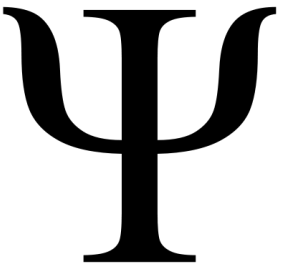
PSyHCoS

User Manual



Current Version
May 6, 2012

Étienne André[1], Yang Liu[2], Jun Sun[3] and Jin-Song Dong[4]

[1] LIPN, CNRS UMR 7030, Université Paris 13, France
[2] Temasek Laboratories, National University of Singapore
[3] Singapore University of Technology and Design
[4] School of Computing, National University of Singapore

Abstract

Real-time systems are often hard to control, due to their complicated structures, quantitative time factors and even unknown values. Checking the correctness of a system for one particular value for each delay does not give any information for other values. It is hence interesting to reason parametrically, by considering that the delays are parameters (unknown constants) and synthesize a constraint guaranteeing a correct behavior. We present here the user manual of PSyHCoS, a tool for analyzing parametric real-time systems specified using the hierarchical modeling language PSTCSP. PSyHCoS supports several algorithms for parameter synthesis and model checking, as well as state space reduction techniques. It comes with a friendly user interface that can be used to edit, simulate and verify PSTCSP models.

Contents

Chapter 1

Introduction

Ensuring the correctness of safety-critical systems, involving complex data structures with timing requirements, is crucial. The correctness of such real-time systems usually depends on the values of some timing delays. Checking the correctness for one particular value for each delay is usually not sufficient for two reasons. Firstly, value for the delays are not always known, and one may precisely want to *find* some values for which the system behaves well. Secondly, even if the system is proved correct for a reference set of values, one has no guarantee that the correctness holds other values around the reference one, which is also known as the *robustness* (see, e.g., [9]) of the system. Hence, it is interesting to consider that the delays are unknown constants, or *parameters*, and synthesize constraints on these parameters to guarantee a correct behavior.

In this work, we present the user manual of PSyHCoS (Parameter SYnthesis for Hierarchical COncurrent Systems) [1]. This tool performs parameter synthesis and parametric model checking for Parametric Stateful Timed CSP (PSTCSP) [4], a parametric extension of Stateful Timed CSP [12], itself an extension of Timed CSP (see, e.g., [11]). PSTCSP offers an intuitive syntax for designing and analyzing hierarchical real-time concurrent systems involving shared variables, complex data structures, and user defined programs.

The expressiveness of PSTCSP is incomparable with Parametric Timed Automata (PTAs) [2], which are an extension of finite state automata with clocks (variables increasing linearly) and parameters. Different from PTAs, clocks are *implicit* in PSTCSP, thus avoiding the designer to write manually clocks constraints, which is error-prone. Another advantage of PSTCSP over PTAs is the ability to easily define hierarchical systems. Sub-systems can be defined independently and referred in the main system. Hierarchy may allow one to handle refinement as well as closed ("black box" or "gray box") systems.

To be best of our knowledge, PSyHCoS is the first tool that synthesizes timing parameters for real-time systems handling both hierarchy and concurrency.

Chapter 2

System Architecture

PSyHCoS offers a complete GUI for the design and the verification of PSTCSP models. We give in Figure 2.1 the architecture of PSyHCoS. Models are described using an intuitive text syntax (see Chapter 3).

PSyHCoS also features a simulator (see Figure 6.2) that can be used for step-by-step analyses, or for the graphical representation of the state space under the form of a directed graph. Beyond educational purpose, the simulator is interesting for better understanding (or debugging) models.

Among the verification algorithms, PSyHCoS first implements the inverse method *IM* for PSTCSP [4]. *IM* takes as input a PSTCSP model as well as a reference valuation for all the parameters; it synthesizes a set of parameters under the form of a convex constraint K, that guarantees the same time-abstract behavior (sequences of actions) as for the reference valuation. A major advantage is that K gives a quantitative measure of the robustness of the system w.r.t. variations of the timing delays. In particular, all linear time properties that hold for the reference valuation hold for any valuation in K. Although parameter synthesis for PSTCSP is proved undecidable (same as for PTAs), *IM* always terminates in practice. Also, a full reachability algorithm *reachAll* is implemented in PSyHCoS in order to compare optimization techniques. Other classical model checking algorithms (such as LTL, deadlock freeness, or refinement checking) are also available. Data structures and functions can be designed within PSTCSP models directly using the C# syntax.

Internal representation and optimization techniques The semantics is defined as a labeled transition system, where the states in the transition system consist of a process and a constraint on clocks and parameters [4]. In PSyHCoS, each state is implemented under the form of a pair (process id, constraint id), both under the form of a string. Although some processing is needed each time a new state is computed, the constraint equality test reduces to string equality. Also, a string format is flexible – which is necessary as we are dealing with hierarchical systems so that different states may have very different system architecture.

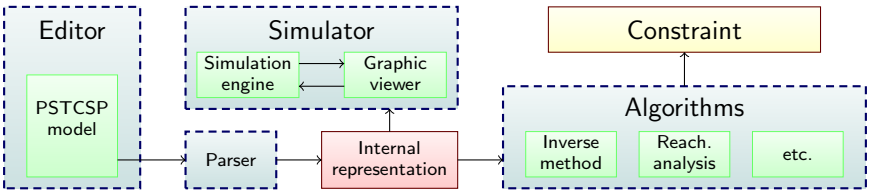


Figure 2.1: Architecture of PSyHCoS

PSyHCoS implements a state-space reduction technique for PSTCSP introduced in [4], which merges states that are equal except the *names* of the clocks. This may lead to an exponential diminution of the number of states, at the cost of several nontrivial operations (lists and strings sorting). The optimized version of *reachAll* (resp. *IM*) is denoted by *reachAll+* (resp. *IM+*).

Implementation PSyHCoS is implemented in C#, based on Microsoft .NET framework. The constraint solver is based on the PPL library [5]. It can run on major OS, including Linux, Unix, Mac or Windows. The tool is available under the GNU General Public License in [1].

Chapter 3

Input Syntax for Models

3.1 Standard Features from PSTCSP

3.1.1 Abstract Syntax of PSTCSP

We first recall here from [4] the syntax of PSTCSP. A process models the control logic of the system using a rich set of process constructs. A process P is defined by the grammar in Figure 3.1, where $u \in U$.[1] Processes marked with * allow the use of parameters instead of timing constants in STCSP. $\mathcal{P}$ denotes the set of all possible processes.

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

Figure 3.1: Syntax of PSTCSP processes

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

Definition 3.1.1 *A Parametric Stateful Timed CSP (or PSTCSP) model is a tuple* $\mathsf{M} = (Var, U, V_0, P, K_0)$ *where* $Var \subset \mathcal{V}ar$, $U \subset \mathcal{U}$, V_0 *is the initial variable valuation,* $P \in \mathcal{P}$ *is a process, and* $K_0 \in \mathcal{K}_U$ *is an initial constraint.*

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

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

Untimed constructs We first briefly describe the untimed constructs, which are identical to the ones in STCSP, and very close to the ones of CSP. Process `Stop` does nothing but idling. Process `Skip` terminates, possibly after idling for some time. Process $e \rightarrow P$ engages in event e first and then behaves as P. Note that e may serve as a synchronization barrier, if combined with parallel composition. In order to seamlessly integrate data operations, sequential programs may be attached with events. Process $a\{program\} \rightarrow P$ performs data operation a (i.e., executing the sequential *program* whilst generating event a) and then behaves as P. The program may be a simple procedure updating data variables (e.g., $a\{v_1 := 5; v_2 := 3\}$, where $v_1, v_2 \in Var$) or a more complicated sequential program. A conditional choice is written as `if` (b) $\{P\}$ `else` $\{Q\}$. If b is true, then it behaves as P; otherwise it behaves as Q. Process $P|Q$ offers an unconditional choice[2] between P and Q. Process $P;Q$ behaves as P until P terminates and then behaves as Q immediately. $P \setminus E$ hides occurrences of events in E. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization (following CSP rules [8]) or shared variables.

Timed Constructs We now explain the parametric timed constructs.

- Given a parameter u, process `Wait`$[u]$ idles for an unknown (constant) number of u time units.
- In process P `timeout`$[u]$ Q, the *first* observable event of P shall occur before u time units elapse. Otherwise, Q takes over control after exactly u time units.
- Process P `interrupt`$[u]$ Q behaves exactly as P until u time units, and then Q takes over. In contrast to P `timeout`$[u]$ Q, P may engage in

[2] For simplicity, external and internal choices from the classic CSP [8] have not been defined in [4]. Nevertheless, both constructions are implemented (see Section 3.2.1), and used in our case studies.

multiple observable events before it is interrupted. Also note that Q will be executed in any case, whereas in P `timeout`$[u]$ Q, process Q will only be executed if no observable event occurs before u time units.

- Process P `within`$[u]$ must react within an unknown number of u time units, i.e., an observable event must be engaged by process P within u time units.
- Process P `deadline`$[u]$ constrains P to terminate, possibly after engaging in multiple observable events, before u time units.

Discussion on deadline The deadline timed construct intuitively means that a process must terminate within a certain amount of time. Different definitions of deadline actually appear in the literature. In [7], 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 [10], the authors use Unifying Theory of Programming in order to formalize the semantics of TCOZ. As in [7], they consider that the deadline imposes a *timing constraint* on P, which thus requires the computation of P to be finished within the time mentioned in the deadline.

Different from [10, 7], we here choose to stick to the semantics of STCSP [12] and consider a deadline semantics as an *attempt* to terminate a process before a certain time. If the process does not terminate before the deadline, it is just stopped[3].

Syntactic sugar Urgent event prefixing [6], written as $e \twoheadrightarrow P$, is defined as $(e \rightarrow P)$ `within`$[0]$, i.e., e must occur as soon as it is enabled.

Also note that some timed constructs can be defined using other timed constructs. For instance, `within` can be defined using `deadline`.

3.1.2 Input Syntax of PSTCSP in PSyHCoS

We give in Figure 3.2 the mapping between PSTCSP syntax and the ASCII-based PSyHCoS syntax. Since some parts of PSyHCoS rely on the PAT model checker [13, 14], PSyHCoS's input syntax is rather similar to the STCSP input syntax in the RTS module of PAT.

[3] Remark that, in that case, time elapsing may be stopped too.

`Stop`	`Stop`
`Skip`	`Skip`
$e \rightarrow P$	`e -> P`
$a\{program\} \rightarrow P$	`a { program } -> P`
`if` (b) $\{P\}$ `else` $\{Q\}$	`if b { P } else { Q }`
$P \vert Q$	`P [] Q`
$P \setminus E$	`P \ { e1,e2,e3 }`
$P; Q$	`P ; Q`
$P \parallel Q$	`P || Q`
`Wait`$[u]$	`Wait[u]`
P `timeout`$[u]$ Q	`P timeout[u] Q`
P `interrupt`$[u]$ Q	`P interrupt[u] Q`
P `within`$[u]$	`P within[u]`
P `deadline`$[u]$	`P deadline[u]`
$P = Q$	`P = Q`

Figure 3.2: Syntax of PSTCSP processes in PSyHCoS

3.2 Syntactic Extensions of PSTCSP

3.2.1 Additional Operations

We give in Figure 3.3 additional operations on PSTCSP not strictly defined in [4], but implemented in PSyHCoS.

`P [*] Q`	External choice
`P <> Q`	Internal choice
`if (cond1) {P} else if (cond2) {Q} else {M}`	Conditional choice (nested)
`ifa (cond) {P} else {Q}`	Atomic conditional choice
`ifb (cond) {P}`	Blocking conditional choice
`[cond] P`	Guarded process
`P ||| Q`	Interleaving
`atomic { P }`	Atomicity

Figure 3.3: Syntax of advanced PSTCSP processes in PSyHCoS

3.2.2 Advanced Constructs

Constants Constants can be defined as follows: `#define max 5;`

Variables Variables can be defined as follows: `var knight = 0;` The value is optional. Example: `var knight;`

Parameters Parameters can be defined as follows: `parameter Delta;` Recall that parameters are unknown constants. As a consequence, they cannot be set, nor updated.

Data structures PSyHCoS supports advanced data structures such as lists, arrays, matrices, and user-defined structures. A fixed-size array may be defined as follows:

`var board = [3, 5, 6, 0, 2, 7, 8, 4, 1];` where board is the array name and its initial value is specified as the sequence, e.g., `board[0] = 3`. The following defines an array of size 3.

`var leader[3];`

All elements in the array are initialized to be 0.

Multi-dimensional arrays may also be defined as follows.

`var matrix[3*N][10];`

Note: To assign values to specific elements in an array, you can use event prefix. For example:

`P() = a { matrix[1][9] = 0 } -> Skip;`

To ease the modeling, PSyHCoS allows users to define any data structures and use them in PSTCSP models. The following shows the syntax.

```
var<Type> x ; //default constructor of Type class will be called.
var<Type> x = new Type(1, 2); //constructor with two int parameters
will be called.
```

3.3 Example: Fischer Mutual Exclusion

We use here an example from [4] in order to present PSyHCoS's syntax, and show PSTCSP's intuitive modeling facilities. Fischer's mutual exclusion algorithm is modeled by a process *FME* making use of a set of timing parameters $U = \{\delta, \gamma\}$, and a set of variables $Var = \{turn, cnt\}$. The *turn* variable indicates which process attempted to access the critical section most recently. The *cnt* variable counts the number of processes accessing the critical section. Initial valuation 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[4].

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

[4]Note that this is not the real ASCII-based PSyHCoS syntax.

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

We give below the PSyHCoS-based syntax for *FME*.

```
1  #define N 3;
2  #define Idle -1;
3
4  var turn = Idle;
5  var counter = 0;
6  parameter Delta;
7  parameter Epsilon;
8
9  proc(i) = ifb(turn == Idle) { Active(i) };
10 Active(i) = ((update.i{turn=i} -> Wait[Epsilon]) within [Delta]);
11    if (turn == i) {
12       cs.i{counter++} -> exit.i{counter--; turn=Idle}->proc(i)
13    } else {
14       proc(i)
15    };
16
17 FME = ||| i:{0..N-1}@proc(i);
```

Chapter 4

Input Syntax for Analysis

4.1 Parameter Synthesis Using the Inverse Method

PSyHCoS implements the inverse method *IM* for PSTCSP [4], initially defined for PTAs [3]. *IM* takes as input a PSTCSP model as well as a reference valuation for all the parameters; it synthesizes a set of parameters under the form of a convex constraint K, that guarantees the same time-abstract behavior (sequences of actions) as for the reference valuation. A major advantage is that K gives a quantitative measure of the robustness of the system w.r.t. variations of the timing delays. In particular, all linear time properties that hold for the reference valuation hold for any valuation in K. Although parameter synthesis for PSTCSP is proved undecidable (same as for PTAs), *IM* always terminates in practice.

The input syntax is of the following form:

```
1 #synthesize process with u1 = 2, u2 = 4;
```

4.2 Full Reachability Analysis

A full reachability algorithm *reachAll* is implemented in PSyHCoS in order to compare optimization techniques (see experiments in [4]). It computes the whole set of reachable states; note that this algorithm may not always terminate.

The input syntax is of the following form:

```
1 #synthesize process reachesall;
```

Returned statistics include the number of states and transitions, the memory occupation and the computation time. All states can also be returned in a text form (for a graphical form, use the simulator).

4.3 Model Checking

Various model checking algorithms are implemented within PSyHCoS.

An example of syntax (for deadlock freeness checking) is given below:

```
1 #assert P deadlockfree;
```

4.4 Example: Fischer Mutual Exclusion

Recall example from Section 3.3. A classical parameter synthesis problem is to find values for δ and γ such that mutual exclusion is guaranteed.

This is achieved by the following call to the inverse method:

```
1 #synthesize FME with Epsilon = 3, Delta = 4;
```

The constraint synthesized by PSyHCoS for Fischer is $\delta < \gamma$, viz., the weakest constraint known to guarantee mutual exclusion.

4.5 State Space Optimization

As explained in [4], some states in PSTCSP considered as different are actually equivalent. Consider the following two states:

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

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

It is obvious that $s_1 = s_2$, except the *names* of the clocks. Merging these states may lead to an exponential diminution of the number of states.

Hence, we implemented a technique of *state normalization*: First, the clocks in the process are renamed so that the first one (from left to right) is named x_1, the second x_2, and so on. Second, the variables in the constraint are swapped accordingly. This technique solves this problem at the cost of several nontrivial operations (lists and strings sorting). We denote by *reachAll+* (resp. *IM+*) the version of *reachAll* (resp. *IM*) using this technique.

This technique can be used for most algorithm; the choice is available in the verification window.

Chapter 5

A Complete Example

We give below an entire example of input model for PSyHCoS, with commands for verification and parameter synthesis.

A full list of benchmarks is available in [1].

```
 parameter pDelta;
parameter pEpsilon;

#define N 3;
#define Delta 3;
#define Epsilon 4;
#define Idle -1;


var x = Idle;
var counter;


//untimed version
uP(i) = ifb(x == Idle) {
          update.i{x = i} ->
           if (x == i) {
             cs.i{counter++} -> exit.i{counter--; x=Idle} -> uP(i)
           } else {
               uP(i)
           }
       };
uFischersProtocol = ||| i:{0..N-1}@uP(i);
#synthesize uFischersProtocol reachesall;


//timed version
P(i) = ifb(x == Idle) {
       ((update.i{x = i} -> Wait[Epsilon]) within[Delta]);
          if (x == i) {
             cs.i{counter++} -> exit.i{counter--; x=Idle} -> P(i)
          } else {
              P(i)
          }
       };
```

```
36
37 FischersProtocol = ||| i:{0..N-1}@P(i);
38 #synthesize FischersProtocol reachesall;
39
40
41 //parametric timed version
42 PP(i) = ifb(x == Idle) {
43       ((update.i{x = i} -> Wait[pEpsilon]) within[pDelta]);
44          if (x == i) {
45             cs.i{counter++} -> exit.i{counter--; x=Idle} -> PP(i)
46          } else {
47               PP(i)
48          }
49       };
50
51 FischersProtocolParam = ||| i:{0..N-1}@PP(i);
52
53 #synthesize FischersProtocolParam with
54    pEpsilon = 3,
55    pDelta = 4
56 ;
57 #synthesize FischersProtocolParam reachesall;
```

Chapter 6

Screenshots

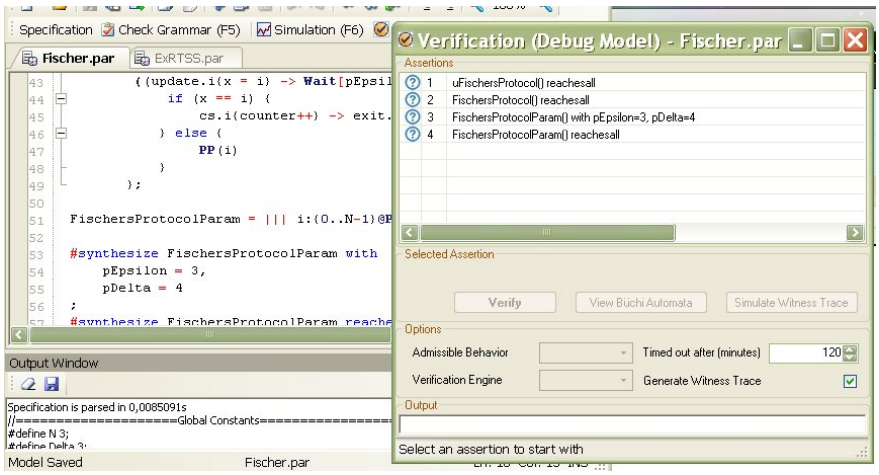


Figure 6.1: Editor and verification engine

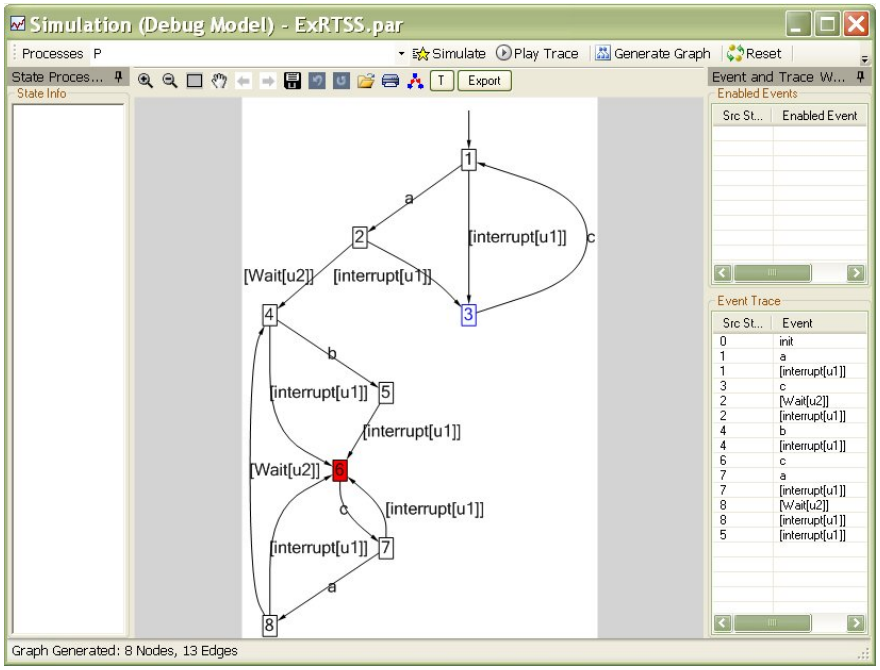


Figure 6.2: Simulation interface

Bibliography

[1] http://www-lipn.univ-paris13.fr/~andre/software/PSyHCoS/. 2, 4, 13

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

[3] É. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *Int. J. of Found. of Comp. Sci.*, 20(5):819–836, 2009. 11

[4] É. André, Y. Liu, J. Sun, and J. S. Dong. Parameter synthesis for hierarchical concurrent real-time systems. In *ICECCS'12*, 2012. To appear (author version available on . Andr's Web page). 2, 3, 4, 5, 6, 8, 9, 11, 12

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008. 4

[6] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993. 7

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

[8] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985. 6

[9] N. Markey. Robustness in real-time systems. In *SIES'11*, pages 28–34. IEEE Computer Society Press, 2011. 2

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

[11] S. Schneider. *Concurrent and Real-time Systems*. John Wiley and Sons, 2000. 2

[12] J. Sun, Y. Liu, J. Dong, and X. Zhang. Verifying stateful timed CSP using implicit clocks and zone abstraction. In *ICFEM'09*, volume 5885 of *LNCS*, pages 581–600, 2009. 2, 7

[13] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV'09*, volume 5643 of *LNCS*. Springer, 2009. 7

[14] J. S. Yang Liu and J. S. Dong. PAT 3: An extensible architecture for building multi-domain model checkers. In *ISSRE 2011*, 2011. Accepted. 7