# IMITATOR II: A Tool for Solving
# the Good Parameters Problem in Timed Automata

Étienne André[*]

LSV, ENS de Cachan & CNRS, Cachan, France

andre@lsv.ens-cachan.fr

We present here IMITATOR II, a new version of IMITATOR, a tool implementing the "inverse method" for parametric timed automata: given a reference valuation of the parameters, it synthesizes a constraint such that, for any valuation satisfying this constraint, the system behaves the same as under the reference valuation in terms of traces, i.e., alternating sequences of locations and actions. IMITATOR II also implements the "behavioral cartography algorithm", allowing us to solve the following good parameters problem: find a set of valuations within a given bounded parametric domain for which the system behaves well. We present new features and optimizations of the tool, and give results of applications to various examples of asynchronous circuits and communication protocols.

## 1  Introduction

Timed automata [2] are finite control automata equipped with *clocks*, which are real-valued variables which increase uniformly, and that are compared with *timing delays*. One can check the correctness of a system modeled by a timed automaton for one particular value for each delay (using model checkers such as, e.g., UPPAAL [23]), but this does not give any information for other values. Actually, checking the correctness of the system for all the delays, even in a bounded interval, would require an infinite number of calls to the model checker, because those delays can have real (or rational) values. It is therefore interesting to reason *parametrically*, by considering that these delays are unknown constants, or *parameters*, and try to synthesize a *constraint* (i.e., a conjunction of linear inequalities) on these parameters 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.**  We aim at solving the *good parameters problem*, as defined in [15] in the framework of linear hybrid automata [1]: "Given a PTA $\mathscr{A}$ and a rectangular parameter domain $V_0$, what is the largest set of parameter values within $V_0$ for which $\mathscr{A}$ is safe?"

The parameter design problem for timed automata (and more generally, for linear hybrid automata) was formulated in [19], 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 [15], the authors propose an extension based on the *counterexample guided abstraction refinement* (CEGAR, [14]). 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.

---

**Contributions.**    The tool IMITATOR II presented in this paper is based on the *inverse method* [5], which starts from a "good instantiation" $\pi_0$ of the parameters that one wants to generalize. More precisely, IMITATOR II synthesizes a constraint $K_0$ on the parameters that corresponds to a dense set of valuations such that, for all instantiation $\pi$ of parameters in this set, the behavior of the timed automaton $\mathscr{A}$ is *(time-abstract) equivalent* to the behavior of $\mathscr{A}$ under $\pi_0$, 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 II implements the *behavioral cartography algorithm* [6], which synthesizes a constraint on the parameters ("tile") by calling the inverse method on integers point located within a given bounded parameter real-valued domain (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 only the *integer* subspace of $V_0$, but two major extensions: first, not only the integer points but a major part of the dense set of *real-valued* points of $V_0$ 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 II is a new version of IMITATOR [7], a prototype written in Python [26] implementing the inverse method, and calling the model checker HYTECH [18]. IMITATOR II has been entirely rewritten and is now a standalone tool, making use of the APRON library [20] and the Parma Polyhedra Library [9]. Compared to IMITATOR, the computation timings of IMITATOR II have dramatically decreased. Moreover, IMITATOR II offers new features, such as the implementation of the cartography algorithm, the visualization of the trace sets of the constraints, and of the cartography (for 2 parameter dimensions).

**Related Tools.**    IMITATOR II has been designed to implement the inverse method and the cartography algorithm and, as far as we know, it is the only tool implementing this kind of algorithms. Although it is thus not possible to compare directly the computation times of IMITATOR II with other tools, it is interesting to mention the following tools allowing to perform related analyses of timed systems.

HYTECH [18] is the first model checker for analyzing parametric hybrid automata. It features an intuitive input syntax, and performs reachability analysis and operations on states sets. Although HYTECH has been used to verify interesting case studies, it can hardly verify even medium sized examples because of its arithmetics with limited precision leading to overflows, and its static composition of the automata, preventing the composition of more than a dozen of automata.

The tool PHAVer [17], designed by Goran Frehse, highly improves the scalability compared to HYTECH, and performs analyses on parametric hybrid systems using exact arithmetics with unlimited precision and convex polyhedra, using the *Parma Polyhedra Library* (PPL) [9]. Moreover, PHAVer offers various features such as automatic partitioning, graphical outputs, and forward/backward abstraction refinement. Various case studies have been verified, in particular in the framework of analog circuits [16].

UPPAAL is a powerful tool for model checking timed automata extended with several data types [23]. In particular, it verifies very efficiently timing properties such as reachability, safety or liveness properties on timed automata. However, although an extension allowing to perform parametric model checking is mentioned in [4], the standard version of UPPAAL does not allow the use of hybrid or parametric systems.

TREX [8] is a model checker allowing to verify properties on parametric timed automata extended with integer counters and finite-domain variables. TREX features on-the-fly verification of safety properties, as well as parameter synthesis either using parametric reachability, or in order to satisfy properties. Various representations are allowed and both forward and backward exploration algorithms can be used.

Finally, the RED library [25] features analysis of real-time systems using Clock-Restriction Diagrams, as well as parametric analysis of hybrid systems using Hybrid-Restriction Diagrams.

**Plan of the Paper.** We first recall the framework of Parametric Timed Automata, the inverse method algorithm and the behavioral cartography algorithm in Section 2. We then introduce IMITATOR II in Section 3 and give details on its internal structure and its various features. We present in Section 4 a range of case studies including hardware devices and unbounded communication protocols. We give final remarks in Section 5.

## 2   Behavioral Cartography of Timed Automata

**Parametric Timed Automata.** We use in this paper the same formalism as in [6]. Throughout this paper, we assume a fixed set $X = \{x_1, \ldots, x_H\}$ of *clocks*, and a fixed set $P = \{p_1, \ldots, p_M\}$ of *parameters*. 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.

Parametric timed automata are an extension of timed automata [2] to the parametric case, allowing within guards and invariants the use of parameters in place of constants [3]. A *parametric timed automaton (PTA)* $\mathscr{A}$ is a 6-tuple of the form $\mathscr{A} = (\Sigma, Q, q_0, K, I, \rightarrow)$, where $\Sigma$ is a finite 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 $(q, g, a, \rho, q')$ 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 $\mathscr{A} = (\Sigma, Q, q_0, K, I, \rightarrow)$. We simply denote this PTA by $\mathscr{A}(K)$, in order to emphasize the fact that only $K$ will change in $\mathscr{A}$. For every parameter valuation $\pi = (\pi_1, \ldots, \pi_M)$, $\mathscr{A}[\pi]$ denotes the PTA $\mathscr{A}(K)$, where $K$ is $\bigwedge_{i=1}^{M} p_i = \pi_i$. This corresponds to the PTA obtained from $\mathscr{A}$ by substituting every occurrence of a parameter $p_i$ by constant $\pi_i$ in the guards and invariants. We say that $p_i$ is *instantiated* with $\pi_i$. Note that $\mathscr{A}[\pi]$ is a standard timed automaton.

A *(symbolic) state s* of $\mathscr{A}(K)$ is a couple $(q, C)$ where $q$ is a location, and $C$ a constraint on the clocks and the parameters. The *initial state* of $\mathscr{A}(K)$ is a state $s_0$ of the form $(q_0, C_0)$, where $C_0 = K \wedge I(q_0) \wedge \bigwedge_{i=1}^{H-1} x_i = x_{i+1}$. In the latter expression, $K$ is the initial constraint on the parameters, $I(q_0)$ is the invariant of the initial state, and the rest of the expression lets clocks evolve from the same initial value. A *run R* of $\mathscr{A}(K)$ is an alternating sequence of states and actions of the form $(q_0, C_0) \overset{a_0}{\Rightarrow} (q_1, C_1) \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (q_m, C_m)$, such that for all $i = 0, \ldots, m-1$, $a_i \in \Sigma$ and $(q_i, C_i) \overset{a_i}{\Rightarrow} (q_{i+1}, C_{i+1})$ is a step of $\mathscr{A}(K)$. 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* $\mathscr{A}(K)$ refers to the set of traces associated to the runs of $\mathscr{A}(K)$.

In the following, we are interested in verifying properties on traces sets. For example, a trace can be said to be "good" if it never contains any "bad" location of a given set, or if a given action always occurs before another one (see [6]). Given such 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. 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.

**The Inverse Method.** We recall here the inverse method algorithm $IM(\mathscr{A}, \pi_0)$, as defined in [5], which synthesizes a constraint $K_0$ on the parameters such that $\pi_0 \models K_0$, and for all $\pi \in K_0$, the trace sets of $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$ are equal. Starting with $K = \texttt{true}$, we iteratively compute a growing set of reachable states. When a $\pi_0$-*incompatible* state $(q, C)$ is encountered (i.e., when $\pi_0 \not\models (\exists X : C)$), $K$ is refined as follows: a $\pi_0$-incompatible inequality $J$ (i.e., such that $\pi_0 \not\models J$) 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 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 $\mathscr{A}[\pi_1]$ and $\mathscr{A}[\pi_2]$ 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 $\mathscr{A}(K)$ will be referred to as "the trace set of $K$".

---

**Algorithm 1:** $IM(\mathscr{A}, \pi_0)$

---

    **input** : A PTA $\mathscr{A}$ of initial state $s_0$
    **input** : Valuation $\pi_0$ of the parameters
    **output**: Constraint $K$ on the parameters

1   $i \leftarrow 0; \ K \leftarrow \texttt{true}; \ S \leftarrow \{s_0\}$
2   **while** $\texttt{true}$ **do**
3      **while** *there are $\pi_0$-incompatible states in S* **do**
4          Select a $\pi_0$-incompatible state $(q,C)$ of $S$ (i.e., s.t. $\pi_0 \not\models (\exists X : C)$) ;
5          Select a $\pi_0$-incompatible inequality $J$ in $(\exists X : C)$ (i.e., s.t. $\pi_0 \not\models J$) ;
6          $K \leftarrow K \wedge \neg J$ ;
7          $S \leftarrow \bigcup_{j=0}^{i} Post_{\mathscr{A}(K)}^{j}(\{s_0\})$ ;
8      **if** $Post_{\mathscr{A}(K)}(S) \sqsubseteq S$ **then return** $K \leftarrow \bigcap_{(q,C) \in S}(\exists X : C)$
9      $i \leftarrow i+1; \ S \leftarrow S \cup Post_{\mathscr{A}(K)}(S)$ ;          // $S = \bigcup_{j=0}^{i} Post_{\mathscr{A}(K)}^{j}(\{s_0\})$

---

The algorithm *IM* is given in Algorithm 1. We define $Post_{\mathscr{A}(K)}^{i}(S)$ as the set of states reachable from $S$ in exactly $i$ steps, and $Post_{\mathscr{A}(K)}^{*}(S)$ as the set of all states reachable from $S$ in $\mathscr{A}(K)$ (i.e., $Post_{\mathscr{A}(K)}^{*}(S) = \bigcup_{i \geq 0} Post_{\mathscr{A}(K)}^{i}(S)$). Given two sets of states $S$ and $S'$, we write $S \sqsubseteq S'$ iff $\forall s \in S, \exists s' \in S'$ s.t. $s = s'$.

**The Behavioral Cartography Algorithm.** 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. We recall in Algorithm 2 the behavioral cartography algorithm, as defined in [6].

---

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

---

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

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

---

In practice, most of (the real-valued space of) $V_0$ is covered by *Tiling* (see case studies in Section 4), although some "holes" (i.e., small zones containing no integer point) may sometimes remain uncovered by *Tiling*. Furthermore, the space covered by *Tiling* often largely exceeds the limits of $V_0$.

According to a given property on traces one wants to check, it is possible to partition trace sets between good and bad, and thus to partition the rectangle $V_0$ into a good subspace (union of good tiles) and a bad subspace (union of bad tiles).

The main advantage of this algorithm is that the cartography does not depend on the property one wants to check. Only the partition between good and bad tiles does. Moreover, the algorithm does not compute the set of all the reachable states; on the contrary, each call to *IM* quickly reduces the state space by removing the "bad" states. This allows us to overcome the state space explosion problem, which often prevents other methods, such as the computation of the whole set of reachable states to terminate.

# 3   Implementation

**Features.**   The input syntax of IMITATOR II to describe the network of PTAs modeling the system is given in [27], and is very close to the HYTECH syntax. IMITATOR II implements the ability to perform a full reachability analysis (computation of the set of all the reachable states), the inverse method algorithm, and the behavioral cartography algorithm.

When applying the inverse method, IMITATOR II takes as input a file describing the network of PTAs, and another file giving the reference valuation. It synthesizes a constraint solving the inverse problem, as well as the corresponding trace set under a graphical form (see example in Figure 1 left). The description of all the parametric reachable states is also returned.
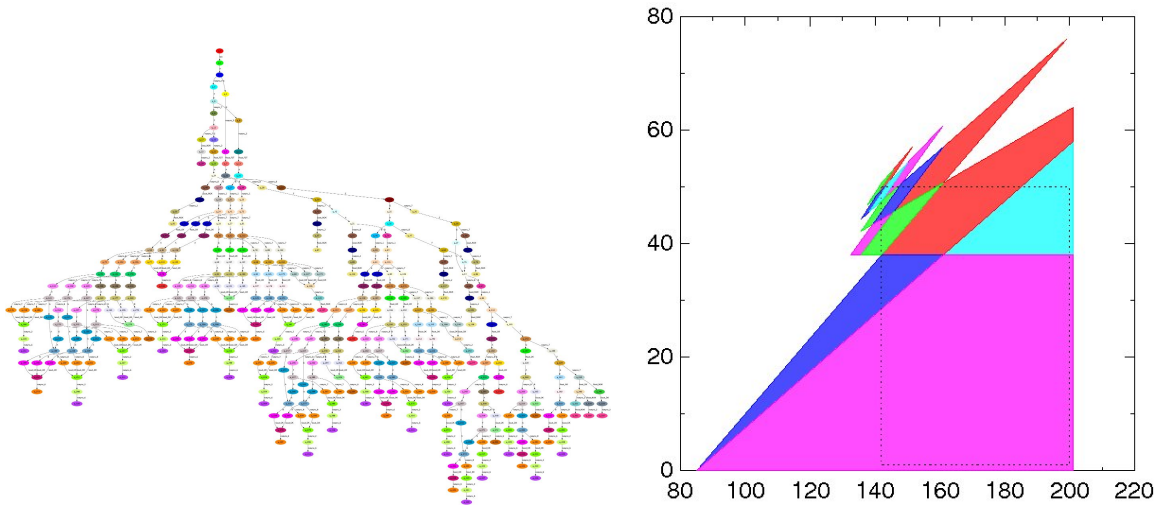


Figure 1: Examples of trace set (left) and of cartography (right)

When applying the behavioral cartography, IMITATOR II takes as input a file describing the network of PTAs, and another file giving the reference rectangle, i.e., the bounds to consider for each parameter. It synthesizes a list of tiles, as well as the trace set corresponding to each tile under a graphical form. For systems with only two parameter dimensions, the cartography is also returned under a graphical form (see example in Figure 1 right). Two different modes can be considered for *BC*: (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). As shown in Table 1, all those features (except the inverse method) are new features which were not available in IMITATOR.

| Tool | Inverse Method | Cartography | Computation of traces | Graphical output |
|---|---|---|---|---|
| IMITATOR | yes | no | no | no |
| IMITATOR II | yes | yes | yes | yes |

Table 1: Comparison of the features of IMITATOR and IMITATOR II

Among the options for the tool (see [27] for an exhaustive list), one can mention the possibility to add a limit for the depth of the *Post* operation, or for the execution time, and an option for acyclic systems avoiding to check whether a state has been computed before.

**Implementation.** IMITATOR II is a tool written in OCaml, making use of an external library for manipulating convex polyhedra, which can be, depending on the user's preference, either the NewPolka library, available in the APRON library [20], or the Parma Polyhedra Library (PPL) [9]. The trace sets, as well as the cartography for 2 parameter dimensions, are output under a graphical form using the DOT module of the graph visualization software Graphviz [28]. IMITATOR II contains about 9000 lines of code, and its development took about 6 man-months.

States are represented using a triple $(q, v, C)$ made of the current location $q$ in each automaton, a value for each discrete variable[1] $v$, 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 $(q, v, C_1)$, $\ldots$, $(q, v, C_n)$. Contrarily to HYTECH, IMITATOR II uses exact arithmetics with unlimited precision, and performs an *on-the-fly* composition of the automata, allowing to analyze bigger systems, and decreasing drastically the computation time compared to IMITATOR (see Section 4).

**Optimization.** Line 7 in Algorithm 1 corresponds to the computation of all the states reachable in up to $i$ steps from $s_0$, with the new constraint $K$ that has just been updated with the addition of some $\neg J$. However, this computation is redundant because no new state can be computed (because $K$ has been restrained with $\neg J$), and no state previously computed can be removed (because both $\neg J$ and the states previously computed are $\pi$-compatible). Instead, we simply update the set $S$ of states by adding $\neg J$ to all the states computed, by replacing line 7 in Algorithm 1 by the portion of algorithm given in Algorithm 3.

---

**Algorithm 3:** Modification of the Inverse Method Algorithm

1 **foreach** $(q, C) \in S$ **do**
2 $\quad \lfloor \ C \leftarrow C \wedge \neg J$

---

## 4  Case Studies

We present in this section a range of case studies of asynchronous circuits and communication protocols. The source code of IMITATOR II and various binaries, as well as the input file for all those case studies can be found in [27]. Experiments were conducted on an Intel Core2 Duo 2.4 GHz with 2 Gb.

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

**Inverse Method.**    The results of the application of the inverse method to various case studies are given in Table 2. We give from left to right the name of the example, the number of PTAs composing the global system $\mathscr{A}$, the lower and upper bounds on the number of locations per PTA, the number of clocks and parameters of $\mathscr{A}$, of iterations of the algorithm, of inequalities within $K_0$, of states and transitions, the computation time in seconds using IMITATOR, and the computation time in seconds using IMITATOR II.

| Example | PTAs | loc./PTA | $\|X\|$ | $\|P\|$ | iter. | $\|K_0\|$ | states | trans. | Time1 | Time2 |
|---|---|---|---|---|---|---|---|---|---|---|
| SR-latch | 3 | $[3,8]$ | 3 | 3 | 5 | 2 | 4 | 3 | 0.11 | 0.007 |
| Flip-flop [13] | 5 | $[4,16]$ | 5 | 12 | 9 | 6 | 11 | 10 | 1.6 | 0.122 |
| And–Or [12] | 3 | $[4,8]$ | 4 | 12 | 14 | 4 | 13 | 13 | 1.81 | 0.15 |
| Valmem Latch | 7 | $[2,5]$ | 8 | 13 | 12 | 6 | 18 | 17 | 14.4 | 0.345 |
| CSMA/CD [22] | 3 | $[3,8]$ | 3 | 3 | 19 | 2 | 219 | 342 | 41 | 1.01 |
| RCP [21] | 5 | $[6,11]$ | 6 | 5 | 20 | 2 | 327 | 518 | 64 | 2.3 |
| SPSMALL$_1$ [11] | 10 | $[3,8]$ | 10 | 26 | 32 | 23 | 31 | 30 | 4680 | 2.6 |
| BRP [24] | 6 | $[2,6]$ | 7 | 6 | 30 | 7 | 429 | 474 | 901 | 34 |
| SPSMALL$_2$ [11] | 28 | $[2,11]$ | 28 | 5 | 92 | 8 | 472 | 548 | - | 1755 |

Table 2: Summary of experiments for the inverse method

The SPSMALL case study corresponds to an asynchronous memory sold by ST-Microelectronics, and studied in the framework of VALMEM project. We considered two versions of this case study: the first one ("SPSMALL$_1$") was manually abstracted from the VHDL code (see [11]) and several gates have been merged into a single PTA. The second model ("SPSMALL$_2$") has been automatically generated from the VHDL code without any simplification. It is impossible to analyze SPSMALL$_2$ using the first version of IMITATOR because HYTECH runs out of memory when trying to statically compose the 28 automata in parallel.

The Valmem latch is an example of latch studied in the framework of VALMEM project.

Note that the computation time using IMITATOR II has dramatically decreased compared to IMITA-TOR for all examples: the time has been divided at least by 10, and up to 2000 for the SPSMALL$_1$ memory. Explanations for this high improvement are the rewriting of the tool using a library of convex polyhedra instead of the call to HYTECH, the on-the-fly composition of the different PTAs, and the optimization of the algorithm described in Section 3.


**Behavioral Cartography.**    The results of the application of the behavioral cartography algorithm to various case studies are given in Table 3. We give from left to right the name of the example, the number of PTAs composing the global system $\mathscr{A}$, the lower and upper bounds on the number of locations per PTA, the number of clocks of $\mathscr{A}$ (those first 4 columns are identical to Table 2), the number of parameters varying in the cartography, of integer points within $V_0$, of tiles computed, the average number per tile of states and transitions of the trace set, and the computation time in seconds using IMITATOR II (since the cartography is a new feature available in IMITATOR II only, no comparison is possible with IMITATOR).

For all those examples, the cartography covers 100 % of the real-valued space of $V_0$, except for the Root Contention Protocol (see Section 9), where "only" 99,99 % of $V_0$ is covered. Moreover, a significant part of the real-valued space outside $V_0$ is also covered.

Note that it is possible to find examples for which the algorithm $BC$ does not terminate for some $V_0$, because the algorithm $IM$ does not terminate for some $\pi \in V_0$. This is in particular the case of the "And–Or" circuit considered in [12], for a different $V_0$ from the one considered in Table 3.

| Example | PTAs | loc./PTA | $|X|$ | $|P|$ | $|V_0|$ | tiles | states | trans. | Time |
|---|---|---|---|---|---|---|---|---|---|
| SR-latch | 3 | $[3,8]$ | 3 | 3 | 1331 | 6 | 5 | 4 | 0.3 |
| Flip-flop [13] | 5 | $[4,16]$ | 5 | 2 | 644 | 8 | 15 | 14 | 3 |
| And–Or [12] | 3 | $[4,8]$ | 4 | 6 | 75600 | 4 | 64 | 72 | 118 |
| Valmem Latch | 7 | $[2,5]$ | 8 | 4 | 73062 | 5 | 21 | 20 | 96.3 |
| CSMA/CD [22] | 3 | $[3,8]$ | 3 | 3 | 2000 | 140 | 349 | 545 | 269 |
| RCP [21] | 5 | $[6,11]$ | 6 | 3 | 186050 | 19 | 5688 | 9312 | 7018 |
| SPSMALL$_1$ [11] | 10 | $[3,8]$ | 10 | 2 | 3149 | 259 | 60 | 61 | 1194 |

Table 3: Summary of experiments for the cartography algorithm

## 5   Conclusion

IMITATOR II allows us to solve the good parameters problem for timed automata by iterating the inverse method on the integer points of a real-valued 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 $V_0$. The tool has been successfully applied to various examples of asynchronous circuits and protocols.

**Ongoing and Future Work.**   Ulrich Kühne is currently extending IMITATOR II to hybrid systems, where clocks evolve at different rates. Romain Soulat is currently implementing variants and optimizations of *IM* in order to verify larger asynchronous memory circuits, in particular using an on-the-fly intersection of the constraints associated to the states, allowing to merge states. A variant of *IM* is also under implementation, where the fixpoint condition (line 8 of Algorithm 1) is modified as follows: instead of checking whether all new states are equal to states computed previously, we check whether all new states are *included* (in the sense of constraint inclusion) into former states.

Future work include the automatic partition into good and bad tiles, using an external tool such as UPPAAL. We are also studying a "dynamic" cartography, where the space unit between the selected points (so far, one integer) can be refined in order to fill the remaining holes. It would also be interesting to reason in a backward manner, i.e., considering a *Pre* operation instead of *Post* in Algorithm 1.

**Acknowledgments.**   I thank Bertrand Jeannet for his help to link IMITATOR II with APRON, Ulrich Kühne for the interface with PPL, and Daphné Dussaud for the graphical output for the cartography. I also thank anonymous referees for their helpful comments.

## References

[1]  R. Alur, C. Courcoubetis, T.A. Henzinger & P.H. Ho (1992): *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems*. In: *Hybrid Systems*, pp. 209–229.

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

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

[4]  T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise & W. Yi (2001): *UPPAAL - Now, Next, and Future*. In: *Proc. MOVEP'00, LNCS 2067*, Springer, pp. 99–124.

[5]  É. André, T. Chatain, E. Encrenaz & L. Fribourg (2009): *An Inverse Method for Parametric Timed Automata*. *International Journal of Foundations of Computer Science* 20(5), pp. 819–836.

[6]  É. André & L. Fribourg (2010): *Behavioral Cartography of Timed Automata*. In: *RP'10*, *LNCS* 6227, Springer. To appear.

[7]  Étienne André (2009): *IMITATOR: A Tool for Synthesizing Constraints on Timing Bounds of Timed Automata*. In: *ICTAC'09*, *LNCS* 5684, Springer, pp. 336–342.

[8]  A. Annichini, A. Bouajjani & M. Sighireanu (2001): *TReX: A Tool for Reachability Analysis of Complex Systems*. In: *CAV '01*, Springer-Verlag, pp. 368–372.

[9]  R. Bagnara, P. M. Hill & E. Zaffanella (2008): *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), pp. 3–21.

[10]  C. Baier & J.-P. Katoen (2008): *Principles of Model Checking*. The MIT Press.

[11]  R. Chevallier, E. Encrenaz, L. Fribourg & W. Xu (2009): *Timed Verification of the Generic Architecture of a Memory Circuit Using Parametric Timed Automata*. Formal Methods in System Design 34(1), pp. 59–81.

[12]  R. Clarisó & J. Cortadella (2005): *Verification of Concurrent Systems with Parametric Delays Using Octahedra*. In: *ACSD '05*, IEEE Computer Society.

[13]  R. Clarisó & J. Cortadella (2007): *The octahedron abstract domain*. Sci. Comput. Program. 64(1), pp. 115–139.

[14]  E. M. Clarke, O. Grumberg, S. Jha, Y. Lu & H. Veith (2000): *Counterexample-Guided Abstraction Refinement*. In: *CAV '00*, Springer-Verlag, pp. 154–169.

[15]  G. Frehse, S.K. Jha & B.H. Krogh (2008): *A Counterexample-Guided Approach to Parameter Synthesis for Linear Hybrid Automata*. In: *HSCC '08*, *LNCS* 4981, Springer, pp. 187–200.

[16]  G. Frehse, B.H. Krogh & R.A. Rutenbar (2006): *Verifying analog oscillator circuits using forward/backward abstraction refinement*. In: *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, European Design and Automation Association, pp. 257–262.

[17]  Goran Frehse (2005): *PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech*. In: *HSCC*, pp. 258–273.

[18]  T.A. Henzinger, P.H. Ho & H. Wong-Toi (1997): *HyTech: A Model Checker for Hybrid Systems*. Software Tools for Technology Transfer 1, pp. 460–463.

[19]  T.A. Henzinger & H. Wong-Toi (1996): *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.

[20]  B. Jeannet & A. Miné (2009): *Apron: A Library of Numerical Abstract Domains for Static Analysis*. In: *CAV '09*, *LNCS* 5643, Springer, pp. 661–667.

[21]  M. Kwiatkowska, G. Norman & J. Sproston (2003): *Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol*. Formal Aspects of Computing 14(3), pp. 295–318.

[22]  M. Kwiatkowska, G. Norman, J. Sproston & F. Wang (2007): *Symbolic Model Checking for Probabilistic Timed Automata*. Information and Computation 205(7), pp. 1027–1077.

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

[24]  P.R. D'Argenio, J.P. Katoen, T.C. Ruys & G.J. Tretmans (1997): *The Bounded Retransmission Protocol Must Be on Time!* In: *TACAS '97*, Springer.

[25]  Farn Wang (2006): *REDLIB for the Formal Verification of Embedded Systems*. In: *ISoLA*, pp. 341–346.

[26]  Python Web Page. `http://www.python.org/`.

[27]  IMITATOR II Web Page. `http://www.lsv.ens-cachan.fr/~andre/IMITATOR2`.

[28]  Graphviz Web Page. `http://www.graphviz.org/`.