IMITATOR User Manual
Version 2.2

Étienne André

August 3, 2010

Abstract

We present here the user manual of Imitator, a tool implementing the "inverse method" in the framework of parametric timed automata: given a reference valuation of the parameters, its generates a constraint such that the system behaves the same as under the reference valuation in terms of traces, i.e., alternating sequences of locations and actions. This is useful for safely relaxing some values of the reference valuation, and optimizing timing bounds of the system. Besides the inverse method, Imitator also implements the "behavioral cartography algorithm", allowing to solve the following good parameters problem: find a set of valuations within a given rectangle for which the system behaves well. We give here the installation requirements and the launching commands of Imitator, as well as the source code of a toy example.

Contents

1 Introduction **3**

2 Behavioral Cartography **4**
2.1 Parametric Timed Automata 4
2.2 A Motivating Example 5
2.3 The Inverse Method 6
2.4 The Behavioral Cartography Algorithm 8

3 Implementation **10**
3.1 Inputs and Outputs 10
3.2 Structure and Implementation 11
3.3 Features 11

4 How to Use Imitator **12**
4.1 Installation 12
4.2 The IMITATOR Input File 12
4.2.1 Variables 13
4.2.2 Parametric Timed Automata 13
4.2.3 Initial region and π_0 13
4.3 Calling IMITATOR 13
4.3.1 Reachability Analysis 13
4.3.2 Inverse Method 14
4.3.3 Cartography 14
4.3.4 Options 14
4.3.5 Examples of Calls 17

5 Example: SR-latch **18**
5.1 Parametric Reachability Analysis 18
5.2 Behavioral Cartography Algorithm 18

6 Final Remarks **22**

A Source Code of the Example **25**
A.1 Main Input File 25
A.2 V_0 File 29

B Complete Grammar **30**
B.1 Grammar of the Input File 30
B.1.1 Automata Descriptions 30
B.1.2 Initial State 32
B.2 Reserved Words 33

1 Introduction

Timed automata [2] are finite control automata equipped with *clocks*, which are real-valued variables which increase uniformly. This model is useful for reasoning about real-time systems with a dense representation of time, because one can specify quantitatively the interval of time during which the transitions can occur, using timing bounds. However, the behavior of a system is very sensitive to the values of these bounds, and it is rather difficult to find their correct values. One can check the correctness of the system for one particular timing value for each timing bound (using model checkers such as, e.g., UPPAAL [20]), but this does not give any information for other values. Actually, testing the correctness of the system for all the timing values, even in a bounded interval, would require an infinite number of calls to the model checker, because those timing bounds can have real (or rational) values.

It is therefore interesting to reason *parametrically*, by considering that these bounds are unknown constants, or parameters, and try to synthesize a *constraint* (i.e., a conjunction of linear inequalities) on these parameters which will guarantee a correct behavior of the system. Such automata are called *parametric timed automata* (PTA) [3].

The Good Parameters Problem for Timed Automata. In order to find correct values of the parameters, we are interested in solving the following *good parameters problem*, as defined in [12] in the framework of linear hybrid automata: "Given a PTA $\mathcal{A}$ and a rectangular parameter domain V_0, what is the largest set of parameter values within V_0 for which $\mathcal{A}$ is safe?"

The parameter design problem for timed automata (and more generally, for linear hybrid automata) was formulated in [15], where a straightforward solution is given, based on the generation of the whole parametric state space until a fixpoint is reached. Unfortunately, in all but the most simple cases, this is is prohibitively expensive due, in particular, to the brute exploration of the whole parametric state space.

In [12], they propose an extension based on the *counterexample guided abstraction refinement* (CEGAR, [11]). When finding a counterexample, the system obtains constraints on the parameters that *make* the counterexample infeasible. When all the counterexamples have been eliminated, the resulting constraints describe a set of parameters for which the system is safe.

The tool IMITATOR presented in this paper is based on the *inverse method* [4], which supposes given a "good instantiation" π_0 of the parameters that one wants to generalize. More precisely, IMITATOR generates a constraint K_0 on the parameters that corresponds to an infinite dense set of valuations such that, for all instantiation π of parameters in this set, the behavior of the timed automaton $\mathcal{A}$ is *(time-abstract) equivalent* to the behavior of $\mathcal{A}$ under π_0, in the sense that they have the same trace sets. This is useful to relax timing bounds, and gives a criterion of *robustness*.

Moreover, IMITATOR implements the *behavioral cartography algorithm* [5], which generates a constraint on the parameters ("tile") by calling the inverse method for each integer point located within a given rectangle V_0. This algorithm allows us to partition the parametric space into a subset of "good" tiles (which correspond to "good behaviors") and a subset of "bad" ones. Often in

practice, what is covered is not the *bounded* and *integer* subspace of the parameter rectangle, but two major extensions: first, not only the integer points but a major part of the dense set of *real-valued* points of the rectangle is covered by the tiles; second, the tiles are often unbounded w.r.t. several dimensions (hence are infinite), and cover most of the parametric space beyond V_0, thus giving a solution to the good parameters problem.

IMITATOR is a new version of IMITATOR [8], a prototype written in Python implementing the inverse method, and calling the model checker HYTECH [14]. IMITATOR has been entirely rewritten and is a now standalone tool, making use of the APRON library [18] and the Parma Polyhedra Library [9]. Compared to IMITATOR, the computation timings of IMITATOR have dramatically decreased. Moreover, IMITATOR offers new features, such as the implementation of the behavioral cartography algorithm, the generation of the trace sets of the models, and a graphical output. We present in this paper the new features and optimizations of IMITATOR, as well as a range of case studies.

This tool is being developed at LSV, ENS Cachan, France. The tool can be downloaded on its Web page[1], as well as a bunch of case studies.

Organization of this user manual. We first recall the framework of Parametric Timed Automata, the inverse method algorithm and the behavioral cartography algorithm in Section 2. We also apply those two algorithms to the Root Contention Protocol. We then introduce IMITATOR in Section 3 and give details on its internal structure and its various features. We give in Section 4 the most useful in order to install and use IMITATOR. We present in Section 5 a full example, and show the application of the inverse method and the behavioral cartography algorithm using IMITATOR. We give final remarks in Section 6. We also give in Appendix A the source code of the example, and in Appendix B the full grammar of IMITATOR.

2 Behavioral Cartography of Timed Automata

In this section, we first briefly recall the framework of Parametric Timed Automata (Section 2.1). We then introduce the Root Contention Protocol as a motivating example (Section 2.2). We then recall the inverse method algorithm described in [4] (Section 2.3), and the behavioral cartography algorithm described in [5] (Section 2.4).

2.1 Parametric Timed Automata

We use in this paper the same formalism as in [5]. Throughout this paper, we assume a fixed set $X = \{x_1, \dots, x_H\}$ of *clocks*, and a fixed set $P = \{p_1, \dots, p_M\}$ of *parameters*.

We assume familiarity with timed automata [2]. Parametric timed automata are an extension of timed automata to the parametric case, allowing within guards and invariants the use of parameters in place of constants [3]. Given a set of clocks X and a set of parameters P, a *parametric timed automaton (PTA)* $\mathcal{A}$ is a 6-tuple of the form $\mathcal{A} = (\Sigma, Q, q_0, K, I, \rightarrow)$, where Σ is a finite

[1] http://www.lsv.ens-cachan.fr/~andre/IMITATOR2/

set of actions, Q is a finite set of locations, $q_0 \in Q$ is the initial location, K is a constraint on the parameters, I is the invariant assigning to every $q \in Q$ a constraint I_q on the clocks and the parameters, and $\rightarrow$ is a step relation consisting in elements of the form (q, g, a, ρ, q') where $q, q' \in Q$, $a \in \Sigma$, $\rho \subseteq X$ is a set of clocks to be reset by the step, and g (the step guard) is a constraint on the clocks and the parameters.

In the sequel, we consider the PTA $\mathcal{A} = (\Sigma, Q, q_0, K, I, \rightarrow)$. We simply denote this PTA by $\mathcal{A}(K)$, in order to emphasize the fact that only K will change in $\mathcal{A}$. For every parameter valuation $\pi = (\pi_1, \ldots, \pi_M)$, $\mathcal{A}[\pi]$ denotes the PTA $\mathcal{A}(K)$, where K is $\bigwedge_{i=1}^{M} p_i = \pi_i$. This corresponds to the PTA obtained from $\mathcal{A}$ by substituting every occurrence of a parameter p_i by constant π_i in the guards and invariants. We say that p_i is *instantiated* with π_i. Note that, as all parameters are instantiated, $\mathcal{A}[\pi]$ is a standard timed automaton. (Strictly speaking, $\mathcal{A}[\pi]$ is only a timed automaton if π assigns an integer to each parameter.)

A *(symbolic) state* s of $\mathcal{A}(K)$ is a couple (q, C) where q is a location, and C a constraint on the clocks and the parameters.

A *run* of $\mathcal{A}(K)$ is an alternating sequence of states and actions of the form $s_0 \overset{a_0}{\Rightarrow} s_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} s_m$, such that for all $i = 0, \ldots, m-1$, $a_i \in \Sigma$ and $s_i \overset{a_i}{\Rightarrow} s_{i+1}$ is a step of $\mathcal{A}(K)$.

Given a PTA $\mathcal{A}$ and a run R of $\mathcal{A}$ of the form $(q_0, C_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (q_m, C_m)$, the *trace associated to* R is the alternating sequence of locations and actions $q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} q_m$. The *trace set of* $\mathcal{A}$ refers to the set of traces associated to the runs of $\mathcal{A}$.

In the following, we are interested in verifying properties on the trace set of $\mathcal{A}$. For example, given a predefined set of "bad locations", a reachability property is satisfied by a trace if this trace never contains a bad location; such a trace is "good" w.r.t. this reachability property. A trace can also be said to be "good" if a given action always occurs before another one within the trace (see [5]). Actually, the good behaviors that can be captured with trace sets are relevant to *linear-time properties* [10], which can express properties more general than reachability properties.

Formally, given a property on traces, we say that a trace is *good* if it satisfies the property, and *bad* otherwise. Likewise, we say that a trace set is *good* if all its traces are good, and bad otherwise.

2.2 A Motivating Example

Consider the Root Contention Protocol of the IEEE 1394 ("FireWire") High Performance Serial Bus, studied in the parametric framework in [17]. As described in [17], this protocol is part of a leader election protocol in the physical layer of the IEEE 1394 standard, which is used to break symmetry between two nodes contending to be the root of a tree, spanned in the network technology. The protocol consists in first drawing a random number (0 or 1), then waiting for some time according to the result drawn, followed by the sending of a message to the contending neighbor. This is repeated by both nodes until one of them receives a message before sending one, at which point the root is appointed.

We consider the following five timing parameters:

- *rc_fast_min* (resp. *rc_fast_max*) gives the lower (resp. upper) bound to the waiting time of a node that has drawn 1;
- *rc_slow_min* (resp. *rc_slow_max*) gives the lower (resp. upper) bound to the waiting time of a node that has drawn 0;
- *delay* indicates the maximum delay of signals sent between the two contending nodes.

Those timing parameters are bound by the following implicit constraint:

$$rc_fast_min \leq rc_fast_max \quad \wedge \quad rc_slow_min \leq rc_slow_max$$

We consider the following instantiation π_0 of the parameters given in [19] (and rescaled from the original IEEE valuation)[2]:

$$\begin{array}{ccc} rc_fast_min = 76 & rc_fast_max = 85 & delay = 36 \\ rc_slow_min = 159 & rc_slow_max = 167 & \end{array}$$

Let us consider the following property $Prop_1$: "The minimum probability that a leader is elected within five rounds is greater or equal to 0.75." We consider that the system behaves well if this property is satisfied[3]. We can show that, for the reference valuation π_0, the system behaves well, i.e., its trace set is a good trace set.

We are now looking for other valuations "around" π_0 such that the system has the same good behavior. More formally, we are interested in solving the following inverse problem:

The Inverse Problem
Given a PTA $\mathcal{A}$ and a reference valuation π_0, generate a constraint K_0 such that

1. $\pi_0 \models K_0$, and
2. for all $\pi_1, \pi_2 \in K_0$, the trace sets of $\mathcal{A}[\pi_1]$ and $\mathcal{A}[\pi_2]$ are equal.

2.3 The Inverse Method

We recall here the inverse method algorithm $IM(\mathcal{A}, \pi)$, as defined in [4], which solves the inverse problem. Starting with $K = \texttt{true}$, we iteratively compute a growing set of reachable states. When a π-*incompatible* state (q, 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 and $\neg J$ is added to K. The procedure is then started again with this new K, and so on, until no new state is computed. We finally return the intersection

[2] The IEEE reference instantiation is given in *ns* but, due to the rescaling, we omit the unit here.

[3] Recall that we model here the Root Contention Protocol using (non-probabilistic) parametric timed automata, in which the random choice between 0 and 1 is modeled by non-determinism, as in [17]. Therefore, in order to compute probabilities, we need to consider a model involving probabilistic timed automata (i.e., timed automata augmented with probabilities). It can be shown (see [6]) that the minimum or maximum probability of satisfying a given property can be computed directly from the (non-probabilistic) trace set. As a consequence, the property that we consider for this example is purely a trace property.

of the projection onto the parameters of all the constraints associated to the reachable states.

The output of *IM* is a *behavioral tile* in the following sense: A constraint K is said to be a *behavioral tile* (or more simply a *tile*), if for all $\pi_1, \pi_2 \in K$, the trace sets of $\mathcal{A}[\pi_1]$ and $\mathcal{A}[\pi_2]$ are equal. Note that a tile corresponds to a convex and dense set of real-valued points.

Given a tile K, the trace set of $\mathcal{A}(K)$ will be simply referred to as "the trace set of K". Note that such a trace set is a possibly infinite set of traces.

Algorithm 1: $IM(\mathcal{A}, \pi)$

input : A PTA $\mathcal{A}$ of initial state $s_0 = (q_0, K_0)$
input : Valuation π of the parameters
output: Constraint K on the parameters

```
1  i ← 0 ;  K ← true ;  S ← {s0}
2  while true do
3      while there are π-incompatible states in S do
4          Select a π-incompatible state (q, C) of S (i.e., s.t. π ⊭ C) ;
5          Select a π-incompatible J in (∃X : C) (i.e., s.t. π ⊭ J) ;
6          K ← K ∧ ¬J ;
7          S ← ⋃_{j=0}^{i} Post^j_{A(K)}({s0}) ;
8      if Post_{A(K)}(S) ⊑ S then return K ← ⋂_{(q,C)∈S} (∃X : C)
9      i ← i + 1 ;
10     S ← S ∪ Post_{A(K)}(S) ;                // S = ⋃_{j=0}^{i} Post^j_{A(K)}({s0})
```

The algorithm *IM* is given in Algorithm 1. Given a linear inequality J of the form $e < e'$ (resp. $e \leq e'$), the expression $\neg J$ denotes the negation of J and corresponds to the linear inequality $e' \leq e$ (resp. $e' < e$). Given a constraint C on the clocks and the parameters, the expression $\exists X : C$ denotes the constraint on the parameters obtained from C after elimination of the clocks, i.e., $\{\pi \mid \pi \models C\}$. We define $Post^i_{\mathcal{A}(K)}(S)$ as the set of states reachable from S in exactly i steps, and $Post^*_{\mathcal{A}(K)}(S)$ as the set of all states reachable from S in $\mathcal{A}(K)$ (i.e., $Post^*_{\mathcal{A}(K)}(S) = \bigcup_{i \geq 0} Post^i_{\mathcal{A}(K)}(S)$). Given two sets of states S and S', we write $S \sqsubseteq S'$ iff $\forall s \in S, \exists s' \in S'$ s.t. $s = s'$.

Application to the Root Contention Protocol. Applying the inverse method algorithm to the PTA modeling the Root Contention Protocol described in Section 2.2 and the reference valuation π_0, the following constraints is generated by IMITATOR in 2.3 seconds:

$$rc_slow_min > 2 * delay + rc_fast_max \quad \wedge \quad rc_fast_min > 2 * delay$$

By definition of the inverse problem, this constraint corresponds to parameter valuations having exactly the same behavior (i.e., exactly the same trace set) as for π_0. However, there may be other parameter valuations having different good behaviors (i.e., different good trace sets). Finding those other parameter valuations is the purpose of the next section.

2.4 The Behavioral Cartography Algorithm

We recall here the behavioral cartography algorithm, as defined in [5]. By iterating the above inverse method *IM* over all the *integer* points of a rectangle V_0 (of which there are a finite number), one is able to decompose (most of) the parametric space included into V_0 into behavioral tiles. Formally:

Algorithm 2: Behavioral Cartography Algorithm $BC(\mathcal{A}, V_0)$

input : A PTA $\mathcal{A}$, a finite rectangle $V_0 \subseteq \mathbb{R}^M_{\geq 0}$
output: *Tiling*: list of tiles (initially empty)

1 **repeat**
2 select an integer point $\pi \in V_0$;
3 **if** π *does not belong to any tile of Tiling* **then**
4   Add $IM(\mathcal{A}, \pi)$ to *Tiling*;
5 **until** *Tiling contains all the integer points of* V_0 ;

Note that two tiles with distinct trace sets are necessarily disjoint. On the other hand, two tiles with the same trace sets may overlap.

In practice, most of (the real-valued space of) V_0 is covered by *Tiling*. Furthermore, the space covered by *Tiling* often largely exceeds the limits of V_0 (see [5] for a sufficient condition of full coverage of the parametric space).

Partition Between Good and Bad Tiles. According to a given property on traces one wants to check, it is possible to partition trace sets between good and bad. Once one has decided which tiles are good and which ones are bad, one can partition the rectangle V_0 into a good subspace (union of good tiles) and a bad subspace (union of bad tiles).

Advantages. First, the cartography itself does not depend on the property one wants to check. Only the partition between good and bad tiles involves the considered property. Moreover, the algorithm is interesting because one does not need to compute the set of all the reachable states. On the contrary, each call to the inverse method algorithm quickly reduces the state space by removing the "bad" states. This allows us to overcome the state space explosion problem, which prevents other methods, such as the computation of the whole set of reachable states (and then the intersection with the bad states), to terminate in practice.

Application to the Root Contention Protocol. To find other valuations of the parameters for which the system still behaves well, we compute a cartography of the Root Contention Protocol with the following V_0: $rc_slow_min \in [140, 200]$, $rc_slow_max \in [140, 200]$ and $delay \in [1, 50]$. The two other parameters remain constant, as in π_0.

The cartography computed by Imitator is given in Figure 1. For the sake of clarity, we project onto *delay* and *rc_slow_min*. In each tile, the parameter *rc_slow_max* is only bound by the implicit constraint $rc_slow_min \leq rc_slow_max$.

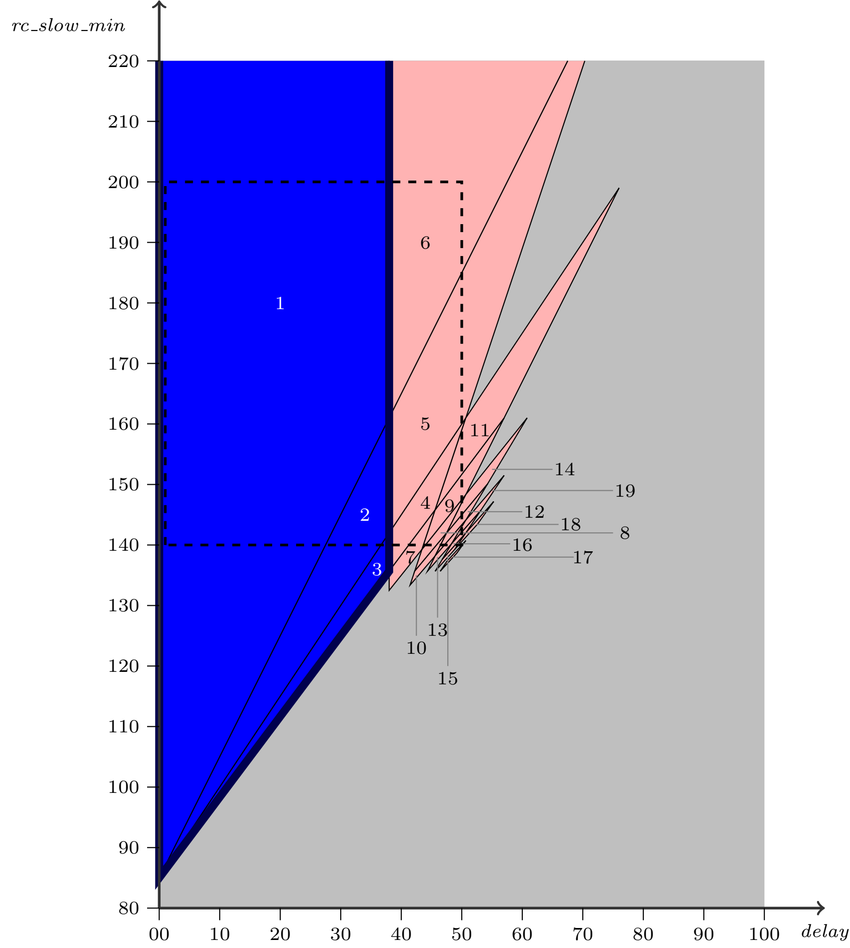

Figure 1: Behavioral cartography of the Root Contention Protocol according to *delay* and *rc_slow_min*

Note that tiles 1 and 6 are infinite towards dimension *rc_slow_min*, and all tiles are infinite towards dimension *rc_slow_max*. Moreover, although all the integer points within V_0 are covered (from the algorithm), note that the real-valued part of V_0 is not fully covered, because there are some "holes" (real-valued zones without integer points) in the lower right corner. An example of point which not covered by the cartography is $delay = 50$, $rc_slow_min = 140.4$ and $rc_slow_max = 141$.

It can be shown that tiles 1 to 3 correspond to *good* tiles, whereas the other tiles correspond to *bad tiles*[4]. As a consequence, the solution to the good parameters problem for this example corresponds to the parameter valuations included in tiles 1, 2 and 3. The corresponding constraint is the following one

[4]Note that what IMITATOR computes is a list of tiles as well as the associated trace sets. We then use an external tool (here, PRISM) in order to verify for each tile whether the considered property $Prop_1$ is satisfied or not. Note that this step could be easily integrated to IMITATOR in automatic manner (see final remarks in Section 6).

(recall that $rc_fast_min = 76$ and $rc_fast_max = 85$):

$$2{*}rc_slow_min > 3{*}delay{+}170 \quad \wedge \quad delay < 38 \quad \wedge \quad rc_slow_min \leq rc_slow_max$$

3 General Structure and Implementation

3.1 Inputs and Outputs

The input syntax of IMITATOR to describe the network of PTAs modeling the system is given in [7], and is very close to the HYTECH syntax. Actually, all standard HYTECH files describing only PTAs (and not more general systems like linear hybrid automata[1]) can be analyzed directly by IMITATOR with very minor changes[5].

Inverse Method. When calling IMITATOR to apply the inverse method algorithm, the tool takes as input two files, one describing the network of PTAs modeling the system, and the other describing the reference valuation. As depicted in Figure 2, it synthesizes a constraint on the parameters solving the inverse problem, as well as the corresponding trace set under a graphical form. The description of all the parametric reachable states is also returned.

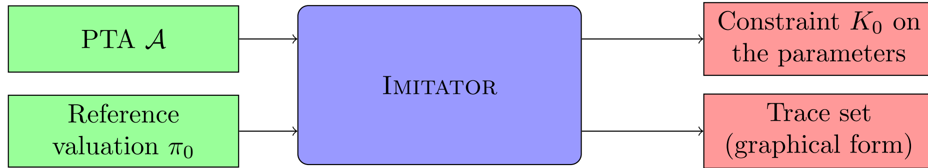

Figure 2: IMITATOR inputs and outputs in inverse method mode

Behavioral Cartography Algorithm. When calling IMITATOR to apply the behavioral cartography algorithm, the tool takes as an input two files, one describing the network of PTAs modeling the system, and the other describing the reference rectangle, i.e., the bounds to consider for each parameter. As depicted in Figure 3, it synthesizes a list of tiles, as well as the trace set corresponding to each tile under a graphical form. The description of all the parametric reachable states for each tile is also returned.

Figure 3: IMITATOR inputs and outputs in behavioral cartography mode

[5] An interface to accept as well files given using the PHAVer syntax is currently being implemented.

3.2 Structure and Implementation

Structure. Whereas IMITATOR was a prototype written in Python calling HYTECH for the computation of the *Post* operation, IMITATOR is a standalone tool written in OCaml. The *Post* operation has been fully implemented, and the inverse method algorithm entirely rewritten. As depicted in Figure 4, IMITATOR makes use of an external library for manipulating convex polyhedra. Depending on the user's preference, IMITATOR can call either the NewPolka library, available in the APRON library [18], or the Parma Polyhedra Library (PPL) [9]. The trace sets are output under a graphical form using the DOT module of the graph visualization software Graphviz.

IMITATOR contains about 9000 lines of code, and its development took about 6 man-months.

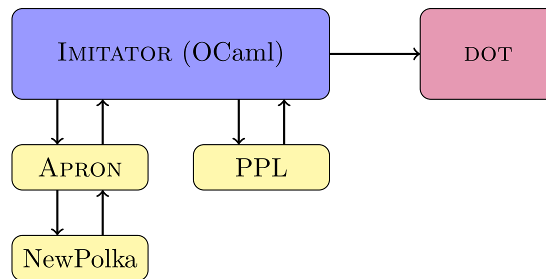

Figure 4: IMITATOR internal structure

Internal Representation. States are represented using a triple (q, v, C) made of the current location q in each automaton, a value for each discrete variable[6] v, and a constraint C on the clocks and the parameters. In order to optimize the test of equality between a new computed state and the set of states computed previously, the states are stored in a hash table as follows: to a given key (q, v) of the hash table, we associate a list of constraints $C_1, \ldots, C_n$, corresponding to the n states (q, v, C_1), ..., (q, v, C_n).

Note that, unlike HYTECH, IMITATOR uses exact arithmetics with unlimited precision.

Contrarily to HYTECH which performs an *a priori* static composition of the automata, thus leading to a dramatical explosion of the number of locations, IMITATOR performs an *on-the-fly* composition of the automata. This *on-the-fly* composition allows to analyze bigger systems, and decrease drastically the computation time compared to IMITATOR.

3.3 Features

IMITATOR (version 2.2) includes the following features:

- Reachability analysis: given a PTA $\mathcal{A}$, compute the set of all the reachable states (as it is done in tools such as, e.g., HYTECH and PHAVer);

[6] Discrete variables are syntactic sugar allowing to factorize several locations into a single one. In IMITATOR, discrete variables are integer variables that can be updated using constants or other discrete variables.

- Inverse method algorithm: given a PTA $\mathcal{A}$ and a reference valuation π_0, synthesize a constraint guaranteeing the same trace set as for $\mathcal{A}[\pi_0]$;
- Behavioral cartography algorithm: given a PTA $\mathcal{A}$ and a rectangular parameter domain V_0, compute a list of tiles. Two different modes can be considered: (1) cover all the integer points of V_0 or, (2) call a given number of times the inverse method on an integer point selected randomly within V_0 (which is interesting for rectangles containing a very big number of integer points but few different tiles);
- Automatic generation of the trace sets, for the reachability analysis and for both algorithms *IM* and *BC*;
- Graphical output of the trace sets;
- Graphical output of the behavioral cartography.

As shown in Table 1, all those features (except the inverse method) are new features which were not available in the original version Imitator 1.0.

Tool	Inverse Method	Cartography	Computation of traces	Graphical output
Imitator 1.0	Yes	-	-	-
Imitator 2.2	Yes	Yes	Yes	Yes

Table 1: Comparison of the features of Imitator and Imitator

See Section 4.3.4 for the list of options available when calling Imitator.

4 How to Use Imitator

4.1 Installation

See the installation files available on the website for the most up-to-date information.

In short, Imitator is written in OCaml, and makes use of the following libraries:

- The OCaml ExtLib library (Extended Standard Library for Objective Caml)
- The Parma Polyhedra Library (PPL) [9]
- The GNU Multiple Precision Arithmetic Library (GMP)

Binaries and source code packages are available on Imitator's Web page.

4.2 The Imitator Input File

The syntax of the automata in Imitator is similar to the syntax of the automata for HyTech.

4.2.1 Variables

Discrete variables, clocks and parameters variable names must be disjoint.

The synchronization label names may be identical to other names (automata or variables). The automata names may be identical to other names (variables synchronization labels).

4.2.2 Parametric Timed Automata

See Appendix B.

4.2.3 Initial region and π_0

See Appendix B.

4.3 Calling Imitator

IMITATOR can be used with three different modes:

1. Reachability analysis: given a PTA $\mathcal{A}$, compute the whole set of reachable states from a given initial state.
2. Inverse Method: given a PTA $\mathcal{A}$ and a valuation π_0 of the parameters, compute a constraint on the parameters guaranteeing the same behavior as under π_0 [4].
3. Behavioral Cartography Algorithm: given a PTA $\mathcal{A}$ and a rectangle V_0 (bounded interval of values for each parameter), compute a cartography of the system [5].

We detail those three modes below.

4.3.1 Reachability Analysis

Given a PTA $\mathcal{A}$, the reachability analysis computes the whole set of reachable states from a given initial state. The syntax in this case is the following one:

```
IMITATOR <input_file> -mode reachability [options]
```

Note that there is no need to provide a π_0 or V_0 file in this case (if one is provided, it will be ignored).

In this case, IMITATOR outputs two files:

- A file `<input_file>.states` containing the set of all the reachable states, with their names and the associated constraint on the clocks and parameters. If one wants to generate also the constraint on the parameters only, use the `-with-parametric-log` option. This file also contains the source for the generation of the graphical file, using the DOT syntax. This log file is not generated in the case where the `-no-log` option is activated.
- A file `<input_file>.gif`, which is a graphical output in the gif format, generated using DOT, corresponding to the trace set. This graphical output is not generated in the case where the `-no-dot` option is activated.

Note that the location and the name of those two files can be changed using the `-log-prefix` option.

4.3.2 Inverse Method

Given a PTA $\mathcal{A}$ and a valuation π_0 of the parameters, the inverse method compute a constraint K_0 on the parameters guaranteeing that, for any $\pi \models K_0$, the trace sets of $\mathcal{A}[\pi]$ and $\mathcal{A}[\pi_0]$ are the same [4]. The syntax in this case is the following one:

`IMITATOR <input_file> <pi0_file> [-mode inversemethod] [options]`

Note that the `-mode inversemethod` option is not necessary, since the default value for `-mode` is precisely `inversemethod`.

Note that, unlike the algorithm given in [4], at a given iteration, the π_0-incompatible state is selected deterministically, for efficiency reasons. However, the π_0-incompatible inequality within a π_0-incompatible state is selected randomly, unless the `-no-random` option is activated.

In this case, IMITATOR outputs the resulting constraint K_0 on the standard output. Moreover, IMITATOR outputs the same two files as in the reachability analysis.

4.3.3 Cartography

Given a PTA $\mathcal{A}$ and a rectangle V_0 (bounded interval of values for each parameter), the Behavioral Cartography Algorithm computes a cartography of the system [5]. Two possible variants of the algorithm can be used:

1. The standard variant covers all the integer points within V_0. The syntax in this case is the following one:
   `IMITATOR <input_file> <V0_file> [-mode cover] [options]`

2. The alternative variant calls the inverse method a certain number of times on a random point V_0. The syntax in this case is the following one:
   `IMITATOR <input_file> <V0_file> [-mode randomX] [options]`
   where `X` represents the number of random points to consider. If a point has already been generated before, the inverse method is not called. If a point belongs to one of the tiles computed before, the inverse method is not called neither. Therefore, in practice, the number of tiles generated is smaller than `X`.

4.3.4 Options

The options available for IMITATOR are explained in the following.

`-acyclic` (default: `false`) Does not test if a new state was already encountered. In a normal use, when IMITATOR encounters a new state, it checks if it has been encountered before. This test may be time consuming for systems with a high number of reachable states. For acyclic systems, all traces pass only once by a given location. As a consquence, there are no cycles, so there should be no need to check if a given state has been encountered before. This is the main purpose of this option.

However, be aware that, even for acyclic systems, several (different) traces can pass by the same state. In such a case, if the `-acyclic` option is activated, IMITATOR will compute *twice* the states after the state common to the two

traces. As a consequence, it is all but sure that activating this option will lead to an increase of speed.

Note also that activating this option for non-acyclic systems may lead to an infinite loop in IMITATOR.

`-cart` (default: `off`) After execution of the behavioral cartography, plots the generated zones as a *PostScript* file. This option takes an integer which limits the number of generated plots, where each plot represents the projection of the parametric zones on two parameters. If the considered rectangle v_0 is spanned by two parameters only, then `-cart 1` will plot the projection of the generated zones on these two parameters.

This option makes use of the external utility `graph`, which is part of the *GNU plotting utils*, available on most Linux platforms. The generated files will be located in the same directory as the source files, unless option `-log-prefix` is used.

`-debug` (default: `standard`) Give some debugging information, that may also be useful to have more details on the way IMITATOR works. The admissible values for `-debug` are given below:

`nodebug` Give only the resulting constraint
`standard` Give little information (number of steps, computation time)
`low` Give little additional debugging information
`medium` Give quite a lot of debugging information
`high` Give much debugging information
`total` Give really too much information

`-fancy` (default: `false`) In the graphical output of the reachable states (see option `-no-dot`), provide detailed information on the local states of the composed automata.

`-inclusion` (default: `false`) Consider an inclusion of region instead of the equality when performing the *Post* operation. In other terms, when encountering a new state, IMITATOR checks if the same state (same location and same constraint) has been encountered before and, if yes, does not consider this "new" state. However, when the `-inclusion` option is activated, it suffices that a previous state with the same location and a constraint *greater or equal* to the constraint of the new state has been encountered to stop the analysis. This option corresponds to the way that, e.g., HYTECH works, and suffices when one wants to check the *non-reachability* of a given bad state.

`-log-prefix` (default: `<input_file>`) Set the prefix for log (`.states`) and graphical (`.gif`) files.

`-mode` (default: `inversemethod`) The mode for IMITATOR.

`reachability`	Parametric reachability analysis (see Section 4.3.1)
`inversemethod`	Inverse method (see Section 4.3.2)
`cover`	Behavioral Cartography Algorithm with full coverage (see Section 4.3.3)
`randomXX`	Behavioral Cartography Algorithm with `XX` iterations (see Section 4.3.3)

`-no-dot` (default: `false`) No graphical output using DOT.

`-no-log` (default: `false`) No generation of the files describing the reachable states.

`-no-random` (default: `false`) No random selection of the π_0-incompatible inequality (select the first found). By default, select an inequality in a random manner.

`-post-limit <limit>` (default: `none`) Limits the number of iterations in the *Post* exploration, i.e., the depth of the traces. In the cartography mode, this option gives a limit to *each* call to the inverse method.

`-sync-auto-detect` (default: `false`) IMITATOR considers that all the automata declaring a given synchronization label must be able to synchronize all together, so that the synchronization can happen. By default, IMITATOR considers that the synchronization labels declared in an automaton are those declared in the `synclabs` section. Therefore, if a synchronization label is declared but never used in (at least) one automaton, this label will never be synchronized in the execution[7].

The option `-sync-auto-detect` allows to detect automatically the synchronization labels in each automaton: the labels declared in the `synclabs` section are ignored, and the IMITATOR considers that only the labels really used in an automaton are those considered to be declared.

`-time-limit <limit>` (default: `none`) Try to limit the execution time (the value `<limit>` is given in seconds). Note that, in the current version of IMITATOR, the test of time limit is performed at the end of an iteration only (i.e., at the end of a given *Post* iteration). In the cartography mode, this option represents a *global* time limit, not a limit for each call to the inverse method.

`-timed` (default: `false`) Add a timing information to each output of the program.

`-with-parametric-log` (default: `false`) For any constraint on the clocks and the parameters in the description of the states in the log files, add the constraint on the parameters only (i.e., eliminate clock variables).

[7] In such a case, the synchronization label is actually completely removed before the execution, in order to optimize the execution, and the user is warned of this removal.

4.3.5 Examples of Calls

`IMITATOR flipflop.imi -mode reachability` Computes a reachability analysis on the automata described in file `flipflop.imi`. Will generate files `flipflop.imi.states`, containing the description of the reachable states, and `flipflop.imi.gif` depicting the reachability graph.

`IMITATOR flipflop.imi flipflop.pi0` Calls the inverse method on the automata described in file `flipflop.imi`, and the reference valuation π_0 given in file `flipflop.pi0`. The resulting constraint K_0 will be given on the standard output. Moreover, IMITATOR will generate the file `flipflop.imi.states`, containing the description of the (parametric) states reachable under K_0, and the file `flipflop.imi.gif` depicting the reachability graph under any point $\pi \models K_0$.

`IMITATOR flipflop.imi flipflop.pi0 -no-dot -no-log` Calls the inverse method on the automata described in file `flipflop.imi`, and the reference valuation given in file `flipflop.pi0`. The resulting constraint will be given on the standard output. No file will be generated.

`IMITATOR flipflop.imi flipflop.pi0 -with-parametric-log` Calls the inverse method on the automata described in file `flipflop.imi`, and the reference valuation π_0 given in file `flipflop.pi0`. The resulting constraint K_0 will be given on the standard output. and IMITATOR will generate the file `flipflop.imi.states`, containing the description of the (parametric) states reachable under K_0. Moveover, for any state in this file, both the constraint on the clocks and the parameters, and the constraint on the parameters will be given. IMITATOR will also generate the file `flipflop.imi.gif` depicting the reachability graph under any point $\pi \models K_0$.

`IMITATOR SRlatch.imi SRlatch.v0 -mode cover` Calls the behavioral cartography algorithm on the automata described in file `flipflop.imi`, and the rectangle V_0 given in file `SRlatch.v0`. The algorithm will cover (at least) all the integer points within V_0. The resulting set of tiles will be given on the standard output. Given n the number of generated tiles (i.e., the number of calls to the inverse method algorithm), the program will generate n files of the form `SRlatch.imi_`i`.states` (resp. n files of the form `SRlatch.imi_`i`.gif`) giving the description of the states (resp. the reachability graph) of tile i, for $i = 1, \ldots, n$.

`IMITATOR SRlatch.imi SRlatch.v0 -mode random100 -no-log` Calls the behavioral cartography algorithm on the automata described in file `flipflop.imi`, and the rectangle V_0 given in file `SRlatch.v0`. The program will call the inverse method on 100 points randomly selected within V_0. Since some points may be generated several times, or some points may belong to previously generated tiles (see Section 4.3.3), the number n of tiles generated will be such that $n \leq 100$. The program will generate n files of the form `SRlatch.imi_`i`.gif` giving the reachability graph of tile i, for $i = 1, \ldots, n$.

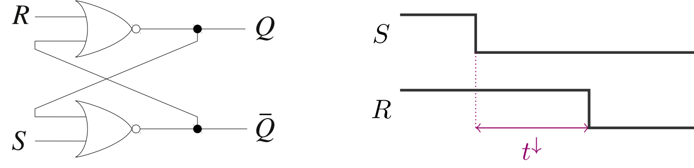

Figure 5: SR latch (left) and environment (right)

5 Example: SR-latch

We consider a SR-latch described in, e.g., [13], and depicted on Fig. 5 left. The possible configurations of the latch are the following ones:

S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

We consider an initial configuration with $R = S = 1$ and $Q = \overline{Q} = 0$. As depicted in Fig. 5, the signal S first goes down. Then, the signal R goes down after a time $t^{\downarrow}$.

We consider that the gate Nor_1 (resp. Nor_2) has a punctual parametric delay δ_1 (resp. δ_2). Moreover, the parameter $t^{\downarrow}$ corresponds to the time duration between the fall of S and the fall of R.

5.1 Parametric Reachability Analysis

We first perform a reachability analysis. The launch command for IMITATOR is the following one:

```
IMITATOR SRlatch.imi -mode reachability
```

Considering this environment, the trace set of this system is given in Fig. 6, where the states q_i, $i = 0, \ldots, 6$ correspond to the following values for each signal:

State	S	R	Q	$\overline{Q}$
q_0	1	1	0	0
q_1	0	1	0	0
q_2	0	0	0	0
q_3	0	1	0	1
q_4	0	0	0	1
q_5	0	0	1	0
q_6	0	0	0	1

5.2 Behavioral Cartography Algorithm

Using IMITATOR, we now perform a behavioral cartography of this system. We consider the following rectangle V_0 for the parameters:

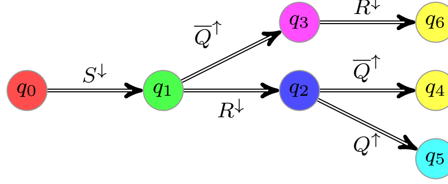

Figure 6: Parametric reachability analysis of the SR latch

$$t^\downarrow \in [0, 10]$$
$$\delta_1 \in [0, 10]$$
$$\delta_2 \in [0, 10]$$

The launch command for IMITATOR is the following one:

```
IMITATOR SRlatch.imi SRlatch.v0 -mode cover
```

We get the following six behavioral tiles. Note that the graphical outputs, automatically generated by IMITATOR in the `gif` format, were rewritten in LaTeX in this document.

Tile 1. This tile corresponds to the values of the parameters verifying the following constraint:

$$t^\downarrow = \delta_2 \quad \wedge \quad \delta_1 = 0$$

The trace set of this tile is given in Fig. 7.

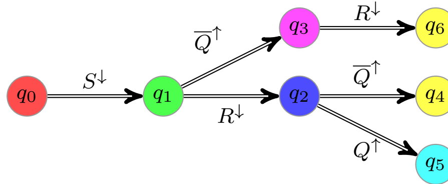

Figure 7: Trace set of tile 1 for the SR latch

Since $t^\downarrow = \delta_2$, $R^\downarrow$ and $\overline{Q}^\uparrow$ will occur at the same time. Thus, the order of those two events is unspecified, which explains the choice between going to q_2 or q_3. When in state q_2, either $Q^\uparrow$ can occur (since $\delta_1 = 0$), in which case the system is stable, or $\overline{Q}^\uparrow$ can occur, which also leads to stability.

Tile 2. This tile corresponds to the values of the parameters verifying the following constraint:

$$t^\downarrow = \delta_2 \quad \wedge \quad \delta_1 > 0$$

The trace set of this tile is given in Fig. 8.

Since $t^\downarrow = \delta_2$, $R^\downarrow$ and $\overline{Q}^\uparrow$ will occur at the same time. Thus, the order of those two events is unspecified, which explains the choice between going to q_2 or q_3. When in state q_2, $Q^\uparrow$ can not occur (since $\delta_1 > 0$), so $\overline{Q}^\uparrow$ occurs immediately after $R^\downarrow$, which leads to stability.

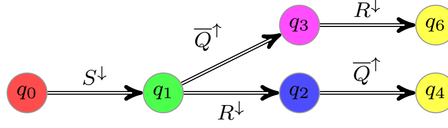

Figure 8: Trace set of tile 2 for the SR latch

Tile 3. This tile corresponds to the values of the parameters verifying the following constraint:

$$\delta_2 > t^\downarrow + \delta_1$$

The trace set of this tile is given in Fig. 9.

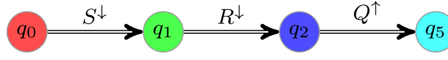

Figure 9: Trace set of tile 3 for the SR latch

In this case, since $\delta_2 > t^\downarrow + \delta_1$, $S^\downarrow$ will occur before the gate Nor_2 has the time to change. For the same reason, $Q^\uparrow$ will change before Nor_1 has the time to change. With $Q = 1$, the system is now stable: Nor_1 does not change.

Tile 4. This tile corresponds to the values of the parameters verifying the following constraint:

$$t^\downarrow + \delta_1 = \delta_2 \quad \wedge \quad \delta_2 \geq \delta_1 \quad \wedge \quad \delta_1 > 0$$

The trace set of this tile is given in Fig. 10.

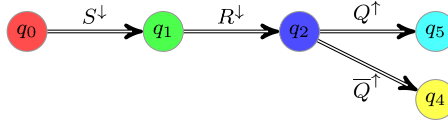

Figure 10: Trace set of tile 4 for the SR latch

Since $t^\downarrow + \delta_1 = \delta_2$, both $Q^\uparrow$ or $\overline{Q}^\uparrow$ can occur. Once one of them occured, the system gets stable, and no other change occurs.

Tile 5. This tile corresponds to the values of the parameters verifying the following constraint:

$$\delta_2 > t^\downarrow \quad \wedge \quad t^\downarrow + \delta_1 > \delta_2$$

The trace set of this tile is given in Fig. 11.

Since $\delta_2 > t^\downarrow$, the gate Nor_2 can not change before $R^\downarrow$ occurs. However, since $t^\downarrow + \delta_1 > \delta_2$, the gate Nor_2 changes before $Q^\uparrow$ can occur, thus leading to event $\overline{Q}^\uparrow$.

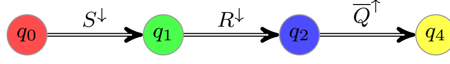

Figure 11: Trace set of tile 5 for the SR latch

Tile 6. This tile corresponds to the values of the parameters verifying the following constraint:

$$t^{\downarrow} > \delta_2$$

The trace set of this tile is given in Fig. 12.

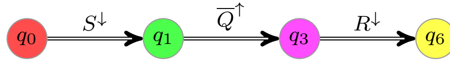

Figure 12: Trace set of tile 6 for the SR latch

Since $t^{\downarrow} > \delta_2$, $\overline{Q}^{\uparrow}$ occurs before $S^{\downarrow}$. The system is then stable.

Cartography. We give in Fig. 13 the cartography of the SR latch example. For the sake of simplicity of representation, we consider only parameters δ_1 and δ_2. Therefore, we set $t^{\downarrow} = 1$.

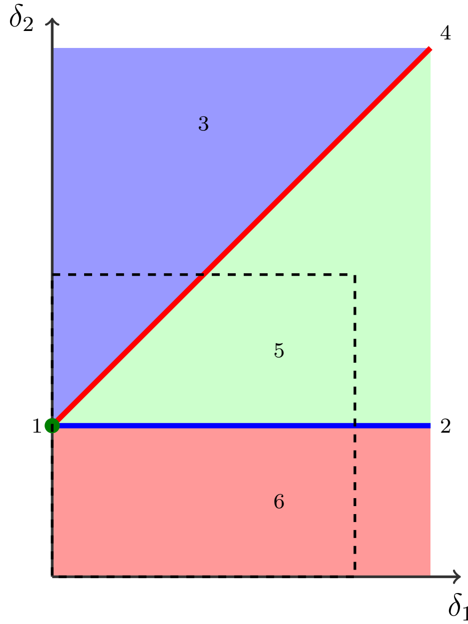

Figure 13: Behavioral cartography of the SR latch according to δ_1 and δ_2

Note that tile 1 corresponds to a point, and tiles 2 and 4 correspond to lines.

The rectangle V_0 has been represented with dashed lines. Note that all tiles (except tile 1) are unbounded, so that they cover, not only V_0, but all the positive real-valued plan.

The source code of this example is available in Appendix A.

6 Final Remarks

The tool IMITATOR allows us to solve the good parameters problem for timed automata by iterating the inverse method algorithm on the integer points of a given rectangular parameter domain V_0. In practice, our cartography algorithm covers not only (most of) V_0 but also a significant part of the whole parametric space beyond V_0. This is of interest to classify the behavior of the system into good or bad for dense and unbounded values of the parameters.

Our tool has been successfully applied to various examples of asynchronous circuits and protocols.

Future works include:

- an automatic verification of the property one wants to check, e.g., by using a tool such as UPPAAL [20];
- a "dynamic" cartography, where the scale (so far, the integers) can be refined to fill the possible holes;
- a backward analysis, i.e., considering a *Pre* operation instead of *Post* computation in Algorithm 1;
- the reachability analysis of a given state, and the generation of a trace from the initial state to this given state;
- the extension to hybrid systems;
- the automatic generation of the probabilities of a given property in the probabilistic framework, without the use of an external tool (e.g., PRISM [16])
- the automatic generation of the "next point" not covered by *Tiling* without testing all the integer points (note that a random generation of points is already implemented);
- the possibility to compute several tiles in parallel in the cartography algorithm;
- a user-friendly graphical interface;
- the possibility to save and load sets of states.

Acknowledgments. Emmanuelle Encrenaz and Laurent Fribourg have been great contributors of IMITATOR, on a theoretical point of view, and to find applications both from the literature and real case studies. Abdelrezzak Bara provided several examples from the hardware literature. Jeremy Sproston provided examples from the probabilistic community. Bertrand Jeannet has been of great help on the installation part, and the linking with Apron [18]. Ulrich Kühne started to improve IMITATOR, and added the link with PPL.

References

[1] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

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

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

[4] É. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009.

[5] É. André and L. Fribourg. Behavioral cartography of timed automata. In *RP'10*, LNCS. Springer, 2010. To appear.

[6] É. André, L. Fribourg, and J. Sproston. An extension of the inverse method to probabilistic timed automata. In *AVoCS'09*, volume 23 of *Electronic Communications of the EASST*, 2009.

[7] Étienne André. IMITATOR web page. `http://www.lsv.ens-cachan.fr/~andre/IMITATOR2`.

[8] Étienne André. IMITATOR: A tool for synthesizing constraints on timing bounds of timed automata. In *ICTAC'09*, volume 5684 of *LNCS*, pages 336–342. Springer, 2009.

[9] 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.

[10] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV '00*, pages 154–169. Springer-Verlag, 2000.

[12] G. Frehse, S.K. Jha, and B.H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC '08*, volume 4981 of *LNCS*, pages 187–200. Springer, 2008.

[13] D. Harris and S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[14] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[15] T.A. Henzinger and H. Wong-Toi. Using HYTECH to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS 1165*. Springer-Verlag, 1996.

[16] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS'06*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[17] T.S. Hune, J.M.T. Romijn, M.I.A. Stoelinga, and F.W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 2002.

[18] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV '09*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[19] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.

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

A Source Code of the Example

A.1 Main Input File

```
1  --**************************************************************--
2  --**************************************************************--
3  --   Laboratoire Specification et Verification
4  --
5  --   Race on a digital circuit (SR Latch)
6  --
7  --   Etienne ANDRE
8  --
9  --   Created:        2010/03/19
10 --   Last modified : 2010/03/24
11 --**************************************************************--
12 --**************************************************************--
13
14 var     ckNor1, ckNor2, s
15                 : clock;
16
17         dNor1_l, dNor1_u,
18         dNor2_l, dNor2_u,
19         t_down
20                         : parameter;
21
22
23 --**************************************************************--
24   automaton norGate1
25 --**************************************************************--
26 synclabs: R_Up, R_Down, overQ_Up, overQ_Down,
27         Q_Up, Q_Down;
28 initially Nor1_110;
29
30 -- UNSTABLE
31 loc Nor1_000: while ckNor1 <= dNor1_u wait {}
32         when True sync R_Up do {} goto Nor1_100;
33         when True sync overQ_Up do {} goto Nor1_010;
34         when ckNor1 >= dNor1_l sync Q_Up do {} goto
               Nor1_001;
35
36 -- STABLE
37 loc Nor1_001: while True wait {}
38         when True sync R_Up do {ckNor1' = 0} goto
               Nor1_101;
39         when True sync overQ_Up do {ckNor1' = 0} goto
               Nor1_011;
40
41 -- STABLE
42 loc Nor1_010: while True wait {}
43         when True sync R_Up do {} goto Nor1_110;
```

```
44          when True sync overQ_Down do {ckNor1' = 0} goto
                Nor1_000;
45
46  -- UNSTABLE
47  loc Nor1_011: while ckNor1 <= dNor1_u wait {}
48          when True sync R_Up do {ckNor1' = 0} goto
                Nor1_111;
49          when True sync overQ_Down do {} goto Nor1_001;
50          when ckNor1 >= dNor1_l sync Q_Down do {} goto
                Nor1_010;
51
52  -- STABLE
53  loc Nor1_100: while True wait {}
54          when True sync R_Down do {ckNor1' = 0} goto
                Nor1_000;
55          when True sync overQ_Up do {} goto Nor1_110;
56
57  -- UNSTABLE
58  loc Nor1_101: while ckNor1 <= dNor1_u wait {}
59          when True sync R_Down do {} goto Nor1_001;
60          when True sync overQ_Up do {ckNor1' = 0} goto
                Nor1_111;
61          when ckNor1 >= dNor1_l sync Q_Down do {} goto
                Nor1_100;
62
63  -- STABLE
64  loc Nor1_110: while True wait {}
65          when True sync R_Down do {} goto Nor1_010;
66          when True sync overQ_Down do {} goto Nor1_100;
67
68  -- UNSTABLE
69  loc Nor1_111: while ckNor1 <= dNor1_u wait {}
70          when True sync R_Down do {ckNor1' = 0} goto
                Nor1_011;
71          when True sync overQ_Down do {ckNor1' = 0} goto
                Nor1_101;
72          when ckNor1 >= dNor1_l sync Q_Down do {} goto
                Nor1_110;
73
74  end -- norGate1
75
76
77  --**********************************************************--
78    automaton norGate2
79  --**********************************************************--
80  synclabs: Q_Up, Q_Down, S_Up, S_Down,
81          overQ_Up, overQ_Down;
82  -- initially Nor2_110;
83  initially Nor2_001;
84
```

```
85  -- UNSTABLE
86  loc Nor2_000: while ckNor2 <= dNor2_u wait {}
87          when True sync Q_Up do {} goto Nor2_100;
88          when True sync S_Up do {} goto Nor2_010;
89          when ckNor2 >= dNor2_l sync overQ_Up do {} goto
                Nor2_001;
90
91  -- STABLE
92  loc Nor2_001: while True wait {}
93          when True sync Q_Up do {ckNor2' = 0} goto
                Nor2_101;
94          when True sync S_Up do {ckNor2' = 0} goto
                Nor2_011;
95
96  -- STABLE
97  loc Nor2_010: while True wait {}
98          when True sync Q_Up do {} goto Nor2_110;
99          when True sync S_Down do {ckNor2' = 0} goto
                Nor2_000;
100
101 -- UNSTABLE
102 loc Nor2_011: while ckNor2 <= dNor2_u wait {}
103         when True sync Q_Up do {ckNor2' = 0} goto
                Nor2_111;
104         when True sync S_Down do {} goto Nor2_001;
105         when ckNor2 >= dNor2_l sync overQ_Down do {} goto
                 Nor2_010;
106
107 -- STABLE
108 loc Nor2_100: while True wait {}
109         when True sync Q_Down do {ckNor2' = 0} goto
                Nor2_000;
110         when True sync S_Up do {} goto Nor2_110;
111
112 -- UNSTABLE
113 loc Nor2_101: while ckNor2 <= dNor2_u wait {}
114         when True sync Q_Down do {} goto Nor2_001;
115         when True sync S_Up do {ckNor2' = 0} goto
                Nor2_111;
116         when ckNor2 >= dNor2_l sync overQ_Down do {} goto
                 Nor2_100;
117
118 -- STABLE
119 loc Nor2_110: while True wait {}
120         when True sync Q_Down do {} goto Nor2_010;
121         when True sync S_Down do {} goto Nor2_100;
122
123 -- UNSTABLE
124 loc Nor2_111: while ckNor2 <= dNor2_u wait {}
```

```
125         when True sync Q_Down do {ckNor2' = 0} goto
                Nor2_011;
126         when True sync S_Down do {ckNor2' = 0} goto
                Nor2_101;
127         when ckNor2 >= dNor2_l sync overQ_Down do {} goto
                 Nor2_110;
128
129 end -- norGate2
130
131
132 --****************************************************--
133   automaton env
134 --****************************************************--
135 synclabs: R_Down, R_Up, S_Down, S_Up;
136 initially env_11;
137
138 -- ENVIRONMENT : first S then R at constant time
139 loc env_11: while True wait {}
140         when True sync S_Down do {s' = 0} goto env_10;
141
142 loc env_10: while s <= t_down wait {}
143         when s = t_down sync R_Down do {} goto env_final;
144
145 loc env_final: while True wait {}
146
147 end -- env
148
149 --****************************************************--
150 -- ANALYSIS
151 --****************************************************--
152
153 var init : region;
154
155 init := True
156         ----------------------------------------------------
157         -- INITIAL LOCATIONS
158         ----------------------------------------------------
159         -- S and R down
160         & loc[norGate1] = Nor1_100
161         & loc[norGate2] = Nor2_010
162         & loc[env]              = env_11
163
164         ----------------------------------------------------
165         -- INITIAL CONSTRAINTS
166         ----------------------------------------------------
167         & ckNor1         = 0
168         & ckNor2         = 0
169         & s                     = 0
170
171         & dNor1_l >= 0
```

```
172         & dNor2_l >= 0
173
174         & dNor1_l <= dNor1_u
175         & dNor2_l <= dNor2_u
176 ;
```

A.2 V_0 File

```
1           ------------------------------------------------------------
2           -- V0
3           ------------------------------------------------------------
4           & dNor1_l       = 3
5           & dNor1_u       = 3 .. 20
6           & dNor2_l       = 5
7           & dNor2_u       = 5 .. 20
8           & t_down        = 10
```

B Complete Grammar

B.1 Grammar of the Input File

IMITATOR input is described by the following grammar. Non-terminals appear within angled parentheses. A non-terminal followed by two colons is defined by the list of immediately following non-blank lines, each of which represents a legal expansion. Input characters of terminals appear in typewritter font. The meta symbol ϵ denotes the empty string.

The text in green is not taken into account by IMITATOR, but is allowed (or sometimes necessary) in order to allow the compatibility with HYTECH files.

⟨*imitator_input*⟩ ::
  ⟨*automata_descriptions*⟩ ⟨*init*⟩

We define each of those two components below.

B.1.1 Automata Descriptions

⟨*automata_descriptions*⟩ ::
  ⟨*declarations*⟩ ⟨*automata*⟩

⟨*declarations*⟩ ::
  `var` ⟨*var_lists*⟩

⟨*var_lists*⟩ ::
  ⟨*var_list*⟩ `:` ⟨*var_type*⟩ `;` ⟨*var_lists*⟩
| ϵ

⟨*var_list*⟩ ::
  `<name>`
| `<name>` `,` ⟨*var_list*⟩

⟨*var_type*⟩ ::
  `clock`
| `discrete`
| `parameter`

⟨*automata*⟩ ::
  ⟨*automaton*⟩ ⟨*automata*⟩
| ϵ

⟨*automaton*⟩ ::
  `automaton` `<name>` ⟨*prolog*⟩ ⟨*locations*⟩ `end`

⟨*prolog*⟩ ::
  ⟨*initialization*⟩ ⟨*sync_labels*⟩
| ⟨*sync_labels*⟩ ⟨*initialization*⟩
| ⟨*sync_labels*⟩

⟨*initialization*⟩ ::
    initially <name> ⟨*state_initialization*⟩ ;

⟨*state_initialization*⟩ ::
    & ⟨*convex_predicate*⟩
| ϵ

⟨*prolog*⟩ ::
    synclabs : ⟨*sync_var_list*⟩ ;

⟨*sync_var_list*⟩ ::
    ⟨*sync_var_nonempty_list*⟩
| ϵ

⟨*sync_var_nonempty_list*⟩ ::
    <name> , ⟨*sync_var_nonempty_list*⟩
| <name>

⟨*locations*⟩ ::
    ⟨*location*⟩ ⟨*locations*⟩
| ϵ

⟨*locations*⟩ ::
    loc <name> : while ⟨*convex_predicate*⟩ wait () ⟨*transitions*⟩
| loc <name> : while ⟨*convex_predicate*⟩ wait ⟨*transitions*⟩

⟨*transitions*⟩ ::
    ⟨*transition*⟩ ⟨*transitions*⟩
| ϵ

⟨*transition*⟩ ::
    when ⟨*convex_predicate*⟩ ⟨*update_synchronization*⟩ goto <name> ;

⟨*update_synchronization*⟩ ::
    ⟨*updates*⟩
| ⟨*syn_label*⟩
| ⟨*updates*⟩ ⟨*syn_label*⟩
| ⟨*syn_label*⟩ ⟨*updates*⟩
| ϵ

⟨*updates*⟩ ::
    do (⟨*update_list*⟩)

⟨*update_list*⟩ ::
    ⟨*update_nonempty_list*⟩
| ϵ

⟨*update_nonempty_list*⟩ ::
    ⟨*update*⟩ , ⟨*update_nonempty_list*⟩
| ⟨*update*⟩

⟨*update_nonempty_list*⟩ ::
    <name> ' = ⟨*linear_expression*⟩

⟨*syn_label*⟩ ::
    sync <name>

⟨*convex_predicate*⟩ ::
    ⟨*linear_constraint*⟩ & ⟨*convex_predicate*⟩
  | ⟨*linear_constraint*⟩

⟨*linear_constraint*⟩ ::
    ⟨*linear_expression*⟩ ⟨*relop*⟩ ⟨*linear_expression*⟩
  | True
  | False

⟨*relop*⟩ ::
    <
  | <=
  | =
  | >=
  | >

⟨*linear_expression*⟩ ::
    ⟨*linear_term*⟩
  | ⟨*linear_expression*⟩ + ⟨*linear_term*⟩
  | ⟨*linear_expression*⟩ - ⟨*linear_term*⟩

⟨*linear_term*⟩ ::
    ⟨*rational*⟩
  | ⟨*rational*⟩ <name>
  | ⟨*rational*⟩ * <name>
  | <name>
  | (⟨*linear_term*⟩)

⟨*rational*⟩ ::
    ⟨*integer*⟩
  | ⟨*integer*⟩ / ⟨*pos_integer*⟩

⟨*integer*⟩ ::
    ⟨*pos_integer*⟩
  | - ⟨*pos_integer*⟩

⟨*pos_integer*⟩ ::
    <int>

B.1.2 Initial State

⟨*init*⟩ ::
    ⟨*init_declaration*⟩ ⟨*init_definition*⟩ ⟨*reach_command*⟩

⟨*init_declaration*⟩ ::
    `var init : region ;`
  | ϵ

⟨*reach_command*⟩ ::
    `print ( reach forward from init endreach ) ;`
  | ϵ

⟨*init_definition*⟩ ::
    `init :=` ⟨*region_expression*⟩ `;`

⟨*region_expression*⟩ ::
    ⟨*state_predicate*⟩
  | `(` ⟨*region_expression*⟩ `)`
  | ⟨*region_expression*⟩ `&` ⟨*region_expression*⟩

⟨*state_predicate*⟩ ::
    `loc [ <name> ] = <name>`
  | ⟨*linear_constraint*⟩

B.2 Reserved Words

The following words are keywords and cannot be used as names for automata, variables, synchronization labels or locations.

`and`, `automaton`, `clock`, `discrete`, `do`, `end`, `endreach`, `False`, `forward`, `from`, `goto`, `if`, `in`, `init`, `initially`, `loc`, `locations`, `not`, `or`, `parameter`, `print`, `reach`, `region`, `sync`, `synclabs`, `True`, `var`, `wait`, `when`, `while`