11-2014

Parameter synthesis for hierarchical concurrent real-time systems

Étienne ANDRÉ

Yang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Jin Song DONG

Real-Time Syst (2014) 50:620–679
DOI 10.1007/s11241-014-9208-6

Parameter synthesis for hierarchical concurrent real-time systems

Étienne André · Yang Liu · Jun Sun · Jin-Song Dong

Published online: 12 September 2014

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 thus interesting to reason parametrically, by considering that the delays are parameters (unknown constants) and synthesizing a constraint guaranteeing a correct behavior. We present here Parametric Stateful Timed Communicating Sequential Processes, a language capable of specifying and verifying parametric hierarchical real-time systems with complex data structures. Although we prove that the synthesis is undecidable in general, we present several semi-algorithms for efficient parameter synthesis, which behave well in practice. This work has been implemented in a real-time model checker, PSyHCoS, and validated on a set of case studies.

Keywords Real-time specification · Parametric timed verification · Model checking · Robustness

É. André (✉)
Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS UMR 7030, Villetaneuse, France

Y. Liu
School of Computer Engineering, Nanyang Technological University, Singapore, Singapore

J. Sun
Singapore University of Technology and Design, Singapore, Singapore

J.-S. Dong
School of Computing, National University of Singapore, Singapore, Singapore

1 Introduction

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

1.1 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. In this paper, the notion of correctness will generally refer to the validity of a property, e.g., a linear-time property. Furthermore, we aim at defining a formalism that is intuitive, powerful (with the use of external variables, structures and user defined functions), and that allows efficient parameter synthesis and verification.

1.2 Parameter synthesis for timed concurrent systems

Timed automata (TA) (Alur et al. 1994) are finite state automata equipped with *clocks*. Clocks are real-valued variables uniformly increasing, and compared with constants in guards and invariants (Henzinger et al. 1994). TA have been widely used in the last two decades to verify timed systems, in particular using the UPPAAL model checker (Larsen et al. 1997). The parametric extension of TA (viz., *parametric timed automata*, or *PTA*) allows the use of parameters within guards and invariants (Alur et al. 1993).

The parameter design problem for PTA was formulated in Henzinger and Wong Toi (1995), where a straightforward solution is given, based on the generation of the whole state space. Unfortunately, this is unrealistic in most cases. The HYTECH model checker (Henzinger et al. 1997), one of the first for parametric timed (and more generally 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, which quickly leads to memory overflow.[1] The parameter synthesis problem has then been applied in particular

[1] For example, the PTA model of the SPSMALL memory (Chevallier et al. 2009) is made of 10 PTA in parallel, but only 31 symbolic states are reachable according to the semantics of Alur et al. (1993). Due to the static composition of PTA, HYTECH crashes by memory overflow even *before* starting the actual

to communication protocols (e.g., Bounded Retransmission protocol (D'Argenio et al. 1997) or Root Contention protocol (Collomb Annichini and Sighireanu 2001) using TREX (Annichini et al. 2001)) and asynchronous circuits (e.g., Yoneda et al. 2002; Clarisó and Cortadella 2007). Efficient optimizations and data structures were developed for timed automata, such as Difference Bound Matrices (DBMs); unfortunately, most of them do not apply to the parametric setting, or to only partially parameterized systems (e.g., Behrmann et al. 2005, where a non-parametric model is verified against a parameterized formula), or are much less efficient than their non-parametric counterpart (e.g., parameterized DBMs (Hune et al. 2002)). In André et al. (2009), André and Soulat (2013), the *inverse method* synthesizes constraints for fully parameterized systems modeled using PTA. Different from CEGAR-based methods (Clarke et al. 2000), this semi-algorithm[2] is based on a "good" parameter valuation, and synthesizes a constraint to guarantee the same time abstract behavior as for the reference parameter valuation, and thus to quantify the robustness of the system. As an interesting consequence, the preservation of the time-abstract behavior guarantees the preservation of linear time properties (expressed, e.g., in LTL). The authors of Knapik and Penczek (2012) 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 PTA. This is done by applying bounded model checking techniques to PTA. Semi-algorithms have been proposed in Traonouez et al. (2009) for synthesizing parameters for time Petri nets with stopwatches, and implemented in Roméo Lime et al. (2009). Different from our setting, the constraint satisfies a formula expressed using a non-recursive subset of parametric TCTL; furthermore, their implementation does not support user defined data structures.

Most problems for parametric timed formalisms are undecidable, including the emptiness problem (that is, the existence of at least one parameter valuation implying the reachability of a discrete state). However, this problem is known to be decidable (actually PSPACE-complete) for L/U PTA, that is a subclass of PTA (Hune et al. 2002) in which each parameter can be used either as an upped bound, or as a lower bound, but not both. However, the implementation (based on parametric DBMs) proposed in Hune et al. (2002) may not terminate. Similarly, the emptiness problem for the corresponding subclass of parametric time Petri nets, called L/U parametric time Petri nets (Traonouez et al. 2009), is also decidable. Further problems (universality and finiteness of the valuation set for infinite runs acceptance properties) have been shown to be decidable for L/U PTA (Bozzelli and La Torre 2009). Most other problems for L/U PTA are undecidable, even for even more restricted subclasses such as L-PTA or U-PTA (Jovanovic et al. 2013). The case of the synthesis of bounded integers (which is trivially decidable, since it suffices to enumerate all possible parameter valuations) has also been considered in Jovanovic et al. (2013): it is shown that the problem of the existence of parameter valuations such that a TCTL property is satisfied is PSPACE-complete, and can be performed efficiently using symbolic techniques.

exploration; in contrast, IMITATOR (André et al. 2012a) finishes the analysis within 0.079 s while using only 3.1 MiB of memory. Details are available in http://www.lsv.ens-cachan.fr/Software/imitator/hytech/.

[2] A semi-algorithm is a procedure that may not terminate but, if it does, then its result is correct.

In Kwak et al. (1998, 1999), 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. These works are closer to our approach, in the sense that they synthesize timing parameters in a process algebra; however, they are dedicated to scheduling problems only, whereas our approach is general.

Although it does not strictly treat the parameter synthesis, the AASAP (Almost As Soon As Possible) semantics by Raskin et al. is a semantics that considers a parameter Δ corresponding to the reaction time of the controller. Hence, this semantics discards infinitely fast behaviors, that are not realistic in practice. Its most interesting property is that, once the system has been proved correct for a given Δ, any implementation using a faster controller (i.e., with a smaller Δ) will be correct too. More generally, the robustness problem consists in studying the influence of infinitesimally small variations of the timing requirements or the clocks speed on the system correctness; many such problems are decidable for (subclasses of) timed automata or time Petri nets (see, e.g., Bouyer et al. 2011; Markey 2011; Jaubert and Reynier 2011; Akshay et al. 2012; Bouyer et al. 2012, 2013; Sankur and Shrinktech 2013). Parameter synthesis techniques have also been used to solve robustness problems (e.g., Traonouez 2012). The inverse method (that we extend to PSTCSP in Sect. 5.3 to show the applicability of our formalism) can also be used to perform robustness analyses, as shown in the setting of parametric time Petri nets with stopwatches (André et al. 2013c).

1.3 Stateful Timed CSP

Communicating Sequential Processes (CSP) (Hoare 1985) is a powerful event-based formalism for describing patterns of interactions in concurrent systems. Timed CSP (see, e.g., Schneider 2000) extends CSP with timed constructs for modeling real-time systems. Stateful Timed CSP (STCSP) (Sun et al. 2013) further extends Timed CSP with more timed constructs and shared variables in the spirit of CSP♯ (Sun et al. 2009a) in order to specify hierarchical complex real-time systems. Through dynamic zone abstraction, *finite-state* zone graphs can be generated automatically from STCSP models, which are subject to model checking. Stateful Timed CSP offers an intuitive way of modeling hierarchical systems, with a textual representation and more flexible recursive definitions. An advantage of Timed CSP over TA is the lower number of clocks necessary to verify the systems (Sun et al. 2013), because, unlike TA, clocks are *implicit* in STCSP, and are only activated when necessary. STCSP is implemented into the PAT model checker (Sun et al. 2009b).

1.4 Contribution

Our first contribution is to introduce Parametric Stateful Timed CSP (PSTCSP). 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. PSTCSP shares similar design principles with integrated specification languages like Timed Communicating Object Z (TCOZ) (Mahony and

Dong 1999) and CSP-OZ-DC (Hoenicke and Olderog 2002). The main idea is to support shared variables and manipulation of global variables (sequential termination programs) using imperative programing languages. The result is a highly expressive modeling language that can be automatically analyzed by tools. Although we show that the expressiveness of PSTCSP is close to the one of PTA, there are key differences of PSTCSP with PTA. First, clocks are implicit in PSTCSP, thus avoiding errors when manually writing constraints using clocks and parameters. Second, hierarchy is a native feature of PSTCSP, allowing the designer to develop the system using nested components. Many systems can be designed more intuitively using hierarchy, and it may allow one to handle refinement as well as closed ("black box" or "gray box") systems. Third, user defined variables, data structures and functions can be defined and used in PSTCSP processes, thus making the specification and verification of real-time systems intuitive.

Although we show that the emptiness problem is undecidable for PSTCSP, our second contribution is to develop and compare three semi-algorithms for parameter synthesis. The first one, computing all reachable states, allows the application of finite state timed model checking techniques defined in Sun et al. (2013), but may not terminate. From the set of reachable states, one can also perform parameter synthesis, and we give as an example an algorithm that synthesizes all parameter valuations such that a given process (or a given variable valuation) is reachable. The second one is an algorithm useful for defining good sets of values for the timing parameters in problems such as schedulability problems. In the third one, we extend the inverse method (André et al. 2009; André and Soulat 2013) 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 requirements are parametric.

Our third contribution is to implement the proposed techniques in a model checker named PSyHCoS to support both an intuitive modeling facility using a graphical interface, and efficient algorithms for verification and parameter synthesis.

This paper is an extended version of André et al. (2012b). In addition to the results of André et al. (2012b), this paper contains all proofs of the theoretical results, refines several results (in particular makes more clear the notion of emptiness for PSTCSP), and contains detailed examples. Furthermore, we prove the semantic equivalence between some syntactic constructs of PSTCSP. We also introduce a new synthesis algorithm dedicated to problems such as schedulability problems. Finally, the inverse method for PSTCSP is fully characterized (correctness, confluence, completeness and termination).

1.5 Plan of the paper

We recall preliminary notions in Sect. 2. We introduce PSTCSP in Sect. 3 and study its expressiveness and decidability questions in Sect. 4. We introduce algorithms for parameter synthesis in Sect. 5, and apply them to case studies in Sect. 6 using our implementation PSyHCoS. We give future directions of research in Sect. 7.

2 Preliminaries

2.1 Finite-domain variables

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

2.2 Constraints on clocks and parameters

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

We also assume a fixed set $\mathcal{U}$ of *parameters* (i.e., unknown constants) disjoint from $\mathcal{V}ar$ and $\mathcal{X}$. Given a finite set $U = \{u_1, \ldots, u_M\} \subset \mathcal{U}$, a *parameter valuation* is a function $\pi : U \to \mathbb{R}_+$ assigning a non-negative real[3] value to each parameter. There is a one-to-one correspondence between valuations and points in $(\mathbb{R}_+)^M$. We will often identify a valuation 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 linear terms of the form $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ with $z_i \in X \cup U$, $\alpha_i \in \mathbb{R}_+$ for $1 \leq i \leq N$, and $d \in \mathbb{R}_+$. We define similarly inequalities over X (resp. U). A constraint is a conjunction of inequalities. We denote by $\mathcal{K}_X$, $\mathcal{K}_U$ and $\mathcal{K}_{X \cup U}$ the sets of all constraints over X, over U, and over X and U, respectively. 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 each clock x in D 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 $C_1 \subseteq C_2$ if $\forall w, \pi :< w, \pi > \models C_1 \Rightarrow < w, \pi > \models C_2$. We write $C_1 = C_2$ if $C_1 \subseteq C_2$ and $C_2 \subseteq C_1$.

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.

[3] In the literature related to parametric timed systems, constants are either in the real or the rational domain. Here, to maintain consistency with STCSP (Sun et al. 2013), where constants are defined in $\mathbb{R}_+$, we choose reals.

Given a subset of clocks $X' \subseteq X$, we denote by $C{\downarrow_{X'\cup U}}$ the constraint obtained from C after elimination of the clocks not in X', i.e., by projecting C onto $X' \cup U$. This is obtained using variable elimination techniques such as Fourier-Motzkin elimination (Schrijver 1986). Formally, $C{\downarrow_{X'\cup U}} = \{< w, \pi > | \ w : X' \to \mathbb{R}_+ \wedge \pi : U \to \mathbb{R}_+ \wedge < w, \pi > \models C\}$. In particular, we denote by $C{\downarrow_U}$ the constraint over U obtained by projecting C onto the parameters, i.e., the constraint obtained from C after elimination of all clocks. Formally, $C{\downarrow_U} = \{\pi \mid \pi \models C\}$. (Note that, by expanding the definition of $\pi \models C$, we have that $C{\downarrow_U} = \{\pi \mid \exists w : X \to \mathbb{R}_+ \text{ s.t. } < w, \pi > \models C\}$.)

Sometimes we will refer to a variable domain X', which is obtained by renaming the variables in X. Explicit renaming of variables is denoted by the substitution operation. Here, $C_{[X \leftarrow X']}$ denotes the constraint obtained by replacing in C the variables of X with the corresponding variables of X'.

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. Formally:

$$C^{\uparrow} = \Big((C \wedge X' = X + d){\downarrow_{X'\cup U}}\Big)_{[X' \leftarrow X]}$$

where d is a new parameter with values in $\mathbb{R}_+$, and X' is a renamed set of clocks. The inner part of the expression adds a delay d to all clocks; the projection onto $X' \cup U$ eliminates the original set of clocks X, as well as the variable d; the outer part of the expression renames clocks X' with X.

We show below two simple results that will be useful in the proofs in Sect.4.

Lemma 1 *Let $C, C_1, C_2 \in \mathcal{K}_{X\cup U}$. If $C_1{\downarrow_U} \subseteq C{\downarrow_U}$, then $(C_1 \wedge C_2){\downarrow_U} \subseteq C{\downarrow_U}$.*

Proof By definition of the projection, $C{\downarrow_U} = \{\pi \mid \pi \models C\}$. $C_1{\downarrow_U} \subseteq C{\downarrow_U}$ implies that $\{\pi \mid \pi \models C_1\} \subseteq \{\pi \mid \pi \models C\}$. Since $C_1 \wedge C_2 \subseteq C_1$, then $\{\pi \mid \pi \models C_1 \wedge C_2\} \subseteq \{\pi \mid \pi \models C_1\}$. Hence $\{\pi \mid \pi \models C_1 \wedge C_2\} \subseteq \{\pi \mid \pi \models C\}$. □

Lemma 2 *Let $C \in \mathcal{K}_{X\cup U}$. Then $(C^{\uparrow}){\downarrow_U} = C{\downarrow_U}$.*

Proof From its definition, time elapsing adds new clock constraints ($X' = X + d$), removes the clocks X, and renames X' with X; all these operations keep the projection onto the parameters unchanged. □

2.3 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 convention is adopted: $e \in \Sigma$ denotes an observable event; $a \in \Sigma_\tau$ denotes an observable event or τ; $E \subseteq \Sigma$ denotes a set of observable events.

2.4 Labeled transition systems

Labeled transition systems will be used later to define the semantics of PSTCSP.

Fig. 1 Syntax of PSTCSP processes

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

Definition 1 A *labeled transition system (LTS)* is a tuple $\mathcal{L} = (S, s_0, Symb, \Rightarrow)$ where S is a set of *states*, $s_0 \in S$ is the *initial state*, $Symb$ is a *set of symbols*, and $\Rightarrow : S \times Symb \times S$ 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 $a_i \in Symb$ in the form of $s_0 \overset{a_0}{\Rightarrow} s_1 \overset{a_1}{\Rightarrow} s_2 \cdots$. A state s_i is *reachable* if it belongs to some run r. We denote by $Runs(\mathcal{L})$ the set of runs of $\mathcal{L}$.

A run is said to be *maximal* if either it is infinite, or it is finite and its last state has no successor.

3 Syntax and semantics of PSTCSP

3.1 Syntax

PSTCSP models the control logic of the system using a rich set of process constructs. A process P is defined by the grammar in Fig. 1, where $u \in U$.[4] Processes marked with * are parametric timed processes; they allow the use of parameters instead of timing constants as in STCSP. $\mathcal{P}$ denotes the set of all possible processes. Note that this grammar allows recursions (and hence cyclic behaviors), since a given process P can refer to itself; for example, the event prefixing rule allows one to define $P \doteq e \rightarrow P$, which may lead to an infinite number of e events.

Definition 2 A Parametric Stateful Timed CSP (or PSTCSP) model is a tuple $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ where $Var \subset \mathcal{V}ar$, $U \subset \mathcal{U}$, V_0 is the initial variable valuation, $P_0 \in \mathcal{P}$ 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

[4] An alternative could be $u \in (U \cup \mathbb{R}_+)$. Our implementation actually allows the definition of either constants or parameters in the timed constructs, but defining $u \in U$ simplifies the subsequent reasoning and proofs.

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 definitions of processes. Each component may have internal hierarchies, and allow for abstraction and refinement: a subprocess may be replaced with 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.

3.1.1 Valuation of a model

Given a PSTCSP model $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ and a parameter valuation $\pi = (\pi_1, \ldots, \pi_M)$, $\mathsf{M}[\pi]$ denotes the *valuation* of M with π, viz., the model $(Var, U, V_0, P_0[\pi], K)$, where $P_0[\pi]$ denotes process P_0 where all occurrences of a parameter u_i were replaced by constant π_i in the timed constructs, and K is $K_0 \wedge \bigwedge_{i=1}^{M}(u_i = \pi_i)$. We say that M is *valuated* with π. This corresponds to the PSTCSP model obtained from M by substituting every occurrence of a parameter u_i with constant π_i in the timed constructs. Note that $\mathsf{M}[\pi]$ is a (non-parametric) STCSP model.

3.1.2 Additional notation

In the following, given a process P, we use $\Sigma(P)$ to denote the alphabet of process P. Hence, $\Sigma(P)$ includes all visible events occurring in P and its subprocesses, including $\checkmark$.

3.1.3 Recursivity

We define below the notion of recursive models (i.e., cyclic dependencies between processes).

Definition 3 (*Recursive model*) A PSTCSP model is *recursive* if at least one process is referred to in one of its subprocesses.

For example, the model of Fischer's mutual exclusion protocol introduced in Sect. 3.3 is recursive, since $Active(i)$ is a subprocess of $proc(i)$, and $proc(i)$ refers to $Active(i)$.

3.2 Informal semantics

3.2.1 Untimed constructs

We first briefly describe the untimed constructs, which are identical to the ones in STCSP.

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 operations on complex data structures, sequential programs may be attached to events. By complex data structures, we mean any user-defined finite-domain structure, such as tuples, lists, but also more complex structures such as hashtables, or combinations of structures, such as a pair made of a string hashtable and a list of bounded integers. Process $a\{program\} \rightarrow P$ executes 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 $v_1, v_2 \in Var$) or a more complicated sequential program. Our implementation (see Sect. 6) supports an imperative language (similar to C, C♯ and Java) to be used inside *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 $P \,\Box\, Q$ offers an external choice between P and Q. As in CSP, the first observable event determines which of P and Q is executed. That is, $P \,\Box\, Q$ is resolved by the first observable event occurring in either P (in which case the process continues with P) or Q (in which case the process continues with Q).[5]

Process $P; Q$ behaves as P until P terminates and then behaves as Q immediately.

$P \setminus E$ hides occurrences of events in E.

The parallel composition of two processes is written as $P \,[\![E]\!]\, Q$, where P and Q may communicate via multi-party event synchronization (following CSP rules Hoare 1985) or shared variables; P and Q synchronize only on events belonging to the specified set E of events. For example, $(a \rightarrow \texttt{Skip}) \,[\![\{a\}]\!]\, (a \rightarrow \texttt{Skip})$ will result in a synchronization on the event a, whereas $(a \rightarrow \texttt{Skip}) \,[\![\{\}]\!]\, (a \rightarrow \texttt{Skip})$ will result in an interleaving between both a events.

3.2.2 Timed constructs

We now explain the parametric timed constructs, where parameter u is an unknown (constant) non-negative real number.

- Process `Wait`$[u]$ idles for u time units.
- In process P `timeout`$[u]$ Q, the *first* observable event of P shall occur no later than u time units. Otherwise, Q takes over the 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.

[5] For simplicity, in the following, we leave out general and internal choices from the classic CSP (Hoare 1985). The terminology is a little ambiguous in the literature: We assume that a general choice $(P \mid Q)$ can be resolved by an occurrence of any event; an external choice $(P \,\Box\, Q)$ can be resolved only by visible events (not τ); and an internal choice $(P \diamond Q)$ is resolved "immediately", hence cannot be delayed (which generates a τ-transition). Although we only consider external choice in the following, all three constructions implemented in PSyHCoS, and used in our case studies.

– Process P `within`$[u]$ must react within 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.

3.2.3 Discussing the notion of 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 Fidge et al. (1999), 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 Qin et al. (2003), the authors use Unifying Theory of Programming in order to formalize the semantics of TCOZ. As in Fidge et al. (1999), 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.

In contrast to Qin et al. (2003), Fidge et al. (1999), we choose here to keep a semantics similar to the one of STCSP (Sun et al. 2013), and we 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 (in that case, time elapsing may be stopped too). Also observe that the `within` construct in (P)STCSP has a similar effect in our setting.

3.2.4 Syntactic sugar

Urgent event prefixing (Davies 1993), written as $e \twoheadrightarrow P$, is defined as $(e \rightarrow P)$ `within`$[0]$, i.e., e must occur immediately, that is within 0 units of time. Furthermore, we use $P \parallel Q$ for $P \llbracket \Sigma(P) \cap \Sigma(Q) \rrbracket Q$.

3.3 Example: Fischer mutual exclusion

We introduce below a simple example[6] to show that PSTCSP is expressive enough to capture hierarchical concurrent real-time models. Although this example is compositional, which reflects the nature of hierarchical systems, it does not feature multiple hierarchy due to space constraint. Complex STCSP system models have been presented (Sun et al. 2013), and all the STCSP examples can be parameterized to obtain a PSTCSP model.

Example 1 Fischer's mutual exclusion algorithm is modeled as a PSTCSP model $(Var, U, V_0, FME, True)$, where $U = \{\delta, \gamma\}$, and $Var = \{turn, cnt\}$. The *turn* variable indicates which process attempted to access the critical section most recently. The

[6] This example is a parametrization of the example from Sun et al. (2013, p. 3:5).

cnt variable counts the number of processes accessing the critical section. The initial valuation V_0 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.

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

where n is a constant representing the number of processes.

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

This model is hierarchical, due to the nested definition of the processes; for example, each process $proc(i)$ is defined using the nested process $Active(i)$. (Actually, this is a recursive definition since $Active(i)$ is itself defined using $proc(i)$.) This model is concurrent because the n processes are executed in parallel. And it is real-time since concurrent timing constraints ($\texttt{Wait}[\gamma]$ and $\texttt{within}[\delta]$) for each process may occur in parallel.

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 Sect. 6.3.

3.4 Formal semantics

In the following, we introduce the formal semantics for PSTCSP in terms of states containing a variable valuation, a process, and a constraint over X and U.

Definition 4 Let $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ be a PSTCSP model. 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 $C \in \mathcal{K}_{X \cup U}$.

3.4.1 Clock activation

As in STCSP, clocks in PSTCSP are *implicitly* associated with timed processes – which is different from PTA. 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. Before defining the semantics, we need to associate clocks

with timed 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 $\texttt{Wait}[u_1]$ (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 start at the same time because the execution of `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 Sun et al. (2013) 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 $\texttt{Wait}[u]_x$ (and, similarly, $P\ \texttt{timeout}[u]_x\ Q$, $P\ \texttt{interrupt}[u]_x\ Q$, $P\ \texttt{within}[u]_x$, $P\ \texttt{deadline}[u]_x$) 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 definitions of $Act(P, x)$. Rules A1 to A5 state that if a process is untimed and none of its subprocesses is activated, then it is unchanged. Rules A6 to A10 state that if the process is timed, then it is associated with x. If a timed process has already been associated with a clock y, then it will not be associated

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

Fig. 2 Clock activation function for PSTCSP

with the new clock. This is captured by rules A11 to A15, where $\texttt{Wait}[u]_y$ denotes that $\texttt{Wait}[u]$ is associated with clock y. 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. For instance, the set of active clocks associated with $P\ \texttt{timeout}[u]_x\ Q$ contains x and the clocks associated with P. Notice that there is no clock associated with Q because it has not yet been activated.

Example 2 In the Fischer's mutual exclusion example, assume that there are three processes. The first and second processes have evaluated the condition "$\texttt{if}\ (turn = -1)$" and become $Active(0)$ and $Active(1)$ respectively, 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:

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

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

3.4.2 Idling function

We adapt in the following the function *idle* (initially defined for STCSP in Sun et al. (2013)) which, given a process in $\mathcal{P}_{act}$, calculates a constraint expressing how long the process can idle. Here, 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 an 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)

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

Fig. 3 Idling calculation

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 $P \,\Box\, Q$ (or $P \,[\![E]\!]\, Q$) 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.

3.4.3 Operational semantics

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

Definition 5 Let $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ be a PSTCSP model. The *semantics of* M, denoted by $\mathcal{L}_{\mathsf{M}}$, is an LTS $(S, s_0, \Rightarrow, \Sigma_\tau)$ where

$$S = \{(V, P, C) \in \mathcal{V}(Var) \times \mathcal{P} \times \mathcal{K}_{X \cup U}\},$$
$$s_0 = (V_0, P_0, K_0)$$

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 $(V, Act(P, x), C \wedge x = 0) \overset{a}{\rightsquigarrow} (V', P', C')$, where C' is satisfiable, then we have: $\big((V, P, C), a, (V', P', C'{\downarrow}_{cl(P') \cup U})\big) \in \Rightarrow$.

We say that a given variable valuation V is *reachable* in M if there exists a state (V, P, C), for some P and some C, that is reachable in $\mathcal{L}_{\mathsf{M}}$. We also say that a run of M *passes by* V. Similarly, we say that a given process P is reachable in M if there exists a state (V, P, C), for some V and some C, that is reachable in $\mathcal{L}_{\mathsf{M}}$.

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

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

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

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, giving $(V, P\ \texttt{timeout}[u]_x\ Q, C)$, 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.

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

Similarly, rule $ato2$ ensures that the occurrence of an observable event e from process P may occur only if $x \leq u$, i.e., before timeout occurs.

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

Rule $ato3$ states that timeout results in a τ-transition when x is exactly equal to u. The constraint $x = u \wedge idle(P)$ ensures that process P may idle all the way until timeout occurs.

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

Let us explain Definition 5 further. First, given a state (V, P, C), a clock x which is not currently associated with P is picked. Then, the state (V, P, C) is transformed into $(V, Act(P, x), C \wedge x = 0)$, i.e., timed processes which just became 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'. Notice that one clock may be introduced and zero or more clocks may be pruned during a transition.

Example 3 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.

3.5 Traces

We now introduce the notion of *trace*, that abstracts part of a system's behavior. In the literature (see, e.g., Baier and Katoen 2008), the approaches considered are either state-based (which corresponds here to a sequence of processes) or event-based (which corresponds here to a sequence of events). As a matter of consistency with previous works (André et al. 2009), we consider here a combined state- and event-based approach: That is, we define a trace as an alternating sequence of processes and events.

The following definition introduces traces for parametric PSTCSP models. Note that, since an STCSP model is a simplified case of a PSTCSP model, this definition can also be used for non-parametric models.

Definition 6 (*Trace*) Given a PSTCSP model M and a run r of $\mathcal{L}_{\mathsf{M}}$ of the form $(V_0, P_0, C_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (V_m, P_m, C_m)$, the *trace associated with* r is the alternating sequence of processes with variables and events $(V_0, P_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (V_m, P_m)$. The *trace set* of M is the set of all traces associated with the runs of M.

Traces abstract away the constraint C; hence, all the continuous information (values of the clocks and of the parameters) is abstracted away.

4 General results for PSTCSP

In this section, we first define a subset of PSTCSP, called regular PSTCSP, so as to maintain the consistency with regular STCSP (Sect. 4.1). We then show that all the parametric timed constructs can be defined using two of them only (Sect. 4.2). We then characterize the expressiveness of regular PSTCSP (Sect. 4.3), study the (un)decidability of the membership and emptiness problems (Sect. 4.4), and prove results relating parametric runs and non-parametric runs (Sect. 4.5).

4.1 Regular PSTCSP

The results stated in Sun et al. (2013) are valid for a subset of STCSP called *regular* STCSP. "A Stateful Timed CSP model is regular if a process expression is constituted by finitely many process constructs, for every reachable configuration." (Sun et al. 2013) In other words, the discrete part of the reachable states is finite. This can be seen as equivalent to the finite number of locations in timed automata, or the finite number of places together with their boundedness in time Petri nets.

We define regular PSTCSP the same way: a PSTCSP model M is regular if P is a process expression constituted by finitely many process constructs, for every reachable configuration (V, P, C). Now, since the state space (even symbolic) is infinite for PSTCSP, we do not need this restriction in our work. However, we will consider regular PSTCSP when comparing to STCSP, i.e., when considering the expressiveness (Sect. 4.3) and some of the decidability problems (Sect. 4.4).

4.2 Equivalence of timed constructs

We show here that all timed constructs in the syntax defined in Fig. 1 can actually be defined using only two timed constructs: `Wait` and `deadline`.

As in Davies (1993), the `timeout` construct can be defined using the `Wait` construct as follows, where e_{to} is a fresh event.

$$P \texttt{ timeout}[u]\ Q = \left(P \,\Box\, (\texttt{Wait}[u];\ e_{to} \twoheadrightarrow Q)\right) \setminus \{e_{to}\}$$

The `interrupt` construct can be defined using `Wait` as follows.

$$P \texttt{ interrupt}[u]\ Q = \big((P \parallel R)\ [\![e_{int}]\!]\ (\texttt{Wait}[u];\ e_{int} \twoheadrightarrow Q)\big) \setminus \{e_{int}\}$$

with $R \doteq \big(\ \Box_{\ e \in \Sigma(P)}(e \rightarrow R)\big) \ \Box\ (e_{int} \rightarrow \texttt{Skip})$, where e_{int} is a fresh event. Intuitively, R synchronizes on any observable event of P as many times as necessary, until event e_{int} occurs, in which case it derives to `Skip`. From the right part of the expression, event e_{int} occurs immediately after `Wait`$[u]$ has completed, i.e., after u units of time. Then Q takes over. After u units of time, any observable event of P is blocked due to the fact that R is now `Skip`, and cannot synchronize with P anymore. Note that we use $\Box_{\ e \in \Sigma(P)}(e \rightarrow R)$ to denote the process $(e_1 \rightarrow R) \ \Box\ \ldots\ \Box\ (e_n \rightarrow R)$, assuming that $\Sigma(P) = \{e_1, \ldots, e_n\}$.

The `within` construct can be defined using the `deadline` construct: considering P `within`$[u]$, this can be achieved by executing P in parallel with Q `deadline`$[u]$; R, with Q a process synchronizing once on any observable event with P, and R a process synchronizing, possibly several times, on any observable event with P. Formally:

$$P \texttt{ within}[u] = P \parallel (Q \texttt{ deadline}[u];\ R)$$

with $Q \doteq \Box_{\ e \in \Sigma(P)}(e \rightarrow \texttt{Skip})$ and $R \doteq (Q;\ R)$.

Although we showed that some timed constructs are equivalent, since one of the advantages of PSTCSP is its conciseness and convenient, user-friendly syntax, we keep these "syntactic sugar" constructs in our language. Furthermore, this maintains consistency with STCSP (for which these constructs are also redundant, but this had not been studied when STCSP was first defined).

4.3 Expressiveness

We consider here the expressive power of regular PSTCSP, i.e., the set of timed words for any parameter valuation. Timed words are alternating sequences of events and real-valued timing delays, and usually characterize the language of a formalism to model real-time systems such as timed automata or time(d) Petri nets. We will show in the following that the expressive power of regular PSTCSP is equal to parametric closed timed automata with ϵ-transitions.

Let us first recall that regular STCSP is equivalent to that of closed timed automata with ϵ-transitions. Recall from Ouaknine and Worrell (2003b) that closed timed automata with ϵ-transitions are timed safety automata (Henzinger et al. 1994) (i.e., timed automata Alur et al. 1994) without acceptance conditions and with location invariants) augmented with ϵ-transitions (Bérard et al. 1998), and with the restriction of 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 (Ouaknine and Worrell 2003a; 2003b; Asarin et al. 1998).

The following result comes from Sun et al. (2013). Although it is not made explicit in Sun et al. (2013), the expressive power is understood in this result in terms of timed words, that is alternating sequences of events and real-valued timing durations.

Lemma 3 (Sun et al. 2013, Section 4.4) *The expressive power of regular Stateful Timed CSP is equal to that of closed timed automata with ϵ-transitions.*

We define parametric closed timed automata with ϵ-transitions as a parametric extension of closed timed automata with ϵ-transitions, following the parameterization of TA into PTA (Alur et al. 1993), i.e., by allowing parameters within guards and invariants. We consider here the expressive power as the set of timed words for all possible parameter valuations.

Proposition 1 *The expressive power of regular Parametric Stateful Timed CSP is equal to that of parametric closed timed automata with ϵ-transitions.*

Proof From the fact that both parametric formalisms are obtained exactly the same way: regular Parametric Stateful Timed CSP is obtained from regular Stateful Timed CSP by allowing the use of parameters in place of any constant in the system, and similarly for parametric closed timed automata with ϵ-transitions. □

Since closed timed automata with ϵ-transitions are a (strict) subclass of timed automata with ϵ-transitions (Bérard et al. 1996; Alur and Madhusudan 2004), then parametric closed timed automata with ϵ-transitions are a subclass of parametric timed automata with ϵ-transitions. From Proposition 1 and the fact that timed automata with ϵ-transitions are incomparable with standard TA (Bérard et al. 1996; Alur and Madhusudan 2004), we can infer that regular PSTCSP is less expressive than parametric timed automata with ϵ-transitions, but incomparable with standard PTA.

Remark 1 (Definition of expressiveness) We considered here a (common) definition of expressiveness that considers the *union* of all timed words for all parameter valuations. An alternative definition could be to consider the *intersection* of all timed words for all parameter valuations. In that case, the result of Proposition 1 holds too, following the same reasoning.

We believe that PSTCSP is an interesting formalism because one can make use of user-defined data structures, and hierarchical composition is supported in PSTCSP, which is missing in PTA. Furthermore, high level real-time system requirements often state 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`$[u]$). We believe that PSTCSP is well-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, to express high level real-time requirements in PTA, one often needs to manually cast those timing patterns into a set of clock variables explicitly and to carefully design constraints. This is error-prone, in the sense that there is a risk to forget clocks, or to write ill-formed clock constraints. Now, it may be useful to provide (parametric) timed automata with timing requirement patterns to simplify the

modeling (see, e.g., Dong et al. 2008); however, PSTCSP features these patterns in a native manner.

Furthermore, for hierarchical systems, PSTCSP offers a rich set of system composition functions which are inspired by CSP and Timed CSP; as a result, it may be easier to model a system where components form a hierarchy of more than 2 levels than using parametric timed automata. Although tools exist for specifying hierarchy or some data structures for (non-parametric) TA, such as UPPAAL (Larsen et al. 1997), PSTCSP is, to the best of our knowledge, the first parametric real-time formalism combining hierarchical aspects, shared variables and complex data structures in a single formalism based on an intuitive syntax.

Remark 2 (Expressiveness of the data structures) It has long been known (see for example Hoare 1985; Roscoe 2001; Sun et al. 2013) that it is possible to model a finite domain variable (and hence any complex finite-domain data structure) as a finite-state process in parallel to the one that uses it. As a consequence, the use of data structures does not increase the expressiveness of regular PSTCSP. Hence, it would have been possible to define the semantics of PSTCSP without these data structures. Similarly, (parametric) timed automata (Alur et al. 1994; 1993) are usually extended with (bounded) integers (or with more complex data structures in tools such as UPPAAL (Larsen et al. 1997)), although their theoretical definition rarely includes them. However, we believe that this is not satisfactory for two reasons.

First, since PSTCSP is a parametric extension of STCSP (that defines variables in its semantics), we make the same choice to include the variables in our semantics. Second, some algorithms may need to explicitly access the value of data structures (this is the case of our algorithm *3VPsynthesis* introduced in Sect. 5.2). Hence, in contrast to the literature on timed automata, where each paper may extend the original formalism for its own needs (bounded integers, broadcast communication, use of lists, etc.), we provide here a unified semantics, that we hope to be rich enough for defining further algorithms making use of variables and data structures.

4.4 Membership and emptiness problems

We show in this section that parameter synthesis is undecidable in general for regular PSTCSP.

We consider the following problems, defined in Alur et al. (1993), Jovanovic et al. (2013) for parametric timed automata, and adapted here to our setting:[7]

1. *EF-membership problem*: Given M a PSTCSP model, V a variable valuation and π a parameter valuation, is V reachable in $\mathsf{M}[\pi]$?

[7] In parametric timed automata (Jovanovic et al. 2013), the notion of reachable state is based on *locations*, viz., discrete control states. In PSTCSP, there are no such discrete control states; hence, we could define reachability based on a variable valuation, on a given process, or a combination of both. We choose here the first option to simplify the proof, but our results extend directly to the two other cases (see Remark 3).

2. *EF-emptiness problem*: Given M a PSTCSP model and V a variable valuation, does there exist a parameter valuation π such that V is reachable in $\mathsf{M}[\pi]$?
3. *AF-membership problem*: Given M a PSTCSP model, V a variable valuation and π a parameter valuation, do all maximal runs of $\mathsf{M}[\pi]$ pass by V?
4. *AF-emptiness problem*: Given M a PSTCSP model and V a variable valuation, does there exist a parameter valuation π such that all maximal runs of $\mathsf{M}[\pi]$ pass by V?

The two former problems refer to reachability whereas the two latter refer to unavoidability.

Membership Both membership problems (1 and 3) are obviously decidable for regular PSTCSP: it suffices to consider the non-parametric regular STCSP model $\mathsf{M}[\pi]$ and solve this problem using techniques developed in Sun et al. (2013), e.g., by building the set of all reachable states, which is finite.

Proposition 2 (Decidability of membership) *The EF-membership and AF-membership problems are decidable for regular PSTCSP.*

EF-Emptiness. We now show that Problem 2 is undecidable. We first consider the case of general PSTCSP.

Theorem 1 (Undecidability of EF-emptiness) *The EF-emptiness problem is undecidable for PSTCSP.*

Proof We reduce the halting problem for 2-counter machines (known to be undecidable Minsky 1967) to the problem of testing if, given M a PSTCSP process and V a variable valuation, there exists a parameter valuation π such that V is reachable in $\mathsf{M}[\pi]$.

As in Alur et al. (1993), 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 $\{l_1, \ldots, l_n\}$. 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 there exists a parameter valuation π such that V is reachable in $\mathsf{M}_{CM}[\pi]$ if and only if *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).

In the following, we use a similar reduction to that of Alur et al. (1993), and adapt it to PSTCSP. Let us first recall the construction used in Alur et al. (1993). That construction constructs a PTA using three clocks x, y, z and six parameters $a, a_{-1}, a_{+1}, b, b_{-1}, b_{+1}$ such that $a = a_{-1} + 1 = a_{+1} - 1$ and $b = b_{-1} + 1 = b_{+1} - 1$. 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. Hence, the value of C_1 is encoded by $b - y$, and the value of C_2 by $b - a - z$. Each control variable is encoded using a dedicated PTA location. Then, for each instruction, a sequence of locations and transitions is added; for example, the instruction "if $l = l_i$ then $C_1 := C_1 + 1$ and $l := l_{i'}$" is modeled in Alur

et al. (1993) by adding a path to the PTA modeling CM, using the scheme recalled in Fig. 4. Indeed, if the initial configuration is $(l_i, b-y, b-a-z)$ then, after this sequence of transitions, the configuration when reaching l_i' is $(l_i, b-y-1, b-a-z)$, correctly encoding the fact that the location changed from l_i to l_i' and that C_1 was incremented by 1. The instructions of the form "if $l = l_i$ then $C_1 := C_1 - 1$ and $l := l_{i'}$" and "if $l = l_i$ and $C_1 = 0$ then $l := l_{i'}$" are encoded in a similar manner (and similarly for C_2).

Our encoding is similar: we also use 6 parameters $a, a_{-1}, a_{+1}, b, b_{-1}, b_{+1}$, and we define processes for each different control variable l_i. The main difficulty when adapting the proof of Alur et al. (1993) to PSTCSP is the fact that clocks are now implicit. We hence encode the 3 clocks x, y and z of Alur et al. (1993) using three *processes* X, Y and Z, respectively, running in parallel and that ensure that the elapsing of time conforms to the value of clocks in the proof of Alur et al. (1993). We also define an additional process W in order to synchronize on events. We finally define a single Boolean variable v, that is initially set to *False*, and will be updated to *True* in only one special process; then we will show that $v = True$ can be reached if and only if the machine halts.

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

- $Var = \{v\}$;
- $U = \{a, a_{-1}, a_{+1}, b, b_{-1}, b_{+1}\}$;
- V_0 initializes v to *False*;
- $K_0 = \{a = a_{+1} - 1 = a_{-1} + 1 \wedge b = b_{+1} - 1 = b_{-1} + 1\}$; and
- P_{CM} is explained in the following.

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 control variable l_i, let I_{ij} be the jth instruction starting in the control variable i. Figure 4 depicts one such instruction I_{ij} for some j. For each instruction I_{ij},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.

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

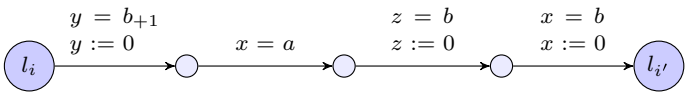

Fig. 4 Undecidability proof of Alur et al. (1993)

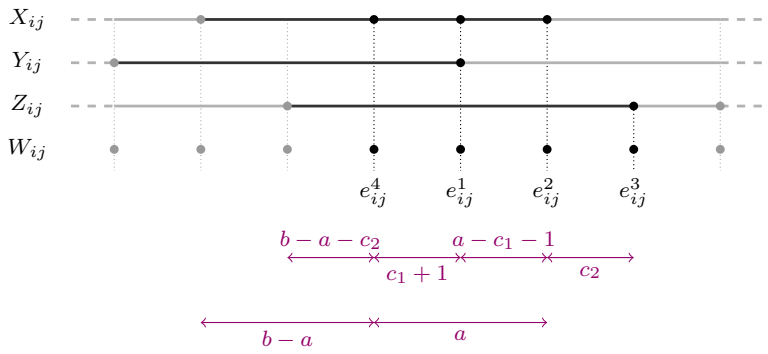

Fig. 5 Proof of undecidability: synchronization between processes

The three processes X, Y, Z correspond to the three clocks x, y, z, respectively, of Fig. 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 Fig. 4. 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}. Figure 5 gives the idea for our construction, and specifies in particular the duration between any two events for the sake of better understanding. Note that our construction is slightly different from that of Alur et al. (1993) in the sense that we start the sequence of transitions from the point where x is reset, hence technically in the previous transition in the PTA model of Alur et al. (1993); hence the first event for the jth instruction in control variable i is e^4_{ij}. This also explains the order of the events in W_{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.

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

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 defined as $e_n \rightarrow \texttt{Skip}$. And we define $X_n \doteq e_n \rightarrow \{v := \mathit{True}\}\ \texttt{Skip}$. This gives the final synchronization updating v to *True*.

The global process encoding our construction scheme is given by:

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

The Skip construction prefixing each process except Y_1 allows these processes to idle for some time before starting, as the four processes are out of phase (see Fig. 5).

Then, if *CM* does not halt, there is no way to reach a configuration where v is *True*, and there exists no parameter valuation such that this valuation is reachable. If *CM* does halt, and suppose the value of C_1 (resp. C_2) never exceeds some constant $\mathbf{c}_1$ (resp. $\mathbf{c}_2$), then the set of parameter valuations for which $v = \mathit{True}$ is reachable is $\{a = a_{+1} - 1 = a_{-1} + 1 \wedge b = b_{+1} - 1 = b_{-1} + 1 \wedge a \geq \mathbf{c}_1 \wedge b - a \geq \mathbf{c}_2\}$. □

Now, we show that the problem remains undecidable even for regular PSTCSP, which directly comes from the expressiveness of regular PSTCSP.

Theorem 2 (Undecidability of EF-emptiness (regular PSTCSP)) *The EF-emptiness problem is undecidable for regular PSTCSP.*

Proof The construction in Alur et al. (1993) uses a translation from a 2-counter machine 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 automata with ϵ-transitions, which has been shown in Sect. 4.3 to be equivalent to regular PSTCSP. □

AF-Membership We now show that problem 4 is undecidable too.

Theorem 3 (Undecidability of AF-emptiness) *The AF-emptiness problem is undecidable for PSTCSP.*

Proof The AF-emptiness problem has been shown to be undecidable for U-PTA (Jovanovic et al. 2013). The formalism of U-PTA is a subclass of PTA. Furthermore, the proof of Jovanovic et al. (2013) only uses non-strict inequalities; as a consequence, the U-PTA used for the proof is a parametric closed timed automaton, and hence a subclass of parametric closed timed automata with ϵ-transitions. From Proposition 1, this U-PTA is equivalent to a regular PSTCSP model. As a consequence, this problem is also undecidable for regular PSTCSP, and hence for full PSTCSP. □

Remark 3 (Reachability) We considered so far the reachability of a variable valuation V. This notion can be generalized to the reachability of a given process, or to the reachability of both a given process and a variable valuation. The first case (viz., the reachability of a given process P) can be obtained from the proof of Theorem 1 by replacing $X_n \doteq e_n \rightarrow \{v := \mathit{True}\}\ \texttt{Skip}$ with $X_n \doteq e_n \rightarrow P$. The second case (viz., the reachability of a given process P and a variable valuation V) can be obtained using $X_n \doteq e_n \rightarrow \{\mathit{Var} := V\}\ P$.

Another interesting problem is to determine, given a PSTCSP model $\mathsf{M} = (Var, U, V_0, P_0, K_0)$, whether there exists a parameter valuation such that P_0 can derive back to P_0 in a non-null number of steps. It follows from Remark 3 that this problem (that could be called return-to-init-emptiness problem) is a subcase of the EF-emptiness problem (by choosing P_{CM} as P in the proof of Theorem 1), and is thus undecidable.

4.5 Time-abstract equivalent runs

We prove here theoretical results that relate parametric runs (in PSTCSP) with non-parametric runs (in STCSP). These results will be needed when proving the correctness of the inverse method (see Sect. 5.3).

First, we need to recall the syntax and formal semantics of STCSP. (We sometime adapt the notations and names to our setting.)

4.5.1 Syntax and semantics of stateful timed CSP

An STCSP model (originally defined in Sun et al. 2013, Section 3.1) is a tuple $M = (Var, V_0, P_0)$ where $Var \subset \mathcal{V}ar$, V_0 is the initial variable valuation, and $P_0 \in \mathcal{P}^{NP}$ is a process. The set $\mathcal{P}^{NP}$ of all possible non-parametric processes (originally defined in Sun et al. 2013, Section 3.1) is the set of all processes defined using the grammar in Fig. 1, with the exception that $u \in \mathbb{R}_+$. That is, only constant reals (instead of parameters) are allowed in the timing constructs.

We now recall the non-parametric semantics of STCSP models (called "time-abstract semantics" in Sun et al. 2013). This semantics relies on the following notion of non-parametric symbolic states (called "abstract system configurations" in Sun et al. 2013).

Definition 7 (*Non-parametric symbolic state*) Given an STCSP model, a *non-parametric symbolic state* is a triple (V, P, D), where V is a variable valuation, $P \in \mathcal{P}^{NP}$ is a process and $D \in \mathcal{K}_X$ is a constraint on the clocks.

We now adapt to our notations the non-parametric semantics of STCSP models (originally defined in Sun et al. 2013, Definition 4.2) as follows.

Definition 8 (*Non-parametric semantics*) Let $Y = \langle x_0, \cdots \rangle$ be a sequence of clocks. Let $M = (Var, V_0, P_0)$ be an STCSP model. The *non-parametric semantics* of M, denoted by $\mathcal{L}_M$, is an LTS $(S, s_0, \Rightarrow_{NP}, \Sigma_\tau)$ where

$$S = \{(V, P, D) \in \mathcal{V}(Var) \times \mathcal{P} \times \mathcal{K}_X\},$$
$$s_0 = (V_0, P_0, True)$$

and the transition relation $\Rightarrow_{NP}$ is the smallest transition relation satisfying the following. For all $(V, P, D) \in S$, if x is the first clock in the sequence Y which is not in $cl(P)$, and $(V, Act^{NP}(P, x), D \wedge x = 0) \rightsquigarrow^a_{NP} (V', P', D')$, where D' is satisfiable, then we have: $\big((V, P, D), a, (V', P', D'{\downarrow}_{cl(P')})\big) \in \Rightarrow_{NP}$.

In this definition, Act^{NP} denotes the activation function for STCSP. This function (originally defined in Sun et al. 2013, Figure 3) is identical to the activation function Act for PSTCSP defined in Fig. 2, with the exception that u denotes a constant real instead of a parameter. As in PSTCSP, $cl(P)$ denotes the set of *active clocks* associated with an STCSP process P. The $\rightsquigarrow_{NP}$ transition relation for STCSP is defined using a set of rules. For the sake of conciseness, these rules (defined in Sun et al. 2013, Figure 6) are not recalled here, but are identical to the firing rules for PSTCSP (defined in Appendix), with the exception that parameters are replaced with constants.

In the following, we will write $s_1 \overset{a}{\Rightarrow}_{NP} s_2$ for $(s_1, a, s_2) \in \Rightarrow_{NP}$.

Remark 4 The semantics of a PSTCSP model can now be understood intuitively as the union of the semantics of the valuated non-parametric STCSP models, for all possible parameter valuations. For each parameter valuation π, we may view a symbolic state $s = (V, P, C)$ as the set of triples (V, P, D) such that for all clock valuation w such that $w \models D$, we have $< w, \pi > \models C$.

4.5.2 Results

The main result of this section will be Theorem 4, that relates non-parametric and parametric semantics. Similar results have been proved for parametric timed automata (Hune et al. 2002) and parametric time Petri nets Traonouez et al. 2009); we will reuse here the same reasoning, with some modifications due to the specific nature of PSTCSP.

Due to the presence of timing constructs in the processes contained in the traces (in contrast to PTA where traces contain only locations with no timing information), we define the notion of π-equivalence for traces. A trace of M and a trace of $\mathsf{M}[\pi]$ are π-equivalent if they agree on all elements (events, variables), except on the processes, that must be such that $P^\pi = P[\pi]$ for each process of the trace.

Definition 9 (π*-equivalence*) Let M be a PSTCSP model, and π be a parameter valuation.

Let $t = (V_0, P_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (V_m, P_m)$ be a trace of M.

Let $t^\pi = (V_0^\pi, P_0^\pi) \overset{a_0^\pi}{\Rightarrow}_{NP} \cdots \overset{a_{m-1}^\pi}{\Rightarrow}_{NP} (V_m^\pi, P_m^\pi)$ be a trace of $\mathsf{M}[\pi]$.

We say that t and t^π are π-equivalent if

1. for all $0 \leq i \leq m-1$, we have $a_i = a_i^\pi$, and
2. for all $0 \leq i \leq m$, we have $V_i = V_i^\pi$ and $P_i[\pi] = P_i^\pi$.

We extend the notion of π-equivalence to trace sets, and say that two trace sets are π-equivalent if each trace of the first one is π-equivalent to a trace of the second one, and vice-versa. We say that two traces (resp. trace sets) are equivalent if there exists some π such that they are π-equivalent. We finally define below the notion of time-abstract equivalence for runs, derived from π-equivalence. The term "time-abstract" comes from the fact that two runs are time-abstract equivalent if they have the same traces, hence the same time-abstract behavior. Also note that this notion is equivalent to the "trace simulation" used in, e.g., Hune et al. (2002), André et al. (2009).

Definition 10 (*Time-abstract equivalence*) Let M be a PSTCSP model, and π be a parameter valuation. Let r be a run of M, and r^π a run of $\mathsf{M}[\pi]$. We say that r and r^π are *time-abstract equivalent* if their associated traces are π-equivalent.

We will show in Propositions 3 and 4 that, given a parameter valuation π that satisfies some conditions, each run in $\mathsf{M}[\pi]$ is time-abstract equivalent to a run in M.

The following lemma states that constraints on the parameters can only become more restrictive along a run.

Lemma 4 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model. Let* $(V, P, C) \stackrel{a}{\Rightarrow} (V', P', C')$ *be a transition in the semantics of* M. *Then* $C'{\downarrow_U} \subseteq C{\downarrow_U}$.

Proof From the symbolic semantics of PSTCSP, C' is obtained from C through the following steps:

(1) C is first conjuncted with $x = 0$. For sake of clarity, let $C_1 = (C \wedge x = 0)$. From Lemma 1, we have $C_1{\downarrow_U} = C{\downarrow_U}$.
(2) Then, a constraint (say C_2) is obtained from C_1 using the transition relation $\rightsquigarrow$. Since the transition relation $\rightsquigarrow$ is recursive, we need to reason by induction to show that $C_2{\downarrow_U} \subseteq C_1{\downarrow_U}$. Base case: let us show that this holds for the non-recursive rules (viz., aki, aev, aac, $co2$, $co3$, $await$, $ato3$, $ait2$). For all these rules, C_2 is of the form ${C_1}^\uparrow \wedge C'$, where $C' \in \mathcal{K}_{X \cup U}$. (The constraint C', possibly equal to *True*, is made of a conjunction of inequalities such as $x = u$ or $idle(P)$.) Recall from Lemma 2 that time elapsing keeps the parametric constraint unchanged. Since $({C_1}^\uparrow){\downarrow_U} = C_1{\downarrow_U}$, then $({C_1}^\uparrow \wedge C'){\downarrow_U} \subseteq C_1{\downarrow_U}$, from Lemma 1. This completes the base case.
Induction case: the general form of C_2 is $C' \wedge C''$ for all recursive rules (except $ase2$ and $apa3$), where C' is obtained from C_1 by recursively applying transition relation $\rightsquigarrow$, and $C'' \in \mathcal{K}_{X \cup U}$. (The constraint C'', possibly equal to *True*, is made of a conjunction of inequalities such as $x \leq u$ or $idle(P)$. For example, in rule $aex1$, C'' is $idle(P)$.) Assume that $C'{\downarrow_U} \subseteq C_1{\downarrow_U}$ and let us show that $C_2{\downarrow_U} \subseteq C_1{\downarrow_U}$. Since $C'{\downarrow_U} \subseteq C_1{\downarrow_U}$, and $C_2 = C' \wedge C''$ then the result is immediate from Lemma 1. The case for rule $ase2$ is similar, except that the form of C_2 is $C_1 \wedge C'$; since $C'{\downarrow_U} \subseteq C_1{\downarrow_U}$ holds (by induction hypothesis), then $(C_1 \wedge C'){\downarrow_U} \subseteq C_1{\downarrow_U}$ holds from Lemma 1. For rule $apa3$, C_2 is of the form $C' \wedge C''$, where both C' and C'' are obtained from C_1 by recursively applying transition relation $\rightsquigarrow$. From the induction hypothesis, $C'{\downarrow_U} \subseteq C_1{\downarrow_U}$ and $C''{\downarrow_U} \subseteq C_1{\downarrow_U}$ hence $(C' \wedge C''){\downarrow_U} \subseteq C_1{\downarrow_U}$ from Lemma 1. This completes the induction step.
(3) Finally, C' is obtained from C_2 by removing the clocks not in $cl(P')$, i.e., by projecting onto $cl(P') \cup U$. By definition of the projection, $C_2{\downarrow_{cl(P') \cup U}} = \{< w, \pi > | \; w : cl(P') \rightarrow \mathbb{R}_+ \wedge \pi : U \rightarrow \mathbb{R}_+ \wedge < w, \pi > \models C\}$. Since $U \subseteq cl(P') \cup U$, then $(C_2{\downarrow_{cl(P') \cup U}}){\downarrow_U} = C_2{\downarrow_U}$. Since $C_2{\downarrow_U} \subseteq C_1{\downarrow_U}$ then $(C_2{\downarrow_{cl(P') \cup U}}){\downarrow_U} \subseteq C_1{\downarrow_U}$. This completes the proof.

□

The following lemma relates the initial state in the parametric and non-parametric semantics.

Lemma 5 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model. Let* $\pi \models K_0$. *Suppose that* (P_0, V_0, K_0) *is the initial state of the semantics of* M. *Then the initial state of the semantics of* $\mathsf{M}[\pi]$ *is* (V_π, P_π, D_π) *with* $P_\pi = P_0[\pi]$, $V_\pi = V_0$, *and* $D_\pi = K_0[\pi]$.

Proof From the definition of the model valuation, $P_\pi = P_0[\pi]$, and $V_\pi = V_0$. From Definition 5, the initial state of the semantics of M is (V_0, P_0, K_0). From Definition 8, the initial state of the semantics of $\mathsf{M}[\pi]$ is $(V_0, P_0[\pi], True)$. Since K_0 is a constraint on the parameters and $\pi \models K_0$, then $K_0[\pi] = True$. Hence $D_\pi = K_0[\pi]$. □

The following two lemmas relate a given transition in the parametric and non-parametric semantics.

Lemma 6 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model. Suppose that* $(V, P, C) \stackrel{a}{\Rightarrow} (V', P', C')$ *is a transition in the semantics of* M. *Let* $\pi \models C'$. *Suppose that* $(V, P[\pi], C[\pi])$ *is a state of the semantics of* $\mathsf{M}[\pi]$. *Then the transition* $(V, P[\pi], C[\pi]) \Rightarrow^a_{NP} (V', P'[\pi], C'[\pi])$ *belongs to the semantics of* $\mathsf{M}[\pi]$.

Proof First note that, from the definition of a transition in the semantics of M, C' is satisfiable. By definition of the safisfiability, there exists at least one π such that $\pi \models C'$. The proof of the result then comes from the fact that each firing rule of PSTCSP (see Appendix) has an equivalent firing rule in STCSP (see Sun et al. 2013, Figure 6), where a parameter is simply replaced with the corresponding constant in π. Hence the transition relation is equivalent in both formalisms. □

Lemma 7 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model. Suppose that* $(V, P_\pi, D_\pi) \stackrel{a}{\Rightarrow} (V', P'_\pi, D'_\pi)$ *is a transition in the semantics of* $\mathsf{M}[\pi]$. *Let* $\pi \models C'$. *Suppose that* (V, P, C) *is a state of the semantics of* M, *with* $P_\pi = P[\pi]$ *and* $D_\pi = C[\pi]$. *Then a transition* $(V, P, C) \stackrel{a}{\Rightarrow} (V', P', C')$ *belongs to the semantics of* M, *with* $P'_\pi = P'[\pi]$ *and* $D'_\pi = C'[\pi]$.

Proof As for Lemma 6, the proof comes from the fact the transition relation is equivalent in both formalisms. □

The following proposition is the equivalent for PSTCSP of Proposition 3.17 in Hune et al. (2002) in the setting of parametric timed automata.

Proposition 3 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model. For each parameter valuation* π, *if there is a run in the semantics of* M *reaching state* (V', P', C'), *with* $\pi \models C'$, *then this run is time-abstract equivalent to a run in the semantics of* $\mathsf{M}[\pi]$ *reaching state* $(V', P'[\pi], C'[\pi])$.

Proof By induction on the number of transitions in the run.

Base case: From Lemma 5.

Induction step: Assume there exists a run in the semantics of M ending with a transition $(V, P, C) \stackrel{a}{\Rightarrow} (V', P', C')$ with $\pi \models C'$. From Lemma 4, $\pi \models C$. From the induction hypothesis, there is a run in the semantics of $\mathsf{M}[\pi]$ leading up to state $(V, P[\pi], C[\pi])$. Since $\pi \models C'$, by Lemma 6, we have that $(V, P[\pi], C[\pi]) \Rightarrow^a_{NP} (V', P'[\pi], C'[\pi])$ is a transition in the semantics of $\mathsf{M}[\pi]$. Hence the run in the semantics of M is time-abstract equivalent to a run in the semantics of $\mathsf{M}[\pi]$.

□

The following proposition is the equivalent for PSTCSP of Proposition 3.18 in Hune et al. (2002) in the setting of parametric timed automata.

Proposition 4 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model. For each parameter valuation* π*, if there is a run in the semantics of* $\mathsf{M}[\pi]$ *reaching state* (V', P'_π, D'_π)*, then this run is time-abstract equivalent to a run in the semantics of* M *reaching state* (V', P', C')*, with* $P'_\pi = P'[\pi]$ *and* $D'_\pi = C'[\pi]$.

Proof By induction on the number of transitions in the run.

Base case: From Lemma 5.
Induction step: Assume there exists a run in the semantics of $\mathsf{M}[\pi]$ ending with a transition $(V', P'_\pi, C'_\pi) \Rightarrow^{a}_{NP} (V, P_\pi, D_\pi)$ From the induction hypothesis, there is a run in the semantics of $\mathsf{M}[\pi]$ leading up to a state (V, P, C) such that $P_\pi = P[\pi]$ and $D_\pi = C[\pi]$. By Lemma 7, we have that a transition $(V, P, C) \stackrel{a}{\Rightarrow} (V', P', C')$ belongs to the semantics of M, with $P'_\pi = P'[\pi]$ and $D'_\pi = C'[\pi]$. Hence the run in the semantics of $\mathsf{M}[\pi]$ is time-abstract equivalent to a run in the semantics of M.

□

The following theorem defines the reachability condition of a process, and relates non-parametric runs and parametric runs. This is the equivalent for PSTCSP of Theorem 13 in Traonouez et al. (2009) in the setting of parametric time Petri nets.

Theorem 4 *Let* $\mathsf{M} = (Var, U, V_0, P_0, K_0)$ *be a PSTCSP model, and let* (V, P, C) *be a state of* M*. Let* π *be a parameter valuation. Then:*

$$(V, P[\pi], C[\pi]) \in \mathsf{M}[\pi] \textit{ iff } \pi \in C{\downarrow_U}.$$

$C{\downarrow_U}$ *is called the* reachability condition *of* C.

Proof From Propositions 3 and 4. □

5 Parameter synthesis

In this section, we define three algorithms, all allowing for synthesizing parameters for PSTCSP corresponding to various notions of correctness. The first one (Sect. 5.1) simply explores the set of all reachable states, and can serve as a basis for classical algorithms for parameter synthesis such as the reachability of a given variable valuation or process. The second one (Sect. 5.2) synthesizes parameter valuations satisfying a 3-value predicate. The third one (Sect. 5.3) is based on a reference parameter valuation and synthesizes other parameter valuations having the same time-abstract behavior.

5.1 State space exploration

We first define a semi-algorithm to explore the state space until a fixpoint is reached, i.e., until no new state can be computed, or all new states have been encountered before.

Algorithm 1: Algorithm *reachAll*(M)

input : A PSTCSP model M with initial state s_0
output: Set of reachable states

```
1 S ← {s0}
2 while True do
3     if Post_M(S) ⊆ S then
4     return S
5     S ← S ∪ Post_M(S)
```

Recall from Definition 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, we define $Post_{\mathsf{M}}(S)$ (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, $Post_{\mathsf{M}}(S) = \{s' \mid \exists s \in S, \exists a \in \Sigma_\tau : s \stackrel{a}{\Rightarrow} s'\}$. 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., $S \subseteq S'$ if $\forall s \in S, s \in S'$. This inclusion test can be implemented using an SMT solver (as it is proposed, e.g., in Hune et al. (2002)); in our implementation PSyHCoS (see Sect. 6), we use the polyhedra inclusion test provided by the Parma Polyhedra Library (PPL) Bagnara et al. (2008).

5.1.1 Application to an example

We first introduce an example of a PSTCSP model that will be used to show the application of *reachAll* (and then of *IM* in Sect. 5.3).

Example 4 Consider the following PSTCSP model.

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

This model is actually non-parametric (*np* stands for non-parametric), with no variables. 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, b never occurs because `interrupt` occurs before `Wait`[2] is achieved. We give in Fig. 6 the set of reachable states in the form of an LTS.

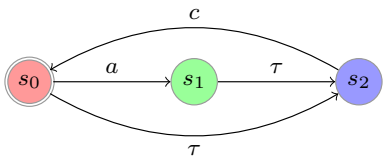

Fig. 6 Reachable states of process P^{np}

Now consider the following parametrized version of $\mathsf{M}_{ex}{}^{np}$.

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

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

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

Let us apply *reachAll* to M_{ex}. Since there are no variables in M_{ex}, 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 (i.e., x_1) and apply *Act* to P with x_1 to get:

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

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

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

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

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

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

We can apply rule $ait1$ (with $ase1$, $await$) to s_1', and get (after pruning of x_2):

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

We can also apply rule $ait2$ (and *idle*8, *idle*10) to s_1', and get:

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

w After pruning of both x_1 and x_2, we get $(c \rightarrow P, \mathit{True})$, which is equal to s_2'.

We can apply rule aev to s_2' to get (P, True), which is equal to s_0.

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

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

We can also apply rule $ait2$ (with *idle*8, *idle*2) to s_3 to get:

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

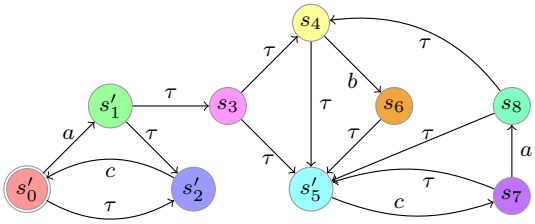

Fig. 7 States reachable in model M_{ex}

Which gives after pruning of x_1: $s'_5 = c \rightarrow P \ , \ u_2 \leq u_1$. Note that s'_5 is not equal to s_2, because the associated constraint is different.

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

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

We 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: $s_7 = P, u_2 \leq u_1$, 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:

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

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 Fig. 7 in the form of an LTS.

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. 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 P^{np} can only reach states (equivalent to) s'_0, s'_1 and s'_2. Indeed, we have that $\mathsf{M}_{ex}{}^{np} = \mathsf{M}_{ex}[\pi]$, where $\pi = \{u_1 = 1, u_2 = 2\}$, hence $u_1 < u_2$.

5.1.2 Application to parameter synthesis

We showed in Theorem 1 that parameter synthesis is undecidable for PSTCSP. Still, from the set of symbolic states computed by *reachAll*, one can apply classical semi-algorithms for parameter synthesis. Suppose the result of *reachAll* is an oriented graph (in the form of the one depicted in Fig. 7), and let $succ(s)$ be the set of successors of a given reachable state s. For example, in Fig. 7, we have $succ(s'_0) = \{s'_1, s'_2\}$.

Then, from the result of *reachAll*, we can for example retrieve the set of parameter valuations such that a given variable valuation, and/or a given process Q (see Remark 3 on page 3) is reachable. This procedure, that we name $\mathsf{EF}_Q(s, R)$, can be defined as follows (following the form defined in Jovanovic et al. (2013)):

$$\mathsf{EF}_Q\big((V, P, C), R\big) = \begin{cases} C{\downarrow_U} & \text{if } P = Q \\ \emptyset & \text{if } (V, P, C) \in R \\ \bigcup\limits_{s' \in succ(V,P,C)} \mathsf{EF}_Q\big(s', R \cup \{(V, P, C)\}\big) & \text{otherwise} \end{cases}$$

In this algorithm, Q denotes a process to be reached (the case with a reachable variable valuation is similar), (V, P, C) denotes the current state explored, and R denotes the set of states explored so far. Recall from Jovanovic et al. (2013) that the algorithm is initially called on the initial state of the model (see Definition 5) and on an empty set of reachable states, viz., $\mathsf{EF}_Q((V_0, P_0, K_0), \emptyset)$. This algorithm returns the projection of a constraint onto the parameters if the process P is the one searched for (case $P = Q$); otherwise, if the current state has been visited (case $(V, P, C) \in R$), it stops; otherwise, it returns the union of the recursive application of the algorithm on all successor states of the current state, and adds the current state to the set of reachable states. Although this algorithm may be used to synthesize "good" (correct) parameter valuations, it is usually used to synthesize "bad" ones: usually, the process Q to be reached is a "bad" process, and the set of correct valuations is then all parameter valuations *except* the ones output by EF_Q.

Let us now synthesize the set of parameter valuations such that the process $Q \doteq (\texttt{Skip}; b \rightarrow \texttt{Stop})\ \texttt{interrupt}[u_1]_{x_1}\ c \rightarrow P$ is reachable in the model of Example 4. Hence, we call $\mathsf{EF}_Q(s'_0, \emptyset)$. Since the process in s'_0 is different from Q, the first case of the algorithm does not apply (recall that the description of states is available above); since $R = \emptyset$, the second case does not apply; from the third case, and since s'_0 has two successors s'_1 and s'_2, we get $\mathsf{EF}_Q(s'_1, \{s'_0\}) \cup \mathsf{EF}_Q(s'_2, \{s'_0\})$. In the call $\mathsf{EF}_Q(s'_1, R\{s'_0\})$, the third case will again apply; since s'_1 has two successors s'_2 and s_3, we get $\mathsf{EF}_Q(s'_2, \{s'_0, s'_1\}) \cup \mathsf{EF}_Q(s_3, \{s'_0, s'_1\})$. In the call $\mathsf{EF}_Q(s_3, R\{s'_0, s'_1\})$, we now have that the process associated with s_3 is equal to Q. Hence, this call returns the projection onto U of the constraint associated with s_3, viz., $u_2 \leq u_1$. The remaining calls, after a few more iterations, will return the same constraint. Hence the result of $\mathsf{EF}_Q(s'_0, \emptyset)$ is $\{u_2 \leq u_1\}$. If Q is a "bad" process, then the set of good parameter valuations is $u_2 > u_1$.

5.1.3 Non-termination

Proposition 5 (Non-termination) *Let* M *be a PSTCSP model. Then Algorithm reach all*(M) *does not terminate in the general case.*

Proof See counterexample in Example 5. □

Example 5 Consider the PSTCSP model $\mathsf{M} = (\emptyset, \{u_1, u_2\}, \emptyset, P, True)$ where P is defined as follows.

$$P \doteq Q \texttt{ interrupt}[u_1]\, b$$
$$Q \doteq a \rightarrow \texttt{Wait}[u_2]; Q$$

It can be shown that *reachAll* will go into an infinite loop, by generating in particular states of the form:

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

with i growing without bound.

5.1.4 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 Sun et al. (2013) for STCSP (e.g., model checking with or without non-Zenoness ass, refinement checking, etc.).

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 setting do not apply anymore.

5.2 Parameter synthesis based on 3-value predicates

5.2.1 3-Value predicates

We consider here 3-value predicates: these predicates are properties on the model variables that can be true, false, or neither true nor false. The application is to differentiate between "good" states, "bad" states and states that are neither good nor bad.

Differentiating between good states, bad states, and neither good nor bad states can be used to encode *observers*. Observers are special processes analyzed in parallel with the system, the discrete state of which depends on the rest of the system's evolution. For example, they can monitor a predefined order of events, or check that some deadlines are met. A major advantage of observers is that they reduce complex properties (encoded in the observer process) to reachability testing. In many cases (see, e.g., observers for timed automata and STCSP (Aceto et al. 1998a, 1998b; André 2013)), observers use good states and bad states (the other states being neither good nor bad). In that case, the property encoded in the observer is true if and only if at least one good state is reachable, and no bad state is reachable. This is the problem we address here.

More specifically, this can be used when modeling scheduling problems. For example, consider an acylic scheduling problem where a set of tasks has to be completed

[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.

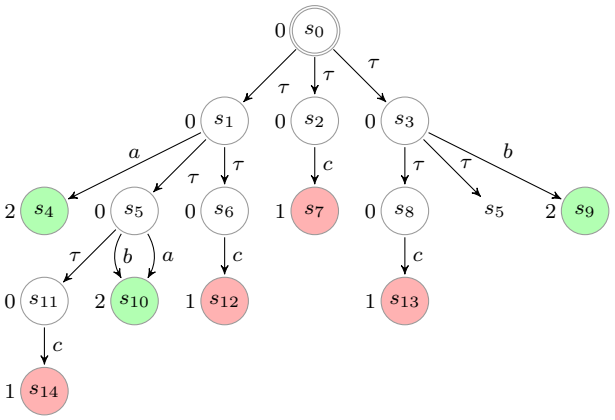

Fig. 8 An example of a 3-value predicate applied to a state space

(only once) within a given timing bound; since the model is acyclic, the state space can be seen as an acyclic directed graph, where each branch encodes one possible order of the events encoding the starting and ending points of the different tasks. In this case, only the terminal states (i.e., the last state of each branch) can be said to be good or bad; the other states are not relevant, in the sense that one does not know (yet) whether the deadline is met or not. The problem consists in defining sets of values for the parameters for which the system is guaranteed to meet its deadline.

Given a valuation V of the variables, we write $\varphi(V) = \mathit{True}$ if the predicate is true, $\varphi(V) = \mathit{False}$ if the predicate is false, and $\varphi(V) = \mathit{Unknown}$ otherwise.

Example 6 Consider the following 3-value predicate on a single integer-valued variable v:

$$\varphi(v) = \begin{cases} \mathit{True} & \text{if } v = 2 \\ \mathit{False} & \text{if } v = 1 \\ \mathit{Unknown} & \text{otherwise} \end{cases}$$

An example of the application of this 3-value predicate to a reachability graph is given in Fig. 8, where the value of v is given to the left of each state. The value according to the predicate is *True* for the states s_4, s_9, and s_{10} (since $v = 2$), *False* for states s_7, s_{12}, s_{13} and s_{14} (since $v = 1$), and *Unknown* for the other states (since $v = 0$).

In the rest of this subsection, we will address the problem of synthesizing parameters ensuring that at least one good state is reachable, and no bad state is reachable.

Remark 5 We consider 3-value predicates on the variables only. This restriction is not strong in practice (see Remark 3), and one could easily extend predicates to processes (e.g., one may want to define a 3-value predicate that is true if the process does not contain any `deadline`, or more generally any timed construct).

5.2.2 Synthesis algorithm

Before describing our synthesis algorithm with respect to 3-value predicates, we make two asss throughout this subsection. First, we assume that models are non-recursive (as defined in Definition 3).

Assumption 1 All models are non-recursive.

The recursive case is discussed at the end of the subsection. Our algorithm does not depend on this ass but, for recursive models, it may not terminate.

Second, we assume that no bad state can occur after a good state on the same run.

Assumption 2 For any model, for any parameter valuation, no bad state can occur after a good state on the same run.

Hence, a good state must be seen as a kind of terminal state. Our algorithm relies directly on this ass: synthesizing parameter valuations for models not satisfying Assumption 2 would require a different algorithm. Note that this ass is true for acyclic schedulability problems: only the last state of a run can be said to be good (if the deadline is met) or bad (otherwise). Hence, no bad state can occur after a good state.

We introduce our algorithm *3VPsynthesis* in Algorithm 2. Given a PSTCSP model M and a 3-value predicate φ, this algorithm synthesizes a set of parameter valuations guaranteeing the following notion of correctness: at least one good state and no bad state is reachable according to φ.

Algorithm 2: Algorithm *3VPsynthesis*(M, φ)

input : PSTCSP model $\mathsf{M} = (Var, U, V_0, P_0, K_0)$
input : 3-value predicate φ
output: (non-convex) constraint K over the parameters

1 $K \leftarrow False$; $K_{bad} \leftarrow K_0$; $S \leftarrow \{(V_0, P_0, K_0)\}$
2 **while** *True* **do**
3 **foreach** $(V, P, C) \in Post_{\mathsf{M}}(S)$ **do**
4 **if** $\varphi(V) = True$ **then**
5 $K \leftarrow K \vee C{\downarrow_U}$;
6
7 **else if** $\varphi(V) = False$ **then**
8 $K_{bad} \leftarrow K_{bad} \wedge \neg C{\downarrow_U}$;
9 **else**
10 $S \leftarrow S \cup \{(V, P, C)\}$;
11 **if** $Post_{\mathsf{M}}(S) \subseteq S$ **then**
12 **return** $K \wedge K_{bad}$;

The algorithm maintains two constraints: the constraint K guaranteeing that at least one good state (according to φ) is reachable, and the constraint K_{bad} guaranteeing that no bad state is reachable. The algorithm also maintains the set S of visited states. Initially, K is set to false (no good state has been reached yet), K_{bad} is set to the

initial constraint, and S is set to the initial state (line 1). Then, the algorithm iteratively computes states in a breadth-first manner. If a new state is such that $\varphi(V) = \mathit{True}$, then the projection onto the parameters of its constraint is added to K as a disjunction (line 5), so as to allow this state to be reached in at least one run. Conversely, if a new state is such that $\varphi(V) = \mathit{False}$, then the negation of the projection onto the parameters of its constraint is added to K as a conjunction (line 8), so as to forbid this state to be reachable in all runs. Otherwise, the state is added to the list of visited states (line 10), and its successors will be visited at the next iteration. Finally, if no new state is computed, or all new states have been visited before (fixpoint condition line 11), then the intersection of K with K_{bad} is returned (line 12). Note that the resulting constraint is in general non-convex due to the use of disjunctions.

5.2.3 Application to an example

Example 7 Consider the following example of a PSTCSP model $\mathsf{M} = (\mathit{Var}, U, V_0, P, K_0)$, where $\mathit{Var} = \{v\}$, $U = \{u_1, u_2, u_3\}$, V_0 assigns v to 0, $K_0 = \mathit{True}$, and P is defined as follows.

$$P \doteq (P_1 \,\square\, P_2)\texttt{timeout}[u_3](c\{v := 1\} \twoheadrightarrow \texttt{Stop})$$
$$P_1 \doteq (\texttt{Wait}[u_1]; a\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_1]$$
$$P_2 \doteq (\texttt{Wait}[u_2]; b\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_2]$$

In this example, P_1 (resp. P_2) is a process that first waits for u_1 (resp. u_2) time units; then event a (resp. b) occurs immediately (due to the urgent transition) and sets v to 2, before deriving to `Skip`. The main process P is an external choice between P_1 and P_2, that will timeout after u_3 units of time (i.e., if neither a nor b occur at that time), in which case v is set to 1, and the process stops. Note that P is non-recursive.

A possible interpretation of this example is that either a task with a duration of u_1 time units or a task with a duration of u_2 time units must be completed before a deadline of u_3 time units. The variable v encodes that one task is completed before the deadline ($v = 2$) or no task is completed before the deadline ($v = 1$), which can be seen as a deadline miss.

It is interesting to know for which values of the parameters it is guaranteed that one task is completed before the deadline. Hence, let us apply *3VPsynthesis* to M and to the 3-value predicate φ defined in Example 6, that assigns *True* to 2, *False* to 1, and *Unknown* otherwise. The initial assignment (line 1 in Algorithm 2) gives $K \leftarrow \mathit{False}$, $K_{bad} \leftarrow \mathit{True}$, and $S \leftarrow \{(v = 0, P_0, \mathit{True})\}$. Let us now iterate the **while** loop. The state space will be the one given in Fig. 8. We store the description of the states in Table 1 for better readability.

First iteration Let us compute $\mathit{Post}_{\mathsf{M}}(S)$. If $\texttt{Wait}[u_1]$ finishes first, then state s_1 is reached from s_0. The value of v is 0 in s_1; hence $\varphi(v) = \mathit{Unknown}$, hence the algorithm only performs $S \leftarrow S \cup \{s_1\}$ (line 10).

If $\texttt{timeout}[u_3]$ occurs first, a state s_2 is reached from s_0. Again, the value of v is 0 in s_2; hence s_2 is added to S.

Table 1 Description of the states in Fig. 8

s	v	P	C
s_0	0	$\big(((\texttt{Wait}[u_1]; a\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_1]) \Box ((\texttt{Wait}[u_2]; b\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_2])\big)\ \texttt{timeout}[u_3](c\{v := 1\} \twoheadrightarrow \texttt{Stop})$	*True*
s_1	0	$\big(((\texttt{Skip}; a\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_1]_{x_1}) \Box ((\texttt{Wait}[u_2]_{x_1}; b\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_2]_{x_1})\big)\ \texttt{timeout}[u_3]_{x_1}(c\{v := 1\} \twoheadrightarrow \texttt{Stop})$	$x_1 = u_1 \wedge x_1 \leq u_2 \wedge x_1 \leq u_3$
s_2	0	$c\{v := 1\} \twoheadrightarrow \texttt{Stop}$	$u_3 \leq u_1 \wedge u_3 \leq u_2$
s_3	0	$\big(((\texttt{Wait}[u_1]_{x_1}; a\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_1]_{x_1}) \Box ((\texttt{Skip}; b\{v := 2\} \twoheadrightarrow \texttt{Skip}))\ \texttt{within}[u_2]_{x_1})\ \texttt{timeout}[u_3]_{x_1}(c\{v := 1\} \twoheadrightarrow \texttt{Stop})$	$x_1 = u_2 \wedge x_1 \leq u_1 \wedge x_1 \leq u_3$
s_4	2	$\texttt{Skip}$	$u_1 \leq u_2 \wedge u_1 \leq u_3$
s_5	0	$\big(((\texttt{Skip}; a\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_1]_{x_1}) \Box ((\texttt{Skip}; b\{v := 2\} \twoheadrightarrow \texttt{Skip})\ \texttt{within}[u_2]_{x_1})\big)\ \texttt{timeout}[u_3]_{x_1}(c\{v := 1\} \twoheadrightarrow \texttt{Stop})$	$x_1 = u_1 \wedge x_1 = u_2 \wedge x_1 \leq u_3$
s_6	0	$c\{v := 1\} \twoheadrightarrow \texttt{Stop}$	$u_1 = u_3 \wedge u_1 \leq u_2$
s_7	1	$\texttt{Stop}$	$u_3 \leq u_1 \wedge u_3 \leq u_2$
s_8	0	$c\{v := 1\} \twoheadrightarrow \texttt{Stop}$	$u_2 = u_3 \wedge u_2 \leq u_1$
s_9	2	$\texttt{Skip}$	$u_2 \leq u_1 \wedge u_2 \leq u_3$
s_{10}	2	$\texttt{Skip}$	$u_1 = u_2 \wedge u_1 \leq u_3$
s_{11}	0	$c\{v := 1\} \twoheadrightarrow \texttt{Stop}$	$u_1 = u_2 = u_3$
s_{12}	1	$\texttt{Stop}$	$u_1 = u_3 \wedge u_1 \leq u_2$
s_{13}	1	$\texttt{Stop}$	$u_2 = u_3 \wedge u_2 \leq u_1$
s_{14}	0	$\texttt{Stop}$	$u_1 = u_2 = u_3$

Symmetrically to s_1, if $\texttt{Wait}[u_2]$ finishes first, a state s_3 is reached from s_0. Again, the value of v is 0 in s_3; hence s_3 is added to S.

All s_1, s_2 and s_3 have successor states that are not yet computed,[9] hence we do not have that $Post_{\mathsf{M}}(S) \subseteq S$ (line 11), hence the algorithm goes one iteration further.

[9] The test $Post_{\mathsf{M}}(S) \subseteq S$ is a classical fixpoint test given in an algorithmic manner. Here, one does not know yet whether $Post_{\mathsf{M}}(S) \subseteq S$, since $Post_{\mathsf{M}}(S)$ will be computed at the next iteration. In practice, this is handled using a set of "old" states (computed at previous iterations), and a set of "new" states (computed at the current iteration).

Second iteration From state s_1, there are three possible successors: if a (which is an urgent event) occurs first, then state s_4 is reached. Since $v = 2$, then $\varphi(v) = \mathit{True}$; hence, we perform $K \leftarrow K \vee C_4{\downarrow_U}$ (line 5), and we now have $K = u_1 \leq u_2 \wedge u_1 \leq u_3$. Alternatively, $\texttt{Wait}[u_2]$ can complete before a occurs, reaching state s_5; since a is an urgent event, this happens in a 0-time duration, which explains the equality $x_1 = u_1 = u_2$ in C_5. Alternatively, $\texttt{timeout}[u_3]$ can occur first, reaching state s_6; again, there is an equality $u_1 = u_3$ in C_6 due to the fact that a is an urgent event. Both s_5 and s_6 are such that $\varphi(v) = \mathit{Unknown}$, hence they are stored within S.

From state s_2, there is only one successor s_7. This state is such that $\varphi(v) = \mathit{False}$; hence, the algorithm performs $K_{bad} \leftarrow K_{bad} \wedge \neg C_7{\downarrow_U}$ (line 8), and we now have $K_{bad} = \neg(u_3 \leq u_1 \wedge u_3 \leq u_2)$.

The successors of s_3 are symmetrical to those of s_1, and lead to states s_8, s_5 (already reached from s_1) and s_9.

At the end of the second iteration, we have $K = u_1 \leq u_2 \wedge u_1 \leq u_3 \vee u_2 \leq u_1 \wedge u_2 \leq u_3$, and $K_{bad} = \neg(u_3 \leq u_1 \wedge u_3 \leq u_2)$. Some of the new computed states have successor states that are not yet computed, hence the algorithm goes one iteration further.

Third iteration From state s_5, there are three possible successors: if a (which is an urgent event) occurs first, then state s_{10} is reached, with $\varphi(v) = \mathit{True}$, hence $u_1 = u_2 \wedge u_1 \leq u_3$ is added to K (as a disjunction). Similarly, if b occurs first, then the same state s_{10} is reached. And if $\texttt{timeout}[u_3]$ occurs first, then state s_{11} is reached, with $\varphi(v) = \mathit{Unknown}$; hence, s_{11} is added to S.

From state s_6, there is one successor s_{12}, where $\varphi(v) = \mathit{False}$; hence, $\neg(u_1 = u_3 \wedge u_1 \leq u_2)$ is added to K_{bad} (as a conjunction).

Symmetrically, from state s_8, there is one successor s_{13}, where $\varphi(v) = \mathit{False}$; hence, $\neg(u_2 = u_3 \wedge u_2 \leq u_1)$ is added to K_{bad}.

At the end of the third iteration, we have $K = (u_1 \leq u_2 \wedge u_1 \leq u_3) \vee (u_2 \leq u_1 \wedge u_2 \leq u_3) \vee (u_1 = u_2 \wedge u_1 \leq u_3)$, and $K_{bad} = \neg(u_3 \leq u_1 \wedge u_3 \leq u_2) \wedge \neg(u_1 = u_3 \wedge u_1 \leq u_2) \wedge \neg(u_2 = u_3 \wedge u_2 \leq u_1)$. Some of the new computed states have successor states that are not yet computed, hence the algorithm goes one iteration further.

Fourth iteration From state s_{11}, there is one successor s_{14}, where $\varphi(v) = \mathit{False}$; hence, $\neg(u_1 = u_2 = u_3)$ is added to K_{bad}.

Now, all states in S have either no successor, or a successor in S. Hence the fixpoint condition (line 11) is verified, and the algorithm terminates.

At the end of the algorithm, we have $K = (u_1 \leq u_2 \wedge u_1 \leq u_3) \vee (u_2 \leq u_1 \wedge u_2 \leq u_3) \vee (u_1 = u_2 \wedge u_1 \leq u_3)$, and $K_{bad} = \neg(u_3 \leq u_1 \wedge u_3 \leq u_2) \wedge \neg(u_1 = u_3 \wedge u_1 \leq u_2) \wedge \neg(u_2 = u_3 \wedge u_2 \leq u_1) \wedge \neg(u_1 = u_2 = u_3)$.

The result of the algorithm $K \wedge K_{bad}$ can be simplified (manually or using a constraint solver) into $(u_1 \leq u_2 \wedge u_1 < u_3) \vee (u_2 \leq u_1 \wedge u_2 < u_3) \vee (u_1 = u_2 \wedge u_1 < u_3)$. It will be shown in Proposition 7 that, for any parameter valuation satisfying this constraint, the system reaches at least one state where $v = 2$, and cannot reach any state where $v = 1$. An example of a parameter valuation satis-

fying this constraint is $u_1 = 1 \wedge u_2 = 2 \wedge u_3 = 3$. We remark that the resulting constraint is equivalent to $u_3 > \min(u_1, u_2)$; this is consistent with the correctness condition of this example, i.e., either the task of length u_1 time units or the task of length u_2 time units must be completed before the deadline of u_3 time units.

5.2.4 Soundness and completeness

The algorithm trivially terminates under Assumption 1 (non-recursivity).

Proposition 6 (Termination) *Let* M *be a PSTCSP model, and* φ *be a 3-value predicate on the variables of* M. *Then 3VPsynthesis*(M, φ) *terminates.*

Proof Due to the finite number of possible process derivations coming from Assumption 1. □

We show below that, for all $\pi \models$ *3VPsynthesis*(M, φ), at least one good state is reachable, and no bad state is reachable. Of course, this result relies on Assumption 2 that states that, once a good state has been reached, no bad state is reachable on the same run.

The following lemma will be used in the proof of Proposition 7.

Lemma 8 (Unreachability of bad states) *Let* M *be a PSTCSP model, and* φ *be a 3-value predicate on the variables of* M. *Suppose 3VPsynthesis*(M, φ) *terminates. Consider the constraint* K_{bad} *just before the end of the algorithm. Then for all* $\pi \models K_{bad}$, *no bad state is reachable in* $\mathsf{M}[\pi]$.

Proof By contradiction. Let $\pi \models K_{bad}$. Suppose a bad state is reachable in $\mathsf{M}[\pi]$, i.e., there exists a run r_π reaching a state (V, P_π, D_π) in the semantics of $\mathsf{M}[\pi]$. From Proposition 4, this run is time-abstract equivalent to a run r in the semantics of M reaching state (V, P, C), with $P_\pi = P[\pi]$ and $D_\pi = C[\pi]$. If this bad state (V, P, C) occurs after a good state on the same run r, then this violates Assumption 2. Hence, no good state occurs on the run leading to (V, P, C). Suppose without loss of generality that this bad state (V, P, C) is the first one along r, i.e., no bad state occurs earlier on the same run. (If this is not the case, then let us consider the first bad state instead.) Now, recall that Algorithm 2 computes all successor states along a run until a good or bad state is met. As a consequence, (V, P, C) has been met by Algorithm 2 and $\neg C{\downarrow_U}$ has been added to K_{bad} (line 8). Since $\pi \models K_{bad}$, then $\pi \not\models C{\downarrow_U}$. Hence, from Theorem 4, no state (V, P_π, D_π) is reachable in $\mathsf{M}[\pi]$, which contradicts the initial ass. □

Proposition 7 (Soundness) *Let* M *be a PSTCSP model, and* φ *be a 3-value predicate on the variables of* M. *Suppose 3VPsynthesis*(M, φ) *terminates with result* K.

Then for all $\pi \models K$, *at least one good state is reachable in* $\mathsf{M}[\pi]$, *and no bad state is reachable in* $\mathsf{M}[\pi]$.

Proof The result of *3VPsynthesis* is of the form $(K_1 \vee K_2 \vee \cdots \vee K_n) \wedge K_{bad}$. This can be rewritten $K_1 \wedge K_{bad} \vee K_2 \wedge K_{bad} \vee \cdots \vee K_n \wedge K_{bad}$. Consider $K_i \wedge K_{bad}$ for some $1 \leq$

$i \leq n$. Observe that K_i characterizes a good state (see line 5 in Algorithm 2). Hence, from Proposition 3, this good state is reachable for any $\pi \models K_i$. Since $K_i \wedge K_{bad} \subseteq K_i$, this good state is also reachable for any $\pi \models K_i \wedge K_{bad}$. Furthermore, from Lemma 8, no bad state is reachable in $\mathsf{M}[\pi]$, since K_{bad} contains the negation of an inequality associated with each of these reachable bad states. □

We finally state that *3VPsynthesis* is complete, i.e., that it synthesizes *all* possible parameter valuations such that at least one good state is reachable and no bad state is reachable.

Proposition 8 (Completeness) *Let* M *be a PSTCSP model, and* φ *be a 3-value predicate on the variables of* M. *Let* π *be a parameter valuation. Let* $K = 3VPsynthesis(\mathsf{M}, \varphi)$.

If at least one good state is reachable in $\mathsf{M}[\pi]$, *and no bad state is reachable in* $\mathsf{M}[\pi]$, *then* $\pi \models K$.

Proof From Theorem 4, the conjunction of the negated parametric constraints associated with the bad states is the minimal constraint (in terms of number of points) guaranteeing the non-reachability of the bad states. Similarly, from Theorem 4, the union of all the parametric constraints associated to the good states is the minimal constraint (in terms of number of points) guaranteeing the reachability of at least one good state. □

Recursive models We briefly discuss the case of recursive models. Although *3VPsynthesis* is guaranteed to terminate for non-recursive models (due to the finite number of possible process derivations), there exist models for which *3VPsynthesis* may not terminate. However, if *3VPsynthesis* does terminate for a given input, then its soundness and completeness are still ensured (since none of these proofs require the ass of non-recursivity). Hence it is a semi-algorithm.

5.3 Parameter synthesis using the inverse method

We extend here the inverse method *IM* to PSTCSP.

History The inverse method was first proposed in the framework of "time separation of events" (Encrenaz and Fribourg 2008). The "direct problem" in the framework of time separation of events can be stated as follows: "Given a system made of several connected components, each one entailing a local delay known with uncertainty, what is the maximum time for traversing the global system?" In Encrenaz and Fribourg (2008), the authors focus on the following *inverse problem*: "find intervals for component delays for which the global traversal time is guaranteed to be no greater than a specified maximum". The authors then introduce a method, the so-called *inverse method*, and show that this method solves the inverse problem in polynomial time. The inverse method was then formalized and extended to PTA in André et al. (2009), André and Soulat (2013), guaranteeing that the discrete behavior of the system (what we call here trace set) is preserved for any parameter valuation satisfying the constraint output by *IM*.

5.3.1 The inverse method for PSTCSP

Similarly to PTA, the main property of *IM* for PSTCSP will be the following: Given a PSTCSP model M and a reference parameter valuation π, *IM* synthesizes a constraint K on the parameters such that, for any $\pi' \models K$, the trace sets of $\mathsf{M}[\pi]$ and $\mathsf{M}[\pi']$ are equivalent. This method guarantees the time-abstract equivalence of the behaviors. Hence, all linear time properties valid in $\mathsf{M}[\pi]$ are also valid in $\mathsf{M}[\pi']$, and vice versa. Note that the algorithm *IM* is somehow independent of the notion of correctness (e.g., in contrast to *3VPsynthesis* that explicitly takes as input a 3-value predicate). However, if the behavior of the original system $\mathsf{M}[\pi]$ is correct (for whatever criterion of correctness based on the trace set), then $\mathsf{M}[\pi']$ is correct for any $\pi' \models K$ (and conversely if $\mathsf{M}[\pi]$ is incorrect).

In *IM*, we need to check whether the constraint associated with a state is satisfied by a given parameter valuation. This refers to the following notion.

Definition 11 (π*-compatibility*) Let M be a PSTCSP model, π be a parameter valuation, and $s = (V, P, C)$ be a state of M. The state s is said to be π *-compatible* if $\pi \models C$, and π *-incompatible* otherwise.

We introduce in Algorithm 3 the inverse method $IM(\mathsf{M}, \pi)$ for PSTCSP. We consider in the following the model $\mathsf{M} = (Var, U, V_0, P_0, K_0)$. 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 (line 5) and the negation $\neg J$ of J is added to K (line 6). The procedure is then started again with this new K, and so on, until a fixpoint is reached, i.e., all states have been visited before, and no new state is reachable (line 8). The algorithm finally returns the intersection of the projection onto the parameters U of the constraints associated with all reachable states (line 9).

The two main 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 $\pi' \models K$ any behavior different from the admissible behaviors under π;
2. the intersection of the constraints associated with all the reachable states guarantees that all the behaviors under π are allowed for all $\pi' \models K$.

5.3.2 Application to Example 4

Let us apply *IM* to M_{ex} and the following reference parameter valuation: $\pi = \{u_1 = 1, u_2 = 2\}$. Since $Var = \emptyset$, we denote each state by (P, C), where P is the current process, and C the current constraint on X and U.

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

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

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 $S \leftarrow S \cup Post_{\mathsf{M}}(S)$.

Algorithm 3: Algorithm $IM(\mathsf{M}, \pi)$

input : PSTCSP model $\mathsf{M} = (Var, U, V_0, P_0, K_0)$
input : Parameter valuation π
output: Constraint K over the parameters

$i \leftarrow 0$; $K \leftarrow K_0$; $S \leftarrow \{(V_0, P_0, K)\}$
while *True* **do**
  while *there are π-incompatible states in S* **do**
    Select a π-incompatible state (V, P, C) of S (i.e., s.t. $\pi \not\models C$) ;
    Select a π-incompatible J in $C{\downarrow_U}$ (i.e., s.t. $\pi \not\models J$) ;
    $K \leftarrow K \wedge \neg J$;
    $S \leftarrow \bigcup_{j=0}^{i} Post_{\mathsf{M}}^{j}(\{(V_0, P_0, K)\})$;
  if $Post_{\mathsf{M}}(S) \subseteq S$ **then**
    return $\bigcap_{(V,P,C)\in S} C{\downarrow_U}$;
  $i \leftarrow i + 1$;
  $S \leftarrow S \cup Post_{\mathsf{M}}(S)$

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

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

and $s'_2 = (c \rightarrow P, True)$. 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

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

The projection onto U of the constraint associated with s_3 gives $u_2 \leq u_1$, which is π-incompatible. As a consequence, we negate this inequality, and add it to K, which gives $K = u_2 > u_1$. Next, we perform $\bigcup_{j=0}^{i} Post_{\mathsf{M}}^{j}(\{(V_0, P_0, K)\})$; 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 associated constraint. The fixpoint is reached, and the intersection of the constraints on the parameters is returned (viz., $u_2 > u_1$).

From Theorem 5 (see below), for all $\pi' \models u_2 > u_1$, the trace set of $\mathsf{M}_{ex}[\pi']$ is equivalent to the one of $\mathsf{M}_{ex}[\pi]$, depicted in Fig.e 6 page 30.

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 $u_2 \leq u_1$.

5.3.3 Correctness

We show in Theorem 5 that *IM* preserves the equivalence of trace sets.

We first show that $\pi \models K$, where K is the result of *IM*. In the following, we consider that M is a PSTCSP model, and π is a parameter valuation.

Proposition 9 *Let* $K = IM(\mathsf{M}, \pi)$. *Then* $\pi \models K$.

Proof By construction, K is the result of intersecting all reachable states of S. Since there are no π-incompatible states in S after the **while** loop, then K contains only π-compatible inequalities. □

We will show in Proposition 10 the equivalence of trace sets for $\mathsf{M}[\pi']$ and $\mathsf{M}(K_{end})$, where $\pi' \models K$ and K_{end} is the value of constraint K before the end of the algorithm IM, i.e., the conjunction of π-incompatible inequalities. The following lemma will be used in the proof of Proposition 10.

Lemma 9 *Let* $K = IM(\mathsf{M}, \pi)$. *Let* K_{end} *be the value of constraint* K *before the end of the algorithm IM. Let* $\pi' \models K$. *Then for all runs of* $\mathsf{M}(K_{end})$ *reaching* (V, P, C), *we have* $\pi' \models C$.

Proof Consider a run of $\mathsf{M}(K_{end})$ reaching (V, P, C). We have $(V, P, C) \in S$, where S is the set of states at the end of the algorithm IM, since $S = Post^*_{\mathsf{M}}(\{(V_0, P_0, K)\})$. Moreover, we have $K = \bigcap_{(V,P,C)\in S} C{\downarrow_U}$. Hence $\pi' \models C{\downarrow_U}$. Hence $\pi' \models C$. □

Proposition 10 *Let* $K = IM(\mathsf{M}, \pi)$. *Let* K_{end} *be the value of constraint* K *before the end of the algorithm IM. Let* $\pi' \models K$. *Then the trace sets of* $\mathsf{M}(K_{end})$ *and* $\mathsf{M}[\pi']$ *are equivalent.*

Proof
1. We first show that each run of $\mathsf{M}(K_{end})$ is time-abstract equivalent to a run in $\mathsf{M}[\pi']$. Consider a run of $\mathsf{M}(K_{end})$ that ends in state (V, P, C). From Lemma 9, $\pi' \models C$. From Proposition 3, this run is time-abstract equivalent to a run in $\mathsf{M}[\pi']$.
2. We then show that each run of $\mathsf{M}[\pi']$ is time-abstract equivalent to a run in $\mathsf{M}(K_{end})$. Since $K \subseteq K_{end}$ and $\pi' \models K$, then $\pi' \models K_{end}$. Hence, from Proposition 4, each run of $\mathsf{M}[\pi']$ is time-abstract equivalent to a run in $\mathsf{M}(K_{end})$.

Theorem 5 *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 equivalent.*

Proof Item 1. From Proposition 9.
Item 2. Since $\pi \models K$ and $\pi' \models K$, their trace sets are both equivalent to the trace set of $\mathsf{M}(K_{end})$, by Proposition 10. Hence their trace sets are equivalent to each other.

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 not only the case for properties expressed using the Linear Temporal Logic (LTL) (Pnueli 1977), but also for properties expressed using the SE-LTL logic (Chaki et al. 2004), which is a linear temporal logic constituted by both atomic state propositions and events. Furthermore, since the programs modifying the values of the variables are only attached to events (and do not depend, e.g., on time), then the values of the variables are the same as well.

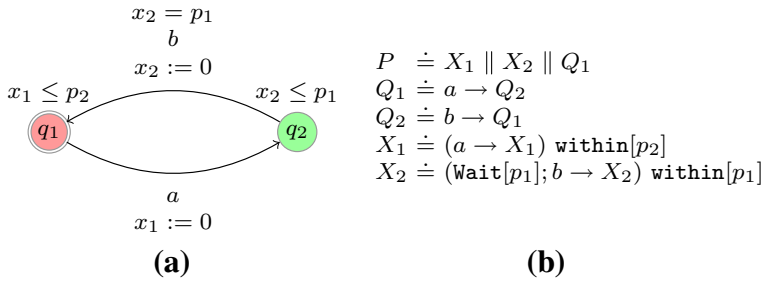

Fig. 9 Counter-example showing the non-termination of *IM*. **a** For PTA. **b** For PSTCSP

5.3.4 Termination

Similarly to the setting of PTA, termination of *IM* for PSTCSP is not guaranteed in the general case. Nevertheless, we give a criterion for termination.

First, consider an example of a PTA, depicted in Fig. 9a, for which *IM* (in the setting of PTA) does not terminate. Clocks can have any initial value (viz., $x_1 \geq 0 \wedge x_2 \geq 0$). Consider the following reference valuation π: $\{p_1 = 1, p_2 = 2\}$. Then no π-incompatible state is found, and an infinite number of states is generated, with constraints of the form $(i + 1) * p_2 \geq i * p_1$, with i growing without bound.

It is possible to translate this PTA to a PSTCSP model with no variables, as given in Fig. 9b. For this PSTCSP process, it can be shown that *IM* does not terminate, also generating constraints of the form $(i + 1) * p_2 \geq i * p_1$, with i infinitely growing.

Proposition 11 *IM*(M, π) *may not terminate in the general case.*

We now show that the non-recursivity of a model is a sufficient condition to ensure termination of *IM*. Note that this condition can be checked syntactically.

Proposition 12 (Termination condition for *IM*) *Let* M *be a PSTCSP model and* π *a valuation of its parameters.*

IM(M, π) *terminates if* M *is not recursive.*

Proof Since processes have no recursion, only a finite number of events can occur. Furthermore, since our semantics considers symbolic time elapsing (a construct $\texttt{Wait}[u]$ will lead to only one successor state in the LTS), the LTS will have a finite number of states.

Now, the inner loop of *IM* will necessarily terminate since, at one depth (encoded by variable i), only a finite number of new inequalities can be generated (this is true even for recursive models). As for the outer loop, since the LTS is finite, it has a bounded depth, and the algorithm will reach a given i for which no state has any successor. □

Despite Proposition 11, termination of the inverse method actually occurs for all our case studies (see Table 2), even those that do not meet the criterion of Proposition 12. For instance, *IM* terminates for Example 5, although it contains a recursive definition (because process P is defined using Q, and Q itself defined using Q). On the other

hand, a standard parametric reachability analysis (using *reachAll*) would go into an infinite loop, because the recursion is under the parameterized `interrupt` construct, where u_1 can be arbitrarily large when compared to u_2. This result is of particular interest since parameter synthesis is undecidable in general for PSTCSP. Exhibiting further syntactic criteria for the termination of *IM* is the subject of future work (see Sect. 7).

5.3.5 Non-completeness

Non-confluence We first show that *IM* for PSTCSP is non-confluent. That is, for a given PSTCSP model M and a given parameter valuation π, the result of $IM(\mathsf{M}, \pi)$ is not necessarily always the same.

Proposition 13 (Non-confluence) *There exist a PSTCSP model* M *and a valuation* π *such that two executions of* $IM(\mathsf{M}, \pi)$ *may output two different constraints.*

Proof Consider the following PSTCSP model $\mathsf{M} = (\emptyset, \{u_1, u_2, u_3\}, \emptyset, P, \mathit{True})$, with $P \doteq (((a \rightarrow P)\ \texttt{within}[u_2])\ \texttt{within}[u_1])\ \texttt{within}[u_3]$. Consider the following reference valuation $\pi = \{u_1 = 1, u_2 = 2, u_3 = 3\}$. Then, depending on the nondeterministic selection of the inequality ("J") to negate, the algorithm will return, besides the two "trivial" inequalities $u_1 \geq 0 \wedge u_2 \leq 0$, either $u_3 > u_1$ or $u_3 > u_2$. Of course, these two constraints are incomparable. □

Non-completeness From the result of non-confluence, we infer that *IM* is not complete; by non-completeness, we mean that there may exist parameter valuations outside of the constraint output by *IM* that still have the same trace set as the reference valuation π.

Corollary 1 (Completeness) *There exist a PSTCSP model* M*, a valuation* π*, and a valuation* π' *such that*

1. $\pi' \not\models IM(\mathsf{M}, \pi)$*, and*
2. $\mathsf{M}[\pi]$ *and* $\mathsf{M}[\pi']$ *have the same trace sets.*

Proof From Proposition 13, there exist a PSTCSP model M and a valuation π such that two runs of $IM(\mathsf{M}, \pi)$ may output two different constraints. Let K_1 and K_2 be two different constraints corresponding to a first and a second run of *IM*, respectively. Note that, from the correctness of *IM*, for any $\pi' \models K_1 \cup K_2$, the trace set of $\mathsf{M}[\pi']$ is always the same. Since $K_1 \neq K_2$, then either (1) $K_1 \subsetneq K_2$ or (2) $K_2 \subsetneq K_1$ or (3) $K_1 \not\subseteq K_2$ and $K_2 \not\subseteq K_1$, where $K_1 \subsetneq K_2$ denotes that K_1 is strictly included in K_2, that is $K_1 \subseteq K_2$ but we do not have that $K_1 = K_2$.

1. If $K_1 \subsetneq K_2$ then there exists $\pi' \models K_2$ such as $\pi' \not\models K_1$. Hence we have that $\pi' \not\models K_1$, and $\mathsf{M}[\pi]$ and $\mathsf{M}[\pi']$ have the same trace sets.
2. The case $K_2 \subsetneq K_1$ is dual.
3. If $K_1 \not\subseteq K_2$ and $K_2 \not\subseteq K_1$ then there exists $\pi' \models K_2$ such as $\pi' \not\models K_1$; hence the reasoning is the same.

Actually, non-completeness comes from the fact that the complete set of parameter valuations having the same trace set as $\mathsf{M}[\pi]$ can be non-convex (or even disjoint), whereas *IM* always outputs a convex constraint. Hence, in general, there may be no parameter valuation such that $IM(\mathsf{M}, \pi)$ outputs the complete set of parameter valuations having the same trace set as $\mathsf{M}[\pi]$.

Although *IM* is not complete, it remains interesting in practice. First, depending on the final application, engineers may not be interested in all possible parameter valuations guaranteeing a good behavior, but only in *some* of them. Indeed, in the end, the real system will be implemented using *one* parameter valuation only; what is important is to guarantee that the system behaves well around the parameter valuation (in case of small variability of the valuation in practice), but not necessarily to know all possible good parameter valuations. Second, even when the resulting constraint is not the maximum set of parameter valuations such that the system is time-abstract equivalent to the system under π, it always outputs a dense set of parameter valuations (see below). Hence, the resulting constraint gives a (possibly partial) measure of the robustness of the system that can help to understand the system behavior around the reference parameter valuation (see Sect. 5.3.6).

Density of the resulting constraint We show here that *IM* always outputs a constraint non-reduced to a point. A constraint reduced to a point means that the only valuation in K is the reference valuation π itself. This cannot happen in *IM*: indeed, the inequalities output come from the parametric reachability of the state space, following the semantics of PSTCSP. No constant can suddenly "appear" in the inequalities, since the model is fully parametric (see our ass in footnote 4). Hence, in the worst case, the inequalities output may be *equalities* of the form $u_1 = u_2 = \cdots = u_M$. Although this constraint is not dense (it is not a volume), it is not reduced to a point: this corresponds to an infinite (hyper)line in M dimensions passing by the origin and by π, and it means that the reference valuation can scale.

Furthermore, we note that a sufficient (but non-necessary) condition for *IM* to output a dense constraint K (non-reduced to a line) is that, in the reference valuation, all constants be different. (That is, the reference value for u_1 must be different from that of u_2, u_3, and so on.) Indeed, by correctness of *IM*, we have that $\pi \models K$, hence all inequalities in K are π-compatible. Since all parameter valuations are different in π, one cannot have an equality between two parameters in K, otherwise it would be π-incompatible.

In all our experiments (see Sect. 6), the set of parameter valuations synthesized is always dense, that is, not reduced to a (hyper)line.

5.3.6 Discussion

Advantages The efficiency of *IM* in practice comes from the fact that only a small portion of the state space is explored; 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 3). Although *IM* is not guaranteed to output the weakest constraint (i.e., the largest set of parameters), it often does (see Sect. 6.3); and it is always guaranteed to output a set of parameter valuations in $|U|$ dimensions, both non-null and non-reduced to a point.

Moreover, the preservation of time-abstract traces is particularly interesting for PSTCSP: much research in real-time systems concentrated in timed temporal properties, some of them allowing the use of clocks within the formulas. But because clocks are implicit in PSTCSP, it does not make much sense to consider properties based on a relationship between clocks, because they do not appear in the original model. As for properties based, e.g., on deadlines ("event a must occur no later than n units of time"), they can be easily encoded using an observer process. This process, in parallel with the rest of the system, fires an event *success* or *failure* depending on whether the deadline is met or not. Hence, this becomes a property on traces.

Last but not least, *IM* quantifies the *robustness* (see, e.g., Markey 2011; Bouyer et al. 2013) of the system: 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. Indeed, when a system is implemented, the values appearing in the system may not be exactly the same as the ones in the model that has been proven correct. The inverse method allows the designer to formally guarantee the correctness of the system not only for the reference valuation, but also for neighboring valuations.

Full coverage of the parametric space One may be interested in covering the whole parametric space. This is unlikely to obtain from a single call to the inverse method: this would mean that *IM* returns *True*, hence that all parameter valuations have exactly the same trace set. However, one can call several times the inverse method on different reference valuations, so as to obtain coverage of the whole parametric space with a set of *tiles*; tiles are convex constraints in which the trace set is always the same (two different tiles have two different trace sets in general). This is the purpose of the *behavioral cartography* defined for PTA in André and Fribourg (2010). It was shown that the behavioral cartography for PTA is usually not able to cover the whole, dense parametric space using a finite number of calls to the inverse method. However, it is possible to cover a set of arbitrarily tight points (e.g., integers, or multiple of 0.1 or 0.01, etc.) using a finite number of calls to *IM*. Although extending the behavioral cartography to PSTCSP is the object of future work, we believe that these properties would be the same for PSTCSP due to the close expressiveness between both formalisms.

Optimized implementation The inverse method for PSTCSP has been defined in Algorithm 3 in an algorithmic form, so as to ease the proofs. However, it is clearly not efficient that way, since the algorithm needs to start from the initial state every time an incompatible inequality J is selected (line 7). It was shown in André (2010) that it is instead equivalent to just add the negated inequality $\neg J$ to all the reached states (in S). Note that no state will become inconsistent (that is, with an unsatisfiable constraint) because $\neg J$ is π-compatible by construction (that is, $\pi \models \neg J$), and so are the constraints of all the reached states. It was this optimized version that we implemented in PSyHCoS.

6 Implementation and experiments

6.1 Implementation within PSyHCoS

This work has been implemented in a tool, PSyHCoS (standing for *Parameter SYnthesis for Hierarchical COncurrent Systems*), which is a self-contained framework to support composing, simulating and automatically verifying parametric hierarchical concurrent real-time systems (André et al. 2013b). PSyHCoS comes with user friendly interfaces, a feature-rich model editor and an animated simulator.

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

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 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 in the form of a pair (process id, constraint id), both stored as strings. This is an implementation choice. Although not everything is represented using strings in PSyHCoS, some identifiers use strings. An advantage of the string representation is that the constraint equality test (when checking whether this new state has been met before) reduces to string equality.

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

6.2 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 clocks. Merging these states may lead to an exponential decrease of the number of states. Hence, we implemented a technique of *state normalization* based on anonymization of the clock: 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.

Table 2 Application of algorithms for parameter synthesis using PSyHCoS

Case study	*reachAll*					*reachAll+*			*IM*			*IM+*		
	$\|U\|$	$\|X\|$	$\|S\|$	$\|T\|$	t	$\|S\|$	$\|T\|$	t	$\|S\|$	$\|T\|$	t	$\|S\|$	$\|T\|$	t
M_{ex}	2	2	8	14	0.008	8	14	0.006	3	5	0.004	3	5	0.005
M'_{ex}	2	2	8	14	0.008	8	14	0.006	8	14	0.016	8	14	0.008
Bridge	4	2	–	–	OoM	–	–	OoM	2.8k	5.5k	253	2.8k	5.5k	455
$Fischer_4$	2	4	–	–	OoM	–	–	OoM	11k	31k	41.9	2k	5.8k	8.65
$Fischer_5$	2	5	–	–	OoM	–	–	OoM	133k	447k	1,176	13k	44k	84.5
$Fischer_6$	2	6	–	–	OoM	–	–	OoM	–	–	OoM	86k	342k	1,144
Jobshop	8	2	14k	20k	21.0	12k	17k	18.1	1,112	1,902	17.1	877	1,497	22.8
RCS_2	4	4	52	64	0.038	52	64	0.059	52	64	0.091	52	64	0.147
RCS_3	4	4	233	296	0.186	233	296	0.300	233	296	0.310	233	296	0.513
RCS_4	4	4	1,070	1,374	1.74	1,070	1,374	1.58	1,070	1,374	1.40	1,070	1,374	2.38
RCS_5	4	4	5.6k	7.2k	10.5	5.6k	7.2k	9.54	5.6k	7.2k	7.83	5.6k	7.2k	16.7
RCS_6	4	4	34k	43k	91.7	34k	43k	54.5	34k	43k	60.4	34k	43k	91.3
TrAHV	6	6	7.2k	13k	14.2	7.2k	13k	15.8	227	321	0.555	227	321	0.655

6.3 Experiments

We give in Table 2 the example name, the number $|U|$ of parameters, the maximum number $|X|$ of clocks required,[10] and, for each algorithm, the number $|S|$ (resp. $|T|$) of states (resp. transitions),[11] and the computation time t on a Windows XP desktop computer with an Intel Quad Core 2.4 GHz processor with 4 GiB memory. "OoM" in a cell denotes "out of memory". Binaries, sources, models and results are available in PSyHCoS' Web page.[12]

6.3.1 Description of the models

Bridge is a classical bridge crossing problem for four persons within 17 min. $Fischer_i$ is the mutual exclusion protocol for i processes. Jobshop is a scheduling problem (Fribourg et al. 2012). TrAHV is the train example from Alur et al. (1993). RCS_i is a railway control system with i trains (Yi et al. 1995).

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 in fact acyclic). When *reachAll* does not terminate (Bridge, Fischer), *IM* is interesting because it synthesizes constraints even for infinite symbolic

[10] In theory, nothing guarantees that the maximum number of clocks is the same for *reachAll*, *reachAll+*, *IM* and *IM+*. Nevertheless, since it is always the same for all experiments, we factor it to save some space in the columns.

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

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

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).

6.3.2 Interpretation of the resulting constraint

First, the synthesized constraint by *IM* solves the good parameter problem, and may even output *all* possible correct parameter valuations. For instance, the constraint synthesized for Fischer ($\delta < \gamma$) is known to be the weakest constraint guaranteeing mutual exclusion. (We used $\delta = 3$ and $\gamma = 4$ as reference valuation.)

Second, it always gives a quantitative measure of the system robustness, 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. As opposed to 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` and `within` can be directly removed, whereas `interrupt` and `timeout` can be replaced with non-deterministic choice. In that case, techniques for untimed systems (usually more efficient) can be applied to the system. Although this refinement might be checked for one particular parameter valuation using refinement techniques designed for STCSP, we prove it here for *any* possible parameter valuation; indeed, since the discrete behavior is the same for any parameter valuation satisfying *True*, hence for any parameter valuation, the timed constructs can be safely dropped.

6.3.3 Performance

The number of clocks is significantly smaller than equivalent models for PTA for some case studies: for instance, the Bridge case study would obviously require four clocks because there are four independent processes in parallel. Similarly, the RCS_i case study would require at least i clocks, one for each 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 Bengtsson and Yi (2004), 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.

Table 2 shows 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 occurs (e.g., Bridge), i.e., when the state space is not reduced using this technique, 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* runs out of memory (Fischer_6).

The implementation of PSTCSP within PSyHCoS seems to be efficient: some case studies (e.g., Fischer_5, Fischer_6, RCS_6) handle several dozens or hundreds of thousands symbolic states in a reasonable amount of time, which, to the best of our knowledge, is unheard of for parametric timed frameworks: We did not find publications mentioning tools for parameter synthesis handling more than a few thousands states.

6.3.4 Comparison with IMITATOR

To the best of our knowledge, 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 PTA and Petri Nets are rather straightforward, their translation into process algebra is much trickier. We actually tried to perform a comparison with IMITATOR 2.5, the implementation of the inverse method for PTA (André et al. 2012a). Unfortunately, this comparison (performed using the same machine with Ubuntu 11.10 64 bits) did not give accurate results. Indeed, the (manual) translation of models from PTSCP to PTA (and conversely) is difficult: in all cases, the tool for which the model was initially designed performs much better than the tool that runs on a translated model. For example, Jobshop (8.96 s) and TrAHV (0.097 s) are quicker on IMITATOR, for which they were initially designed. Conversely, IMITATOR does not terminate for Fischer_i for all i because of the explicit representation of the clocks in PTA (constraints of the form $x_2 \geq j * \epsilon + x_1$, with j infinitely growing, are generated), whereas the implicit clocks in PSTCSP prevent this. We did not find a better way to encode an equivalent PTA model of our Fischer example for PSTCSP. Other models (Bridge, RCS_i) are too large to be manually translated. An automated efficient translation mechanism, that could ease such a comparison, is the subject of future work. Unfortunately, nothing guarantees that encoding complex data structures from PSTCSP into PTA is practicable. And, in any case, some features specific to PSTCSP, such as hierarchy and implicit clocks, would be lost by the translation.

6.3.5 Application to scheduling problems

The case studies we considered include protocols (Fischer_i), common puzzles (Bridge) and train control systems (RCS_i, TrAHV). These models naturally fit with process algebras, as similar case studies have been used in the literature in the setting of (timed or untimed) process algebras. We also included one scheduling problem (more precisely a jobshop problem) (Fribourg et al. 2012). PSTCSP can indeed model parametric schedulability problems, that consists in deciding whether there exists a parameter valuation for which a given number of tasks can be executed on a set of processors within a given (constant) time. Recall that schedulability problems can be modeled (among other formalisms) using timed automata or timed automata extended with stopwatches (Adbeddaïm and Maler 2002; Adbeddaïm et al. 2006). Stopwatches add the power of *stopping* time, which is necessary to handle preemptive jobs, where a task with a higher priority can (temporarily) stop another one, that will resume once the higher priority task is completed. PSTCSP cannot express the power of stop-

ping time, and hence is not able (in general) to model task preemption; however, PSTCSP can model non-preemptive schedulability problems. Indeed, these problems can be expressed and solved using parametric timed automata without strict constraints (see, e.g., Fribourg et al. 2012), that have a smaller expressiveness than PSTCSP (see Sect. 4.3).

7 Conclusion and future work

7.1 Conclusion

We introduced here 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 we give a simple sufficient termination condition. We also introduce an algorithm *3VPsynthesis* synthesizing a set of parameter valuations guaranteeing that at least one good state and no bad state is reachable according to a 3-value property. Our implementation within PSyHCoS leads to efficient parameter synthesis, and handles more than 100,000 reachable symbolic states in a very reasonable amount of time.

7.2 Future work

Among the theoretical questions is the decidability of the membership problem for non-regular PSTCSP. This problem is actually not related to parameter synthesis, but rather to the existence or not of a finite abstraction of the state space of a non-regular STCSP model; this could be solved using techniques for infinite-state systems.

We wish to improve the state space representation, following the lines of the optimization of Sect. 6.2, and develop further state space reduction techniques, such as the merging technique developed in André et al. (2013a).

Beyond the algorithms developed in Sect. 5, we are interested in further algorithms for parameter synthesis. An interesting problem is the existence of (at least) one parameter valuation for which at least one unbounded run is guaranteed to occur (and that we could name "EG-emptiness" problem). Other synthesis algorithms should also be developed or adapted, for instance following the lines of algorithms for PTA (Knapik and Penczek 2012; Jovanovic et al. 2013). Furthermore, parametric refinement checking is the subject of future work.

It is also interesting to note that, although *IM* does not terminate in the general case, it does for all of our case studies, even when they do not meet the termination criterion of Proposition 12. As a consequence, it would be of interest to exhibit a syntactical criterion that ensures termination in PSTCSP.

Improving our implementation PSyHCoS, and in particular implementing Algorithm *3VPsynthesis*, is also in our agenda.

Finally, although PSTCSP provides the designer with a rather high-level syntax, an interesting future work will be to define higher-level patterns dedicated to specific applications such as scheduling. For example, defining a set of patterns modeling scheduling processes, schedulers with or without preemption, is of interest so as to allow designers to model a system by just assembling predefined PSTCSP language blocks, in the line of Khatib et al. (2001) (for models) or André (2013) (for properties).

Acknowledgments We are grateful to the anonymous reviewers for their very constructive comments, and to Zhu Huiquan for solving several implementation issues in our model checking tool PSyHCoS. This manuscript also benefited from discussions with Didier Lime. We thank Emmanuelle Encrenaz and Laurent Fribourg for discussions regarding the comparison between IMITATOR and HYTECH. Yang Liu is supported by "Formal Verification on Cloud" project under Grant No: M4081155.020 and "Verification of Security Protocol Implementations" project under Grant No: M4080996.020. 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". All four authors are supported by STIC Asie project "CATS (Compositional Analysis of Timed Systems)". Étienne André is partially supported by the ANR national research program PACS (ANR-2014).

Appendix: A firing rules for PSTCSP

Given a program *program* and a valuation V, the valuation obtained by executing *program* with V is denoted as $program(V)$. Let $active(V, P)$ 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). We give below all firing rules for PSTCSP.

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

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

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

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

$$\frac{V \nvDash b}{(V, \texttt{if}\ b\ \texttt{then}\ \{P\}\ \texttt{else}\ \{Q\}, C) \overset{\tau}{\rightsquigarrow} (V, Q, C^{\uparrow})}(co3)$$

$$\frac{(V, P, C) \overset{e}{\rightsquigarrow} (V', P', C')}{(V, P \,\square\, Q, C) \overset{e}{\rightsquigarrow} (V', P', C' \wedge idle(Q))}(aex1)$$

$$\frac{(V, Q, C) \overset{e}{\rightsquigarrow} (V', Q', C)}{(V, P \,\square\, Q, C) \overset{e}{\rightsquigarrow} (V', Q', C' \wedge idle(P))}(aex2)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

References

Aceto L, Bouyer P, Burgueño A, Larsen KG (1998a) The power of reachability testing for timed automata. In: Arvind V, Ramanujam R (eds) FSTTCS, lecture notes in computer science, vol 1530. Springer, Berlin, pp 245–256

Aceto L, Burgueño A, Larsen KG (1998b) Model checking via reachability testing for timed automata. In: Steffen B (ed) TACAS, lecture notes in computer science, vol 1384. Springer, Berlin, pp 263–280

Adbeddaïm Y, Maler O (2002) Preemptive job-shop scheduling using stopwatch automata. In: Katoen JP, Stevens P (eds) TACAS, lecture notes in computer science, vol 2280. Springer, Berlin, pp 113–126

Adbeddaïm Y, Asarin E, Maler O (2006) Scheduling with timed automata. Theor Comput Sci 354(2):272–300

Akshay S, Hélouët L, Jard C, Reynier PA (2012) Robustness of time Petri nets under guard enlargement. RP, lecture notes in computer science, vol 7550. Springer, Berlin, pp 92–106

Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

Alur R, Madhusudan P (2004) Decision problems for timed automata: a survey. In: Bernardo M, Corradini F (eds) SFM-RT, lecture notes in computer science, vol 3185. Springer, Berlin, pp 1–24

Alur R, Henzinger TA, Vardi MY (1993) Parametric real-time reasoning. In: Kosaraju SR, Johnson DS, Aggarwal A (eds) Proceedings of the twenty-fifth annual ACM symposium on theory of computing, 16–18 May 1993, San Diego, CA. ACM

André É (2010) An inverse method for the synthesis of timing parameters in concurrent systems. Ph.d. thesis, Laboratoire Spécification et Vérification, ENS Cachan, France

André É (2013) Observer patterns for real-time systems. In: Liu Y, Martin A (eds) ICECCS. IEEE Computer Society, Washington, DC, pp 125–134

André É, Soulat R (2013) The inverse method. FOCUS series in computer engineering and information technology. ISTE Ltd and John Wiley & Sons Inc

André É, Chatain T, Encrenaz E, Fribourg L (2009) An inverse method for parametric timed automata. Int J Found Comput Sci 20(5):819–836

André É, Fribourg L (2010) Behavioral cartography of timed automata. In: Kučera A, Potapov I (eds) RP, lecture notes in computer science, vol 6227. Springer, Berlin, pp 76–90

André É, Fribourg L, Kühne U, Soulat R (2012a) IMITATOR 2.5: a tool for analyzing robustness in scheduling problems. In: FM, lecture notes in computer science, vol 7436. Springer, Berlin, pp 33–36
André É, Liu Y, Sun J, Dong JS (2012b) Parameter synthesis for hierarchical concurrent real-time systems. In: Perseil I, Pouzet M, Breitman K (eds) ICECCS. IEEE Computer Society, Washington, DC, pp 253–262
André É, Fribourg L, Soulat R (2013a) Merge and conquer: state merging in parametric timed automata. In: Hung DV, Ogawa M (eds) ATVA, lecture notes in computer science, vol 8172. Springer, Berlin, pp 381–396
André É, Liu Y, Sun J, Dong JS, Lin SW (2013b) PSyHCoS: parameter synthesis for hierarchical concurrent real-time systems. In: Sharygina N, Veith H (eds) CAV, lecture notes in computer science, vol 8044. Springer, Berlin, pp 984–989
André É, Petrucci L, Pellegrino G (2013c) Precise robustness analysis of time Petri nets with inhibitor arcs. In: Braberman V, Fribourg L (eds) FORMATS, lecture notes in computer science, vol 8053. Springer, Berlin, pp 1–15
Annichini A, Bouajjani A, Sighireanu M (2001) TReX: a tool for reachability analysis of complex systems. CAV, lecture notes in computer science, vol 2102. Springer, Berlin, pp 368–372
Asarin E, Maler O, Pnueli A (1998) On discretization of delays in timed automata and digital circuits. CONCUR, lecture notes in computer science, vol 1466. Springer, Berlin, pp 470–484
Bagnara R, Hill PM, Zaffanella E (2008) The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci Comput Program 72(1–2):3–21
Baier C, Katoen JP (2008) Principles of model checking. MIT Press, Cambridge, MA
Behrmann G, Larsen KG, Rasmussen JI (2005) Beyond liveness: efficient parameter synthesis for time bounded liveness. FORMATS, lecture notes in computer science, vol 3829. Springer, Berlin, pp 81–94
Bengtsson J, Yi W (2004) Timed automata: semantics, algorithms and tools. Lectures on concurrency and Petri Nets, lecture notes in computer science, vol 3098. Springer, Berlin, pp 87–124
Bérard B, Gastin P, Petit A (1996) On the power of non-observable actions in timed automata. STACS, lecture notes in computer science, vol 1046. Springer, Berlin, pp 257–268
Bérard B, Petit A, Diekert V, Gastin P (1998) Characterization of the expressive power of silent transitions in timed automata. Fundam Inform 36:145–182
Bouyer P, Larsen KG, Markey N, Sankur O, Thrane CR (2011) Timed automata can always be made implementable. In: Katoen JP, König B (eds) CONCUR, lecture notes in computer science, vol 6901. Springer, Berlin, pp 76–91
Bouyer P, Markey N, Sankur O (2012) Robust reachability in timed automata: a game-based approach. In: Czumaj A, Mehlhorn K, Pitts AM, Wattenhofer R (eds) ICALP 2012, lecture notes in computer science, vol 7392. Springer, Berlin, pp 128–140
Bouyer P, Markey N, Sankur O (2013) Robustness in timed automata. In: Abdulla PA, Potapov I (eds) RP, lecture notes in computer science, vol 8169. Springer, Berlin, pp 1–18
Bozzelli L, La Torre S (2009) Decision problems for lower/upper bound parametric timed automata. Form Methods Syst Des 35(2):121–151
Chaki S, Clarke EM, Ouaknine J, Sharygina N, Sinha N (2004) State/event-based software model checking. iFM, lecture notes in computer science, vol 2999. Springer, Berlin, pp 128–147
Chevallier R, Encrenaz-Tiphène E, Fribourg L, Xu W (2009) Timed verification of the generic architecture of a memory circuit using parametric timed automata. Form Methods Syst Des 34(1):59–81
Clarisó R, Cortadella J (2007) The octahedron abstract domain. Sci Comput Program 64(1):115–139
Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. CAV. Springer, Berlin, pp 154–169
Collomb-Annichini A, Sighireanu M (2001) Parameterized reachability analysis of the IEEE 1394 root contention protocol using TReX. In: RT-TOOLS
D'Argenio PR, Katoen JP, Ruys TC, Tretmans J (1997) The bounded retransmission protocol must be on time!. TACAS, lecture notes in computer science, vol 1217. Springer, Berlin, pp 416–431
Davies J (1993) Specification and proof in real-time CSP. Cambridge University Press, Cambridge
Dong JS, Hao P, Qin S, Sun J, Yi W (2008) Timed automata patterns. IEEE Trans Softw Eng 34(6):844–859
Encrenaz E, Fribourg L (2008) Time separation of events: an inverse method. LIX, electronic notes in theoretical computer science, vol 209. Elsevier Science Publishers, Palaiseau, pp 135–148
Fidge CJ, Hayes IJ, Watson G (1999) The deadline command. IEE Proc Softw 146(2):104–111

Fribourg L, Lesens D, Moro P, Soulat R (2012) Robustness analysis for scheduling problems using the inverse method. TIME. IEEE Computer Society Press, Washington, DC, pp 73–80
Henzinger TA, Wong-Toi H (1995) Using HyTech to synthesize control parameters for a steam boiler. Formal methods for industrial applications, lecture notes in computer science, vol 1165. Springer, Berlin, pp 265–282
Henzinger TA, Nicollin X, Sifakis J, Yovine S (1994) Symbolic model checking for real-time systems. Inf Comput 111(2):193–244
Henzinger TA, Ho PH, Wong-Toi H (1997) HyTech: a model checker for hybrid systems. Softw Tools Technol Transf 1:460–463
Hoare C (1985) Communicating sequential processes. Prentice-Hall, International series in computer science
Hoenicke J, Olderog ER (2002) Combining specification techniques for processes, data and time. iFM, lecture notes in computer science, vol 2335. Springer, Berlin, pp 245–266
Hune T, Romijn J, Stoelinga M, Vaandrager FW (2002) Linear parametric model checking of timed automata. J Log Algebr Program 52–53:183–220
Jaubert R, Reynier PA (2011) Quantitative robustness analysis of flat timed automata. In: Hofmann M (ed) FoSSaCS, lecture notes in computer science, vol 6604. Springer, Berlin, pp 229–244
Jovanovic A, Lime D, Roux OH (2013) Integer parameter synthesis for timed automata. In: Piterman N, Smolka SA (eds) TACAS, lecture notes in computer science, vol 7795. Springer, Berlin, pp 401–415
Khatib L, Muscettola N, Havelund K (2001) Mapping temporal planning constraints into timed automata. TIME. IEEE Computer Society, Washington, DC, pp 21–27
Knapik M, Penczek W (2012) Bounded model checking for parametric timed automata. Trans Petri Nets Other Models Concurr 5:141–159
Kwak HH, Lee I, Philippou A, Choi JY, Sokolsky O (1998) Symbolic schedulability analysis of real-time systems. IEEE RTSS. IEEE Computer Society, Washington, DC, pp 409–418
Kwak HH, Lee I, Sokolsky O (1999) Parametric approach to the specification and analysis of real-time system designs based on ACSR-VP. Electron Notes Theor Comput Sci 25:38–49
Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a nutshell. Int J Softw Tools Technol Transf 1(1–2):134–152
Lime D, Roux OH, Seidner C, Traonouez LM (2009) Romeo: a parametric model-checker for Petri nets with stopwatches. In: Kowalewski S, Philippou A (eds) TACAS, lecture notes in computer science, vol 5505. Springer, Berlin, pp 54–57
Mahony BP, Dong JS (1999) Overview of the semantics of TCOZ. iFM. Springer, Berlin, pp 66–85
Markey N (2011) Robustness in real-time systems. SIES. IEEE Computer Society Press, Washington, DC, pp 28–34
Minsky ML (1967) Computation: finite and infinite machines. Prentice-Hall Inc, Upper Saddle River, NJ
Ouaknine J, Worrell J (2003a) Revisiting digitization, robustness, and decidability for timed automata. LICS. IEEE Computer Society, Washington, DC, pp 198–207
Ouaknine J, Worrell J (2003b) Timed CSP = closed timed ϵ-automata. Nord J Comput 10(2):99–133
Pnueli A (1977) The temporal logic of programs. FOCS. IEEE Computer Society, Washington, DC, pp 46–57
Qin S, Dong JS, Chin WN (2003) A semantic foundation for TCOZ in unifying theories of programming. FME, lecture notes in computer science, vol 2805. Springer, Berlin, pp 321–340
Roscoe AW (2001) Compiling shared variable programs into CSP. In: PROGRESS workshop
Sankur O (2013) Shrinktech: a tool for the robustness analysis of timed automata. In: Sharygina N, Veith H (eds) CAV, lecture notes in computer science, vol 8044. Springer, Berlin, pp 1006–1012
Schneider S (2000) Concurrent and real-time systems. Wiley, Hoboken, NJ
Schrijver A (1986) Theory of linear and integer programming. Wiley, Hoboken, NJ
Sun J, Liu Y, Dong JS, Chen C (2009a) Integrating specification and programs for system modeling and verification. In: Chin WN, Qin S (eds) TASE. IEEE Computer Society, Washington, DC, pp 127–135
Sun J, Liu Y, Dong JS, Pang J (2009b) PAT: towards flexible verification under fairness. CAV, lecture notes in computer science, vol 5643. Springer, Berlin, pp 709–714
Sun J, Liu Y, Dong JS, Liu Y, Shi L, André É (2013) Modeling and verifying hierarchical real-time systems using stateful timed CSP. ACM Trans Softw Eng Methodol 22(1):3.1–3.29. doi:10.1145/2430536.2430537
Traonouez LM, Lime D, Roux OH (2009) Parametric model-checking of stopwatch Petri nets. J Univers Comput Sci 15(17):3273–3304

Traonouez LM (2012) A parametric counterexample refinement approach for robust timed specifications. FIT, electronic proceedings in theoretical computer science 87:17–33
Yi W, Pettersson P, Daniels M (1995) Automatic verification of real-time communicating systems by constraint-solving. FORTE, IFIP conference proceedings, vol 6. Chapman & Hall, London, pp 243–258
Yoneda T, Kitai T, Myers CJ (2002) Automatic derivation of timing constraints by failure analysis. CAV, lecture notes in computer science, vol 2404. Springer, Berlin, pp 195–208

Étienne André received the master degree (with honors) from the Université de Rennes 1 (France) in 2007, and the Ph.D. degree in computer science from École Normale Supérieure de Cachan (France) in 2010. He was then a research follow in Prof. Dong Jin Song's team in the National University of Singapore for 9 months. Since September 2011, he has been an Associate Professor in the Laboratoire d'Informatique de Paris Nord, in the University of Paris 13 (Sorbonne Paris Cité) in France. His current research interests include the specification and the parametric verification of real-time concurrent systems. More details about his research background can be found at http://lipn.univ-paris13.fr/~andre/.

Yang Liu graduated in 2005 with a Bachelor of Computing in the National University of Singapore (NUS). In 2010, he obtained his Ph.D. and continued with his post doctoral work in NUS. Since 2012, he joined Nanyang Technological University as an Assistant Professor. His research focuses on software engineering, formal methods and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the-art model checker, Process Analysis Toolkit.

Jun Sun received Bachelor and Ph.D. degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship in School of Computing of NUS. Since 2010, he joined Singapore University of Technology and Design (SUTD) as an Assistant Professor. He was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, formal methods and cyber-security. He is the co-founder of the PAT model checker. To this date, he has more than 100 journal articles or peer-reviewed conference papers. He is the program co-Chair of FM'14 and the general chair of a number of international conferences.

Jin-Song Dong received Bachelor and Ph.D. degrees in Computing from University of Queensland in 1992 and 1996. From 1995–1998, he was Research Scientist at CSIRO in Australia. Since 1998 he has been in the School of Computing at the National University of Singapore (NUS) where he is currently Associate Professor and a member of PhD supervisors at NUS Graduate School. He is on the editorial board of Formal Aspects of Computing and Innovations in Systems and Software Engineering. His research interests include formal methods, software engineering, pervasive computing and semantic technologies.