Parameter Synthesis for Hierarchical Concurrent Real-Time Systems

Étienne André*, Yang Liu†, Jun Sun‡ and Jin-Song Dong§
*LIPN, CNRS UMR 7030, Université Paris 13, France
Email: Etienne.Andre@lipn.univ-paris13.fr
†Temasek Laboratories, National University of Singapore
Email: tslliuya@nus.edu.sg
‡Singapore University of Technology and Design
Email: sunjun@sutd.edu.sg
§School of Computing, National University of Singapore
Email: dongjs@comp.nus.edu.sg

***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, 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; model checking; robustness; refinement.**

I. 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 (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 timed 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: 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 [21]. 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 [17], 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, has been used to solve several case studies; Unfortunately, it can hardly verify even medium sized examples due to 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 [14] or Root Contention protocol [13] using TREX [7]) and asynchronous circuits (e.g., [30], [12]). 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., [8], 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 (expressed, e.g., in LTL).

In [20], parametric analyses of scheduling problems are 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 [29] 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) [18] is a powerful event based formalism for describing patterns of interaction in concurrent systems. Timed CSP (see, e.g., [25]) 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 [27]. 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 [27], but does not often terminate. We also extend the inverse method [5] to PSTCSP, and give a sufficient termination condition; this algorithm behaves 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 PSyHCoS 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 Timed Communicating Object Z (TCOZ) [22] and CSP-OZ-DC [19]. 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 II. We introduce PSTCSP in Section III and study its expressiveness and decidability questions in Section IV. We introduce algorithms for parameter synthesis in Section V, and apply them to case studies. We conclude in Section VI.

II. 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: 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 $X = \{x_1, \ldots, x_H\} \subset \mathcal{X}$, a *clock valuation* is a function $w : X \to \mathbb{R}_{\geq 0}$. We will often identify w with the point $(w(x_1), \ldots, w(x_H))$. Given $d \in \mathbb{R}_{\geq 0}$, we use $X + d$ to denote $\{x_1 + d, \ldots, x_H + d\}$.

We also assume a set $\mathcal{U}$ of *parameters* (i.e., unknown constants) disjoint with $\mathcal{V}ar$ and $\mathcal{X}$. Given $U = \{u_1, \ldots, u_M\} \subset \mathcal{U}$, a *parameter valuation* is a function $\pi : U \to \mathbb{R}_{\geq 0}$. We will often identify π with the point $(\pi(u_1), \ldots, \pi(u_M))$.

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 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 $\exists w : <w, \pi> \models C$, then π *satisfies* C, denoted by $\pi \models C$.

Similarly, 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 $X' \subseteq X$, 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 of all clocks. We denote by $C_{/X'}$ the constraint $\exists (X \setminus X') : C$. We define $C^{\uparrow}$ as the constraint over X and U obtained from C by delaying time, i.e., by renaming X' with X in the 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

[1] Using variable elimination techniques such as Fourier-Motzkin [26].

$P \doteq$	$\texttt{Stop}$	inaction
	$\mid \texttt{Skip}$	termination
	$\mid e \to P$	event prefixing
	$\mid a\{prg\} \to P$	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

Figure 1. Syntax of PSTCSP processes

$\checkmark \in \Sigma$; $\Sigma_\tau = \Sigma \cup \{\tau\}$. Furthermore, the following event naming conversion is adapted: $e \in \Sigma$; $a \in \Sigma_\tau$; $E \subseteq \Sigma$.

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 $Runs(\mathcal{L})$ the set of runs of $\mathcal{L}$.

III. Syntax and Semantics of PSTCSP

A. Syntax

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 $\mathsf{M} = (Var, U, V_0, P, K_0)$ where $Var \subset \mathcal{V}ar$, $U \subset \mathcal{U}$, V_0 is the initial variable valuation, $P \in \mathcal{P}$, 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., $K_0 = \{min \leq max\}$.

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

B. Informal Semantics

We first briefly describe the untimed constructs, which are identical to STCSP. Process `Stop` does nothing but idling. Process `Skip` terminates, possibly after idling for some time. Process $e \to P$ engages in event e first and then behaves as P. Note that e may serve as a synchronization barrier, if combined with parallel composition. In order to seamlessly integrate data operations, sequential programs may be attached with events. Process $a\{prg\} \to P$ performs data operation a (i.e., executing the sequential prg 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 $v_1, v_2 \in Var$) or a more complicated sequential program. A conditional choice is written as $\texttt{if } (b) \ \{P\} \ \texttt{else} \ \{Q\}$. Process $P \vert Q$ 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 [18]) or shared variables.

We now explain the parametric timed constructs.

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

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

[3] For simplicity, in the discussion, we leave out external and internal choices from the classic CSP [18]. Nevertheless, both constructions are defined in PSTCSP, implemented, and used in our case studies.

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 [16], 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 [24], the authors use Unifying Theory of Programming in order to formalize the semantics of TCOZ. As in [16], 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.

Different from [24], [16], we here choose to stick to the semantics of STCSP [27] 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].

C. Example: Fischer Mutual Exclusion

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

Example 3.2: Fischer's mutual exclusion algorithm is modeled as $(Var, U, v_i, FME, True)$, where $U = \{\delta, \gamma\}$, and $Var = \{turn, cnt\}$. 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 (no process is attempting initially) and cnt to 0 (no process is in the critical section initially). Process FME is defined as follows.

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

where n is a constant representing the number of processes. Process $proc(i)$ models a process with a unique integer identify i. If $turn$ is -1 (i.e., no other process is attempting), $proc(i)$ behaves as specified by $Active(i)$. In $Active(i)$, $turn$ is first set to i (i.e., the ith process is now attempting) by action $update.i$. Note that $update.i$ must occur within δ time units (captured by $\texttt{within}[\delta]$). Next, the process idles for γ time units. 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.

A classical parameter synthesis problem is to find values of δ and γ for which mutual exclusion is guaranteed. A solution will be given in Section V-C2. □

D. 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 $P\ \texttt{timeout}[u]\ Q$, an implicit clock should start whenever this process is activated. A clock starts ticking once the process becomes activated. 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 $\texttt{Wait}[u_1]$ (say x_1), one with $\texttt{Wait}[u_2]$ (say x_2) and one with P (because of $\texttt{interrupt}[u_3]$, say x_3). Clocks x_1 and x_3 are starting at the same time because the execution of $\texttt{interrupt}$ is linked with $\texttt{Wait}[u_1]$. In contrast, clock x_2 starts only when $\texttt{Wait}[u_1]$ 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 [27] how to systematically associate clocks with timed processes. We write $\texttt{Wait}[u]_x$ to denote that the process $\texttt{Wait}[u]$ is associated with clock x. Given a process P and a clock x, we use function $Act(P, x)$ to define the process with activated clocks. The definition of Act is very similar to the one for STCSP (see [27]) and is given in [6]. For instance, we have $Act(P\ \texttt{timeout}[u]\ Q, x) = Act(P, x)\ \texttt{timeout}[u]_x\ Q$, which means that we associate clock x to the $\texttt{timeout}$ construct and recursively activate P with x. However, Q is not concerned because it is not activated yet.

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 $P\ \texttt{timeout}[u]_x\ Q$ contains x and the clocks associated with P.

E. Semantics

In the following, we introduce the semantics for PSTCSP in terms of states containing constraints over X and U. Formally, 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 $C \in \mathcal{K}_{X \cup U}$. 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 $<w, \pi> \models C$.

1) Idling Function: We adapt in the following the function $idle$, defined in [27], which, given a process, 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 2 shows the detailed definition. Rules $i1$ to $i5$ state that if the process is untimed and none of

$$
\begin{array}{lll}
idle(\texttt{Stop}) & = True & i1 \\
idle(\texttt{Skip}) & = True & i2 \\
idle(e \to P) & = True & i3 \\
idle(a\{prg\} \to P) & = True & i4 \\
idle(\texttt{if } (b) \ \{P\} \ \texttt{else} \ \{Q\}) & = True & i5 \\
idle(P|Q) & = idle(P) \wedge idle(Q) & i6 \\
idle(P \setminus E) & = idle(P) & i7 \\
idle(P;Q) & = idle(P) & i8 \\
idle(P \parallel Q) & = idle(P) \wedge idle(Q) & i9 \\
idle(\texttt{Wait}[u]_x) & = x \leq u & i10 \\
idle(P \ \texttt{timeout}[u]_x \ Q) & = x \leq u \wedge idle(P) & i11 \\
idle(P \ \texttt{interrupt}[u]_x \ Q) & = x \leq u \wedge idle(P) & i12 \\
idle(P \ \texttt{within}[u]_x) & = x \leq u \wedge idle(P) & i13 \\
idle(P \ \texttt{deadline}[u]_x) & = x \leq u \wedge idle(P) & i14 \\
idle(P) & = idle(Q) \quad \text{if } P \doteq Q & i15
\end{array}
$$

Figure 2. Idling calculation

its subprocesses is activated, then the function returns true. Intuitively, it means that the process may idle for arbitrary amount of time. Rules $i6$ to $i9$ 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 $i6$) or a parallel composition (rule $i9$) 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 $i10$ to $i14$ 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.

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 $S = \{(V, P, C) \in \mathcal{V}(Var) \times \mathcal{P} \times \mathcal{K}_{X \cup U}\}$, $s_0 = (V_0, P, K_0)$ 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 $((V, P, C), a, (V', P', C'_{/cl(P')})) \in \Rightarrow$.

The transition relation $\rightsquigarrow$ is specified by a set of rules; we give in Figure 3 the rules for the parametric timed constructs of PSTCSP. Other rules are quite similar to STCSP, and are detailed in [6].

The rule $await$ defining $\rightsquigarrow$ for $\texttt{Wait}$ says that a τ-transition occurs exactly when $x = u$. Intuitively, $C^\uparrow \wedge x = u$ denotes the time when u time units elapsed since x has started. Other rules can be explained in a similar manner.

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

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

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

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

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

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

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

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

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

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

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

Figure 3. Firing rules for the parametric timed constructs

state (V', P', C'). 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.

Example 3.4: Let us consider the following 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 $\texttt{Skip}$; this gives the 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 $(V, \texttt{Skip} \ \texttt{interrupt}[u_2]_{x_1}, C')$ 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. □

IV. Expressiveness and Undecidability

A. Expressiveness

We first state that STCSP is equivalent to closed timed ϵ-automata [23], i.e., timed safety automata with ϵ-transitions [10] and exclusively closed guards and invariants (i.e., whose inequalities are of the form $e \leq e'$, with e, e' linear terms).

Lemma 4.1: Stateful Timed CSP is as expressive as closed timed ϵ-automata.

Proof: We first show that STCSP without the $\texttt{deadline}$ and the $\texttt{within}$ constructs is equivalent to Timed CSP. It is known that all Timed CSP constructs, including $\texttt{timeout}$

and `interrupt` can be derived from `Wait`$[d]$ and CSP constructs [15]. It has been shown that the expressive power of Timed CSP is equal to closed timed ϵ-automata [23]. 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. ■

We define parametric closed timed ϵ-automata as a parametric extension of closed timed ϵ-automata, following the parameterization of TAs into PTAs [3]. 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 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.

B. 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: does an execution starting from a process P_0 lead to a given process P?

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., $\{\pi \in U \mid \exists V, C : (V_0, P_0, C_0) \rightsquigarrow (V, P, C) \in Runs(\mathsf{M}[\pi])\}$.

The membership problem is decidable for PSTCSP: it suffices to consider the STCSP model $\mathsf{M}[\pi]$ and solve this problem using techniques developed in [27].

Theorem 4.3 (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: By reduction of the halting problem for 2-counter machines to the problem of testing if there exists a parameter valuation consistent with a PSTCSP model, following the reduction used in [3] (see proof in [6]). ■

An immediate corollary is that parameter synthesis is undecidable in general.

V. PARAMETER SYNTHESIS

We use in this section a model $\mathsf{M}_{ex} = \{\emptyset, \{u_1, u_2\}, P, True\}$ with $P \doteq (a \rightarrow$ `Wait`$[u_2]; b \rightarrow$ `Stop`$)$ `interrupt`$[u_1]$ $c \rightarrow P$, in order to illustrate our algorithms.

A. State Space Exploration

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 can define a semi-algorithm $reachAll(\mathsf{M})$ as a classical fixpoint computation, which iteratively computes $Post^*_{\mathsf{M}}(S)$ (and does not terminate if it is infinite).

Application: Let us apply $reachAll$ to M_{ex}. Since we have no variable, we denote for the sake of conciseness the states by (P, C), where P is the current process, and C the current constraint over X and U. We get the following states, depicted in Figure 4 using a directed graph whose edges are labeled with actions.

$s_0 = ((a \rightarrow$ `Wait`$[u_2]; b \rightarrow$ `Stop`$)$ `interrupt`$[u_1]_{x_1}$ $c \rightarrow P$, $0 \leq x_1 \leq u_1)$

$s_1 = (($`Wait`$[u_2]_{x_2}; b \rightarrow$ `Stop`$)$ `interrupt`$[u_1]_{x_1}$ $c \rightarrow P$, $0 \leq x_2 \leq x_1 \leq u_1 \wedge x_2 \leq u_2)$

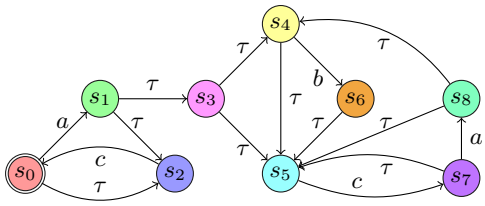

Figure 4. States reachable in model M_{ex}

$s_2 = (c \to P, \mathit{True})$
$s_3 = ((\texttt{Skip}; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P\ ,\ u_2 \leq x_1 \leq u_1)$
$s_4 = ((b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P\ ,\ u_2 \leq x_1 \leq u_1)$
$s_5 = (c \to P\ ,\ u_2 \leq u_1)$
$s_6 = (\texttt{Stop}\ \texttt{interrupt}[u_1]_{x_1}\ c \to P\ ,\ u_2 \leq x_1 \leq u_1)$
$s_7 = (P\ ,\ u_2 \leq u_1)$
$s_8 = ((\texttt{Wait}[u_2]_{x_2}; b \to \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \to P,$
$0 \leq x_2 \leq x_1 \leq u_1 \wedge u_2 \leq u_1)$

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. □

Proposition 5.1: Let M be a PSTCSP model. Then Algorithm $\mathit{reachAll}(\mathsf{M})$ does not terminate in the general case.

Proof: See counterexample in Example 5.2. ■

Example 5.2: Consider the PSTCSP model $\mathsf{M} = (\emptyset, \{u_1, u_2\}, \emptyset, P, \mathit{True})$ where $P \doteq Q\ \texttt{interrupt}[u_1]\ b \to \texttt{Skip}$ and $Q \doteq a \to \texttt{Wait}[u_2]; Q$. Starting from the initial state, $\mathit{reachAll}$ will go into an infinite loop, generating in particular states of the form $(\emptyset, P, i * u_2 \leq x_1 \leq u_1)$, with i infinitely growing (details are given in [6]). □

Model Checking: When the set of reachable states is finite, one can apply to the reachability graph finite-state model checking techniques, such as most techniques defined in [27] for STCSP (e.g., model checking with and without non-Zenoness assumption, and refinement checking). One can also extend such techniques to perform parameter synthesis. Instead of replying "yes" or "no" to a request, 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[5]. Hence the techniques (even on-the-fly) defined in the non-parametric framework do not apply anymore.

B. Parameter Synthesis Using the Inverse Method

We show here how to adapt to PSTCSP the inverse method IM 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. 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 5.3 (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 $\pi \models C$.

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

Definition 5.4 (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 Figure 5 the adaptation of $IM(\mathsf{M}, \pi)$ to PSTCSP. Starting with a constraint $K = K_0$, 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 $\neg J$ 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 U of the constraints associated with all reachable states.

Most properties of IM for PTAs also apply to our framework. In particular, IM preserves the equality of trace sets, as defined below.

Proposition 5.5: 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 LTL, but also using the SE-LTL logics [11].

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 Figure 5). Although IM is not guaranteed to output the weakest constraint (i.e., the largest set of parameters), it

[5]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.

Require: PSTCSP model $\mathsf{M} = (Var, U, V_0, P, K_0)$
Require: Parameter valuation π
Ensure: 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 π-incompatible states in S **do**
4: Select a π-incompatible state (V, P, C) of S
5: Select a π-incompatible J in $C_{/U}$
6: $K \leftarrow K \wedge \neg J$
7: $S \leftarrow \bigcup_{j=0}^{i} Post^j_{\mathsf{M}}(\{(V_0, P, K)\})$
8: **end while**
9: **if** $Post_{\mathsf{M}}(S) \subseteq S$ **then**
10: **return** $\bigcap_{(V,P,C)\in S} C_{/U}$
11: **end if**
12: $i \leftarrow i + 1$; $S \leftarrow S \cup Post_{\mathsf{M}}(S)$
13: **end while**

Figure 5. Algorithm $IM(\mathsf{M}, \pi)$

often does (see Section V-C2); 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 5.6: $IM(\mathsf{M}, \pi)$ 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. 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 $IM(\mathsf{M}, \pi)$). This gives a quantitative measure of the *implementability* of a timed system.

Application: Let us apply IM to M_{ex} and π: $u_1 = 1 \wedge u_2 = 2$. One can intuitively understand IM by looking at the graph of Figure 4: states s_0, s_1, and s_2 are computed with a constraint projected onto U equal to $True$, hence π-compatible. Then state s_3 is computed, with constraint $u_2 \leq u_1$, which is π-incompatible because π is such that $u_2 > u_1$. When computing again S with $u_2 > u_1$, the constraint associated to s_3 now becomes unsatisfiable, and this state is discarded. Hence, fixpoint is reached, and the intersection of projection of the constraints onto U is returned (viz., $u_2 > u_1$). By Proposition 5.5, for all $\pi' \models u_2 > u_1$, the trace set of $\mathsf{M}_{ex}[\pi']$ is the same as for $\mathsf{M}_{ex}[\pi]$.

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

C. Implementation and Experiments

This work has been implemented within PSyHCoS (standing for *Parameter SYnthesis for Hierarchical COncurrent Systems*), a self-contained framework implemented in C# and able to support composing, simulating and automatic verification of concurrent real-time systems. The tool adopts some bits and pieces from PAT's model checking library [28]. PSyHCoS comes with user friendly interfaces, featured model editor and animated simulator.

The implementation of PSTCSP within PSyHCoS allows in particular the use (within the process definitions) of data structures, such as counters, 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.

1) State Space Reduction: In PSTCSP, some states considered as different are actually equivalent. Consider states $s_1 = (\emptyset, \texttt{Wait}[u_1]_{x_1}\texttt{deadline}[u_2]_{x_2}, x_1 \leq x_2 \leq u_2)$ and $s_2 = (\emptyset, \texttt{Wait}[u_1]_{x_2}\texttt{deadline}[u_2]_{x_1}, x_2 \leq x_1 \leq u_2)$. 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.

2) Experiments: We give in Table I the example name, the number $|U|$ of parameters and, for each algorithm, the number $|S|$ (resp. $|T|$) of states (resp. transitions), 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.[6]

Bridge is a classical bridge crossing problem for 4 persons within 17 minutes. 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$_i$).

Furthermore, the constraint output has several advantages. First, it solves the good parameter problem. For instance, the constraint synthesized for Fischer ($\delta < \gamma$) is 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. Beyond the fact that it has been shown that the fewer clocks, the more efficient real-time model checking is [9], a smaller number of clocks implies a more compact state space in our setting: 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. However, with no surprise, when no state duplication is met (e.g., Bridge), the computation time is longer. 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]Binaries, sources, models and results are available in [1].

VI. Conclusion and Future Work

We introduced Parametric Stateful Timed CSP, a formalism for reasoning parametrically in hierarchical real-time concurrent systems with shared variables and complex data structures. The adaptation of the inverse method IM for PSTCSP synthesizes a set of parameters around a reference parameter valuation, guaranteeing the same time abstract behavior, and providing the system with a measure of robustness. IM behaves well in practice and, although we showed that parameter synthesis is undecidable for PSTCSP, is given a sufficient termination condition. Our implementation within PSyHCoS leads to efficient parameter synthesis.

As future work, we wish to improve the state space representation, following the lines of the optimization of Section V-C1, and develop further state space reduction techniques. Also, 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 in PSyHCoS.

References

[1] http://www-lipn.univ-paris13.fr/~andre/software/PSyHCoS/.

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

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

Case study	$\|U\|$	*reachAll* $\|S\|$	$\|T\|$	$\|X\|$	t	*reachAll+* $\|S\|$	$\|T\|$	$\|X\|$	t	*IM* $\|S\|$	$\|X\|$	t	*IM+* $\|S\|$	$\|X\|$	t
M_{ex}	2	8	14	2	0.008	8	14	2	0.006	3	2	0.004	3	2	0.005
Bridge	4	-	-	-	M.O.	-	-	-	M.O.	2.8k	2	253	2.8k	2	455
$Fischer_4$	2	-	-	-	M.O.	-	-	-	M.O.	11k	4	41.9	2k	4	8.65
$Fischer_5$	2	-	-	-	M.O.	-	-	-	M.O.	133k	5	1176	13k	5	84.5
$Fischer_6$	2	-	-	-	M.O.	-	-	-	M.O.	-	-	M.O.	86k	6	1144
Jobshop	8	14k	20k	2	21.0	12k	17k	2	18.1	1112	2	17.1	877	2	22.8
RCS_5	4	5.6k	7.2k	4	10.5	5.6k	7.2k	4	9.54	5.6k	4	7.83	5.6k	4	16.7
RCS_6	4	34k	43k	4	91.7	34k	43k	4	54.5	34k	4	60.4	34k	4	91.3
TrAHV	6	7.2k	13k	6	14.2	7.2k	13k	6	15.8	227	6	0.555	227	6	0.655

Table I
APPLICATION OF ALGORITHMS FOR PARAMETER SYNTHESIS USING PSYHCOS

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

[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.

[6] É. André, J. Sun, Y. Liu, and J.-S. Dong. Parameter synthesis for hierarchical concurrent real-time systems (full version). Research report, National University of Singapore, 2012. www-lipn.univ-paris13.fr/~andre/documents/PSTCSP.pdf.

[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.

[8] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Beyond liveness: Efficient parameter synthesis for time bounded liveness. In *FORMATS'05*, pages 81–94, 2005.

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

[10] B. Bérard, A. Petit, V. Diekert, and P. Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36:145–182, 1998.

[11] 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.

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

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

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

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

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

[17] T. A. Henzinger and H. Wong-Toi. Using HYTECH to synthesize control parameters for a steam boiler. In *FMIA'95*, pages 265–282, 1995.

[18] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[19] J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes, data and time. In *IFM'02*, pages 245–266, 2002.

[20] H.-H. Kwak, I. Lee, A. Philippou, J.-Y. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *IEEE RTSS'98*, pages 409–418, 1998.

[21] 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.

[22] B. P. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In *IFM'99*, pages 66–85, 1999.

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

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

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

[26] A. Schrijver. *Theory of linear and integer programming*. John Wiley and Sons, 1986.

[27] J. Sun, Y. Liu, J. 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.

[28] 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.

[29] 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.

[30] 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.