k	6	14.2	7.2k	13k	6	15.8	227	321	6	0.555	227	321	6	0.655

Tab. 1: Application of algorithms for parameter synthesis using PAT

Furthermore, the constraint output has several advantages. First, it solves the good parameter problem, and may even output *all* correct parameter valuations. For instance, the constraint synthesized for Fischer ($\delta < \gamma$) is known to be the weakest constraint guaranteeing mutual exclusion. Second, it always gives a criterion of robustness to the system, by defining a safety domain around each parameter, guaranteeing that the system will keep the same (time-abstract) behavior, as long as all parameters remain within K. Different from a simple "ball" output by robust timed automata techniques, this domain is a convex constraint in $|U|$ dimensions. Third, it happens that the constraint is *True* (e.g., RCS$_i$ for all i). In this case, one can safely *refine* the model by removing all timing constructs (`Wait`, `deadline`, etc.). Although this might be checked using refinement techniques in STCSP for one particular parameter valuation, we prove it here for *any* parameter valuation – and the designers of the RCS example were actually not even aware of this possible refinement.

As for the number of clocks, it is significantly smaller than equivalent models for PTAs for some case studies: for instance, the Bridge case study would obviously require 4 clocks because there are 4 independent processes in parallel. Similarly, the RCS$_i$ case study would require at least i clocks, one by train (plus some other clocks for the environment); however, in our setting, the maximum number of clocks is constant, and equal to 4, for all i. Beyond the fact that it has been shown that the fewer clocks, the more efficient real-time model checking is [11], a smaller number of clocks implies a more compact state space in our setting: constraints are represented using arrays and matrices; the fewer clocks, the smaller the constraints are, the more compact the state space is.

Also observe that, when *IM+* indeed reduces the number of states, it is much more efficient than *IM*, not only w.r.t. memory, but also w.r.t. time (e.g.,

Fischer_i for all i). However, with no surprise, when no state duplication is met (e.g., Bridge), viz., when the state space is not reduced using this technique, the computation time is bigger. Although reducing this computation is a subject of ongoing work, we do not consider it as a significant drawback: parameter synthesis' largest limitations are usually non-termination and memory saturation. Slower analyses for some case studies (up to +80% for Bridge) are acceptable when others benefit from a dramatic memory (and time) reduction (-90% for Fischer_5), allowing parameter synthesis even when *IM* goes out of memory (Fischer_6).

Most importantly, our framework is efficient: some case studies handle more than 100,000 reachable symbolic states in a very reasonable time, which, as far as we know, is unseen for parametric timed frameworks. As far as we know, no other tool performs parameter synthesis for timed extensions of CSP; as for other formalisms, fair comparisons would be difficult due to model translations: whereas translations between PTAs and Petri Nets are rather straightforward, their translation into process algebra is much trickier.

6 Conclusion and Future Work

We introduced Parametric Stateful Timed CSP, an intuitive formalism for reasoning parametrically in hierarchical real-time concurrent systems with shared variables and complex data structures. A simple semi-algorithm *reachAll* computing the set of reachable states is not guaranteed to terminate, as we showed that parameter synthesis is undecidable. We then adapted the inverse method *IM*, which synthesizes a set of parameters around a reference parameter valuation, guaranteeing the same time abstract behavior (in term of traces), and providing the system with a measure of robustness. *IM* behaves well in practice, and is given a sufficient termination condition. Our implementation within PAT leads to efficient parameter synthesis, handling more than 100,000 reachable symbolic states.

As future work, we wish to improve the state space representation, following the lines of the optimization of Section 5.4.1, and develop further state space reduction techniques. Other synthesis algorithms should also be developed or adapted, for instance following the lines of algorithms for PTAs [6]. In particular, parametric refinement checking is the subject of ongoing work.

Acknowledgment

Yang Liu is supported by research grant "Research and Development in the Formal Verification of System Design and Implementation". Jun Sun is supported by research grant "IDD11100102 / IDG31100105" from Singapore University of Technology and Design. Jin-Song Dong is supported by MOE T2 Project "Advanced Model Checking Systems".

We are grateful to Zhu Huiquan for solving several implementation issues.

References

[1] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *ICALP'90*, ICALP'90, pages 322–335. Springer-Verlag, 1990. 13

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. 2

[3] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *STOC'93*, pages 592–601. ACM, 1993. 2, 13, 15, 16, 17, 18, 27

[4] Rajeev Alur and Parthasarathy Madhusudan. Decision problems for timed automata: A survey. In *SFM-RT'04*, volume 3185 of *LNCS*, pages 1–24. Springer-Verlag, 2004. 14

[5] É. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *Int. J. of Found. of Comput. Sci.*, 20(5):819–836, 2009. 2, 3, 23, 24

[6] Étienne André and Romain Soulat. Synthesis of timing parameters satisfying safety properties. In *RP'11*, volume 6945 of *LNCS*, pages 31–44. Springer, 2011. 24, 29

[7] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *CAV'01*, pages 368–372. Springer-Verlag, 2001. 2

[8] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In *CONCUR '98*, CONCUR '98, pages 470–484. Springer-Verlag, 1998. 13

[9] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *CAV'99*, volume 1633 of *LNCS*, pages 341–353. Springer, 1999. 10

[10] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. Beyond liveness: Efficient parameter synthesis for time bounded liveness. In *FORMATS'05*, pages 81–94, 2005. 2

[11] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *LCPN'03*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003. 10, 28

[12] Batrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36:145–182, 1998. 13

[13] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *IFM'04*, volume 2999 of *LNCS*, pages 128–147, 2004. 24

[14] R. Clarisó and J. Cortadella. The octahedron abstract domain. *Science of Computer Programming*, 64(1):115–139, 2007. 2

[15] A. Collomb–Annichini and M. Sighireanu. Parameterized reachability analysis of the IEEE 1394 Root Contention Protocol using TReX. In *RT-TOOLS'01*, 2001. 2

[16] P.R. D'Argenio, J.P. Katoen, T.C. Ruys, and G.J. Tretmans. The bounded retransmission protocol must be on time! In *TACAS'97*. Springer, 1997. 2

[17] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993. 8, 13

[18] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212. Springer-Verlag, 1990. 10

[19] C.J. Fidge, I.J. Hayes, and G. Watson. The deadline command. *IEE Proceedings—Software*, 146(2):104–111, 1999. 7, 8

[20] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994. 13

[21] Thomas A. Henzinger and Howard Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *FMIA'95*, pages 265–282, 1995. 2

[22] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985. 3, 7

[23] Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining specification techniques for processes, data and time. In *IFM'02*, pages 245–266, 2002. 3

[24] M. Knapik and W. Penczek. Bounded model checking for parametric time automata. In *SUMo'10*, 2010. 2

[25] Hee-Hwan Kwak, Insup Lee, Anna Philippou, Jin-Young Choi, and Oleg Sokolsky. Symbolic schedulability analysis of real-time systems. In *IEEE RTSS'98*, pages 409–418, 1998. 2

[26] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. 2

[27] Brendan P. Mahony and Jin Song Dong. Overview of the semantics of TCOZ. In *IFM'99*, pages 66–85, 1999. 3

[28] J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *LICS'03*, pages 198–. IEEE Computer Society, 2003. 13

[29] J. Ouaknine and J. Worrell. Timed CSP = closed timed ϵ-automata. *Nordic Journal of Computing*, 10:99–133, 2003. 13

[30] Amir Pnueli. The temporal logic of programs. In *SFCS'77*, pages 46–57. IEEE Computer Society, 1977. 24

[31] S. Qin, J.S. Dong, and W.-N. Chin. A semantic foundation for TCOZ in unifying theories of programming. In *FME'03*, pages 321–340, 2003. 7, 8

[32] S. Schneider. *Concurrent and Real-time Systems*. John Wiley and Sons, 2000. 3

[33] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley and Sons, 1986. 4

[34] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV'09*, volume 5643 of *LNCS*. Springer, 2009. 26

[35] J. Sun, Y. Liu, J.S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *ICFEM'09*, volume 5885 of *LNCS*, pages 581–600, 2009. 3, 5, 8, 9, 10, 11, 12, 15, 18, 22, 23

[36] L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric model-checking of time petri nets with stopwatches using the state-class graph. In *FORMATS'08*, pages 280–294. Springer-Verlag, 2008. 3

[37] Jun Sun Yang Liu and Jin Song Dong. PAT 3: An extensible architecture for building multi-domain model checkers. In *ISSRE 2011*, 2011. Accepted. 26

[38] T. Yoneda, T. Kitai, and C. J. Myers. Automatic derivation of timing constraints by failure analysis. In *CAV'02*, pages 195–208. Springer-Verlag, 2002. 2

A Firing Rules for PSTCSP

Given a program pr and a valuation V, the valuation obtained by executing pr with V is denoted as $pr(V)$. Let $\mathit{active}(V, P)$ be the set of enabled events given P and V, i.e., the set of events that can be fired at the current state (and which lead to states with satisfiable constraints). Let αP denote the alphabet of process P. Process of the form $P \parallel Q$ is transformed to $P \parallel [\alpha P \cap \alpha Q]Q$.

We give below all firing rules for PSTCSP.

$$\frac{}{(V, \texttt{Skip}, C) \overset{\checkmark}{\rightsquigarrow} (V, \texttt{Stop}, C^{\uparrow})} \; (aki)$$

$$\frac{}{(V, e \rightarrow P, C) \overset{e}{\rightsquigarrow} (V, P, C^{\uparrow})} \; (aev)$$

$$\frac{}{(V, a\{pr\} \rightarrow P, C) \overset{a}{\rightsquigarrow} (pr(V), P, C^{\uparrow})} \; (aac)$$

$$\frac{V \models b}{(V, \texttt{if } b \texttt{ then } \{P\} \texttt{ else } \{Q\}, C) \overset{\tau}{\rightsquigarrow} (V, P, C^{\uparrow})} \; (co2)$$

$$\frac{V \not\models b}{(V, \texttt{if } b \texttt{ then } \{P\} \texttt{ else } \{Q\}, C) \overset{\tau}{\rightsquigarrow} (V, Q, C^{\uparrow})} \; (co3)$$

$$\frac{(V, P, C) \overset{a}{\rightsquigarrow} (V', P', C')}{(V, P|Q, C) \overset{a}{\rightsquigarrow} (V', P', C' \wedge \mathit{idle}(Q))} \; (aex1)$$

$$\frac{(V, Q, C) \overset{a}{\rightsquigarrow} (V', Q', C)}{(V, P|Q, C) \overset{a}{\rightsquigarrow} (V', Q', C' \wedge \mathit{idle}(P))} \; (aex2)$$

$$\frac{(V, P, C) \overset{a}{\rightsquigarrow} (V', Q', C')}{(V, P \setminus E, C) \overset{a}{\rightsquigarrow} (V', Q', C')} \; (ahi1)$$

$$\frac{(V, P, C) \overset{a}{\rightsquigarrow} (V', Q', C') \;, \quad \mathit{active}(V, P, C) \cap E \neq \emptyset \;, \quad a \notin E}{(V, P \setminus E, C) \overset{a}{\rightsquigarrow} (V', Q', C' \wedge C)} \; (ahi2)$$

$$\frac{(V, P, C) \overset{a}{\rightsquigarrow} (V', Q', C'), \mathit{active}(V, P, C) \cap E \neq \emptyset \;, \quad a \in E}{(V, P \setminus E, C) \overset{\tau}{\rightsquigarrow} (V', Q', C' \wedge C)} \; (ahi3)$$

$$\frac{(V, P, C) \overset{a}{\rightsquigarrow} (V', P', C') \;, \quad \checkmark \notin \mathit{active}(P, V, C)}{(V, P; Q, C) \overset{a}{\rightsquigarrow} (V', P'; Q, C')} \; (ase1)$$

$$\frac{(V, P, C) \overset{\checkmark}{\rightsquigarrow} (V', P', C')}{(V, P; Q, C) \overset{\tau}{\rightsquigarrow} (V, Q, C \wedge C')} \; (ase2)$$

$$\frac{(V,P,C) \overset{a}{\rightsquigarrow} (V',P',C')\ ,\quad a \notin E}{(V, P \parallel [E]Q, C) \overset{a}{\rightsquigarrow} (V', P' \parallel [E]Q, C' \wedge idle(Q))}\ (apa1)$$

$$\frac{(V,Q,C) \overset{a}{\rightsquigarrow} (V',Q',C')\ ,\quad a \notin E}{(V, P \parallel [E]Q, C) \overset{a}{\rightsquigarrow} (V', P \parallel [E]Q', C' \wedge idle(P))}\ (apa2)$$

$$\frac{(V,P,C) \overset{e}{\rightsquigarrow} (V,P',C')\ ,\quad (V,Q,C) \overset{e}{\rightsquigarrow} (V,Q',C'')\ ,\quad e \in E}{(V, P \parallel [E]Q, C) \overset{e}{\rightsquigarrow} (V, P' \parallel [E]Q', C' \wedge C'')}\ (apa3)$$

$$\frac{(V,Q,C) \overset{a}{\rightsquigarrow} (V',Q',C')\ ,\quad P \doteq Q}{(V,P,C) \overset{a}{\rightsquigarrow} (V',Q',C')}\ (ade)$$

$$\frac{}{(V, \mathtt{Wait}[u]_x, C) \overset{\tau}{\rightsquigarrow} (V, \mathtt{Skip}, C^{\uparrow} \wedge x = u)}\ (await)$$

$$\frac{(V,P,C) \overset{\tau}{\rightsquigarrow} (V',P',C')}{(V, P\ \mathtt{timeout}[u]_x\ Q, C) \overset{\tau}{\rightsquigarrow} (V', P'\ \mathtt{timeout}[u]_x\ Q, C' \wedge x \le u)}\ (ato1)$$

$$\frac{(V,P,C) \overset{e}{\rightsquigarrow} (V',P',C')}{(V, P\ \mathtt{timeout}[u]_x\ Q, C) \overset{e}{\rightsquigarrow} (V', P', C' \wedge x \le u)}\ (ato2)$$

$$\frac{}{(V, P\ \mathtt{timeout}[u]_x\ Q, C) \overset{\tau}{\rightsquigarrow} (V, Q, C^{\uparrow} \wedge x = u \wedge idle(P))}\ (ato3)$$

$$\frac{(V,P,C) \overset{a}{\rightsquigarrow} (V',P',C')}{(V, P\ \mathtt{interrupt}[u]_x\ Q, C) \overset{a}{\rightsquigarrow} (V', P'\ \mathtt{interrupt}[u]_x\ Q, C' \wedge x \le u)}\ (ait1)$$

$$\frac{}{(V, P\ \mathtt{interrupt}[u]_x\ Q, C) \overset{\tau}{\rightsquigarrow} (V, Q, C^{\uparrow} \wedge x = u \wedge idle(P))}\ (ait2)$$

$$\frac{(V,P,C) \overset{\tau}{\rightsquigarrow} (V',P',C')}{(V, P\ \mathtt{within}[u]_x, C) \overset{\tau}{\rightsquigarrow} (V', P'\ \mathtt{within}[u]_x, C' \wedge x \le u)}\ (awi1)$$

$$\frac{(V,P,C) \overset{e}{\rightsquigarrow} (V',P',C')}{(V, P\ \mathtt{within}[u]_x, C) \overset{e}{\rightsquigarrow} (V', P', C' \wedge x \le u)}\ (awi2)$$

$$\frac{(V,P,C) \overset{a}{\rightsquigarrow} (V',P',C')\ ,\quad a \neq \checkmark}{(V, P\ \mathtt{deadline}[u]_x, C) \overset{a}{\rightsquigarrow} (V', P'\ \mathtt{deadline}[u]_x, C' \wedge x \le u)}\ (adl1)$$

$$\frac{(V,P,C) \overset{\checkmark}{\rightsquigarrow} (V',P',C')}{(V, P\ \mathtt{deadline}[u]_x, C) \overset{\checkmark}{\rightsquigarrow} (V', P', C' \wedge x \le u)}\ (adl2)$$

Index