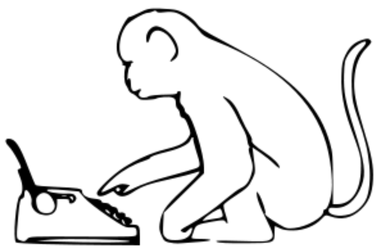
IMITATOR User Manual
Version 2.5.0

August 21, 2012



Abstract

IMITATOR is a tool for parameter synthesis in timed automata with stopwatches [5]. It is based on the inverse method: 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. 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

IMITATOR 2.5 (for *Inverse Method for Inferring Time AbstracT behaviOR*) is a tool for parameter synthesis in the framework of real-time systems based on the inverse method *IM* for Parametric Timed Automata (PTAs) augmented with stopwatches [5]. Different from CEGAR-based methods (see [8]), this algorithm for parameter synthesis makes use of a "good" parameter valuation π_0 instead of a set of "bad" states [3]. IMITATOR takes as input a network of PTAs with stopwatches and a reference valuation π_0; it synthesizes a constraint K on the parameters such that (1) $\pi_0 \models K$ and (2) for all parameter valuation π satisfying K, the trace set (i.e., the discrete behavior) of $\mathcal{A}$ under π is the same as for $\mathcal{A}$ under π_0. This provides the system with a criterion of *robustness* (see, e.g., [11]) around π_0.

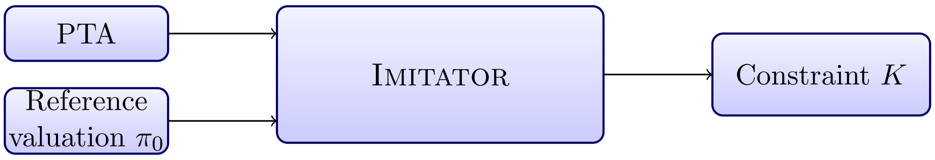


Figure 1: Functional view of IMITATOR

2 Installing IMITATOR

Sources, binaries for Linux platforms, and installation instructions are available on IMITATOR's Web page [1].

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 originally based on the HYTECH syntax, with several improvements. Actually, all standard HYTECH files describing only PTAs (and not more general systems like linear hybrid automata[2]) can be analyzed directly by IMITATOR with very minor changes.

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 optionally the corresponding trace set under a graphical form. The description of all the parametric reachable states can also be optionally output.

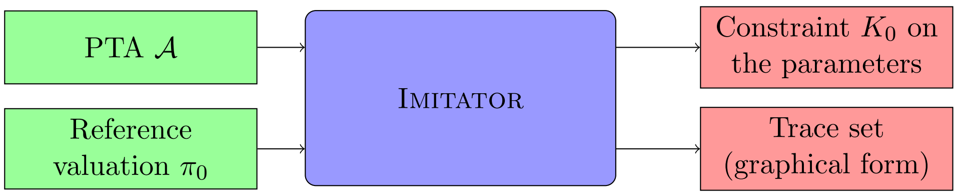

PTA $\mathcal{A}$
Reference valuation π_0
IMITATOR
Constraint K_0 on the parameters
Trace set (graphical form)

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 optionally the trace set corresponding to each tile under a graphical form. The description of all the parametric reachable states for each tile may also optionally be output.

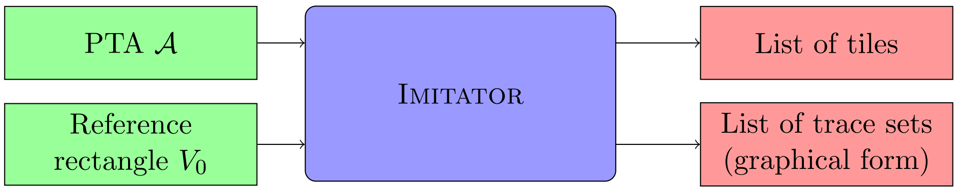

PTA $\mathcal{A}$
Reference rectangle V_0
IMITATOR
List of tiles
List of trace sets (graphical form)

Figure 3: IMITATOR inputs and outputs in behavioral cartography mode

3.2 Features

IMITATOR (version 2.5.0) includes the following features:

- Analysis of a network of parametric timed automata augmented with stopwatches and discrete rational-valued variables;
- 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);
- 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;
- Optional use of the merging technique of [6].

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) [7]
- The GNU Multiple Precision Arithmetic Library (GMP)

Binaries and source code packages are available on Imitator's Web page [1].

4.2 The Imitator Input File

Examples of input files can be found on Imitator's Web page [1]. A complete example is given in Appendix A. A tentative grammar is given in Appendix B.

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 [3].
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 [4].

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).

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 [3]. 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 [3], 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.

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 [4]. 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

`-depth-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.

`-fancy` (default: `false`) In the graphical output of the reachable states (see option `-with-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-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.

`-states-limit` (default: `none`) Will try to stop after reaching this number of states. Warning: the program may have to first finish computing the current iteration before stopping.

`-statistics` (default: `false`) Print info on number of calls to PPL, and other statistics about memory and time. Warning: enabling this option may slow down the analysis, and will certainly induce some extra computational time at the end.

`-step` (default: `1`) Step for the behavioral cartography.

`-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[1].

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

[1]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.

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-dot` **(default: `false`)** Graphical output using DOT. In this case, IMITATOR outputs a file `<input_file>.jpg`, which is a graphical output in the jpg format, generated using DOT, corresponding to the trace set.

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

`-with-log` **(default: `false`)** Generation of the files describing the reachable states. In this case, IMITATOR outputs 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.

`-with-merging` **(default: `false`)** Use the merging technique of [6].

`-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).

4.3.5 Examples of Calls

`IMITATOR flipflop.imi -mode reachability` Computes a reachability analysis on the automata described in file `flipflop.imi`.

`IMITATOR flipflop.imi -mode reachability -with-dot -with-log` 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.jpg` 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.

`IMITATOR flipflop.imi flipflop.pi0 -with-log -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 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.

`IMITATOR SRlatch.imi SRlatch.v0 -mode cover -with-dot -with-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 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`.jpg`) giving the description of the states (resp. the reachability graph) of tile i, for $i = 1, \dots, n$.

`IMITATOR SRlatch.imi SRlatch.v0 -mode random100 -with-dot` 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`.jpg` giving the reachability graph of tile i, for $i = 1, \dots, n$.

5 Example: SR-latch

We consider a SR-latch described in, e.g., [9], and depicted on Fig. 4 left. The possible configurations of the latch are the following ones:

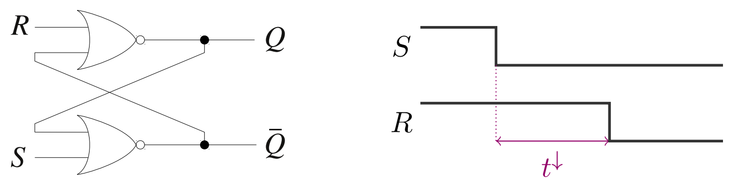


Figure 4: SR latch (left) and environment (right)

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. 4, 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. 5, 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:

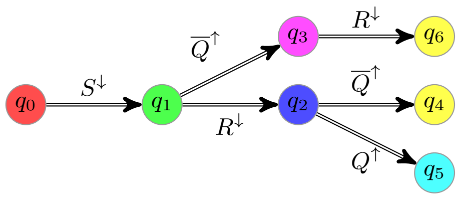


Figure 5: Parametric reachability analysis of the SR latch

$$\begin{aligned} t^{\downarrow} &\in [0, 10] \\ \delta_1 &\in [0, 10] \\ \delta_2 &\in [0, 10] \end{aligned}$$

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 in the `jpg` format by IMITATOR using the `-with-dot` option, 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. 6.

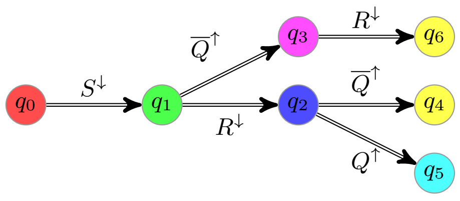


Figure 6: Trace set of tile 1 for the SR latch

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. 7.

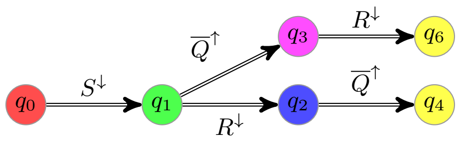


Figure 7: Trace set of tile 2 for the SR latch

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, $Q^{\uparrow}$ can not occur (since $\delta_1 > 0$), so $\overline{Q}^{\uparrow}$ occurs immediately after $R^{\downarrow}$, which leads to stability.

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. 8.

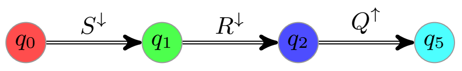


Figure 8: Trace set of tile 3 for the SR latch

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. 9.

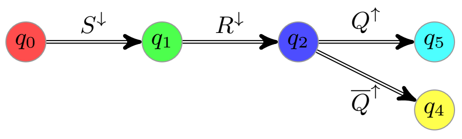


Figure 9: Trace set of tile 4 for the SR latch

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. 10.

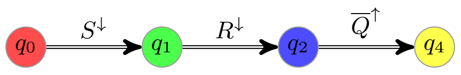


Figure 10: Trace set of tile 5 for the SR latch

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 $\overline{Q}^{\uparrow}$.

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. 11.

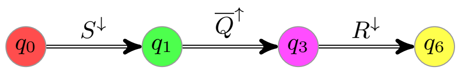


Figure 11: Trace set of tile 6 for the SR latch

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

Cartography. We give in Fig. 12 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 $t^{\downarrow} = 1$.

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.

Acknowledgments

Étienne André instigated the development of IMITATOR, and keeps developing it. Emmanuelle Encrenaz and Laurent Fribourg have been great contributors of IMITATOR, on a theoretical point of view, and to find applications

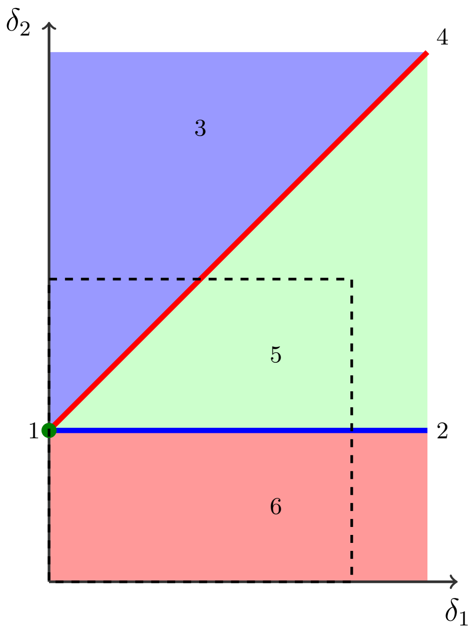


Figure 12: Behavioral cartography of the SR latch according to δ_1 and δ_2

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 linking with Apron [10] on a previous version of IMITATOR. Ulrich Kühne made several important improvements to IMITATOR, and linked the tool to PPL. Daphne Dussaud implemented the graphical output of the behavioral cartography. Romain Soulat implemented powerful algorithmic optimizations, and brought many case studies.

IMITATOR's logo comes from KaterBegemot's `Typing monkey.svg` (Licence: Creative Commons Attribution-Share Alike 3.0 Unported).

References

[1] IMITATOR web page. http://www.lsv.ens-cachan.fr/Software/imitator/. 3, 5

[2] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992. 3

[3] 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. 3, 6

[4] É. André and L. Fribourg. Behavioral cartography of timed automata. In *RP'10*, volume 6227 of *LNCS*, pages 76–90. Springer, 2010. 6, 7

[5] É. André, L. Fribourg, U. Kühne, and R. Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM'12*, volume 7436 of *LNCS*, pages 33–36, Paris, France, 2012. Springer. 1, 3

[6] É. André, L. Fribourg, and R. Soulat. Enhancing the inverse method with state merging. In *NFM'12*, volume 7226 of *LNCS*, pages 100–105. Springer, 2012. 5, 10

[7] 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. 5

[8] 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. 3

[9] D. Harris and S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. 11

[10] 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. 16

[11] N. Markey. Robustness in real-time systems. In *SIES'11*, pages 28–34. IEEE, 2011. 3

A Source Code of the Example

A.1 Main Input File

```
--************************************************************--
--************************************************************--
--   Laboratoire Specification et Verification
--
--   Race on a digital circuit (SR Latch)
--
--   Etienne ANDRE
--
--   Created:        2010/03/19
--   Last modified : 2010/03/24
--************************************************************--
--************************************************************--

var     ckNor1, ckNor2, s
                : clock;

        dNor1_l, dNor1_u,
        dNor2_l, dNor2_u,
        t_down
                        : parameter;


--************************************************************--
  automaton norGate1
--************************************************************--
synclabs: R_Up, R_Down, overQ_Up, overQ_Down,
        Q_Up, Q_Down;
initially Nor1_110;

-- UNSTABLE
loc Nor1_000: while ckNor1 <= dNor1_u wait {}
        when True sync R_Up do {} goto Nor1_100;
        when True sync overQ_Up do {} goto Nor1_010;
        when ckNor1 >= dNor1_l sync Q_Up do {} goto
            Nor1_001;

-- STABLE
loc Nor1_001: while True wait {}
        when True sync R_Up do {ckNor1' = 0} goto
            Nor1_101;
        when True sync overQ_Up do {ckNor1' = 0} goto
           Nor1_011;

-- STABLE
loc Nor1_010: while True wait {}
        when True sync R_Up do {} goto Nor1_110;
        when True sync overQ_Down do {ckNor1' = 0}
           goto Nor1_000;

-- UNSTABLE
loc Nor1_011: while ckNor1 <= dNor1_u wait {}
        when True sync R_Up do {ckNor1' = 0} goto
           Nor1_111;
        when True sync overQ_Down do {} goto Nor1_001;
        when ckNor1 >= dNor1_l sync Q_Down do {} goto
           Nor1_010;

-- STABLE
loc Nor1_100: while True wait {}
        when True sync R_Down do {ckNor1' = 0} goto
           Nor1_000;
        when True sync overQ_Up do {} goto Nor1_110;

-- UNSTABLE
loc Nor1_101: while ckNor1 <= dNor1_u wait {}
        when True sync R_Down do {} goto Nor1_001;
        when True sync overQ_Up do {ckNor1' = 0} goto
           Nor1_111;
        when ckNor1 >= dNor1_l sync Q_Down do {} goto
           Nor1_100;

-- STABLE
loc Nor1_110: while True wait {}
        when True sync R_Down do {} goto Nor1_010;
        when True sync overQ_Down do {} goto Nor1_100;

-- UNSTABLE
loc Nor1_111: while ckNor1 <= dNor1_u wait {}
        when True sync R_Down do {ckNor1' = 0} goto
           Nor1_011;
        when True sync overQ_Down do {ckNor1' = 0}
           goto Nor1_101;
        when ckNor1 >= dNor1_l sync Q_Down do {} goto
           Nor1_110;
```

```

end -- norGate1


--************************************************************--
  automaton norGate2
--************************************************************--
synclabs: Q_Up, Q_Down, S_Up, S_Down,
        overQ_Up, overQ_Down;
-- initially Nor2_110;
initially Nor2_001;

-- UNSTABLE
loc Nor2_000: while ckNor2 <= dNor2_u wait {}
        when True sync Q_Up do {} goto Nor2_100;
        when True sync S_Up do {} goto Nor2_010;
        when ckNor2 >= dNor2_l sync overQ_Up do {}
           goto Nor2_001;

-- STABLE
loc Nor2_001: while True wait {}
        when True sync Q_Up do {ckNor2' = 0} goto
           Nor2_101;
        when True sync S_Up do {ckNor2' = 0} goto
           Nor2_011;

-- STABLE
loc Nor2_010: while True wait {}
        when True sync Q_Up do {} goto Nor2_110;
        when True sync S_Down do {ckNor2' = 0} goto
           Nor2_000;

-- UNSTABLE
loc Nor2_011: while ckNor2 <= dNor2_u wait {}
        when True sync Q_Up do {ckNor2' = 0} goto
           Nor2_111;
        when True sync S_Down do {} goto Nor2_001;
        when ckNor2 >= dNor2_l sync overQ_Down do {}
           goto Nor2_010;

-- STABLE
loc Nor2_100: while True wait {}
        when True sync Q_Down do {ckNor2' = 0} goto
           Nor2_000;
        when True sync S_Up do {} goto Nor2_110;

-- UNSTABLE
loc Nor2_101: while ckNor2 <= dNor2_u wait {}
        when True sync Q_Down do {} goto Nor2_001;
        when True sync S_Up do {ckNor2' = 0} goto
            Nor2_111;
        when ckNor2 >= dNor2_l sync overQ_Down do {}
            goto Nor2_100;

-- STABLE
loc Nor2_110: while True wait {}
        when True sync Q_Down do {} goto Nor2_010;
        when True sync S_Down do {} goto Nor2_100;

-- UNSTABLE
loc Nor2_111: while ckNor2 <= dNor2_u wait {}
        when True sync Q_Down do {ckNor2' = 0} goto
            Nor2_011;
        when True sync S_Down do {ckNor2' = 0} goto
            Nor2_101;
        when ckNor2 >= dNor2_l sync overQ_Down do {}
            goto Nor2_110;

end -- norGate2


--************************************************************--
  automaton env
--************************************************************--
synclabs: R_Down, R_Up, S_Down, S_Up;
initially env_11;

-- ENVIRONMENT : first S then R at constant time
loc env_11: while True wait {}
        when True sync S_Down do {s' = 0} goto env_10;

loc env_10: while s <= t_down wait {}
        when s = t_down sync R_Down do {} goto
            env_final;

loc env_final: while True wait {}

end -- env
```

```

--*********************************************************--
-- ANALYSIS
--*********************************************************--

var init : region;

init := True
		------------------------------------------------------------
		-- INITIAL LOCATIONS
		------------------------------------------------------------
		-- S and R down
		& loc[norGate1] = Nor1_100
		& loc[norGate2] = Nor2_010
		& loc[env]			= env_11

		------------------------------------------------------------
		-- INITIAL CONSTRAINTS
		------------------------------------------------------------
		& ckNor1		= 0
		& ckNor2		= 0
		& s				= 0

		& dNor1_l >= 0
		& dNor2_l >= 0

		& dNor1_l <= dNor1_u
		& dNor2_l <= dNor2_u
;
```

A.2 V_0 File

```
		------------------------------------------------------------
		-- V0
		------------------------------------------------------------
		& dNor1_l		= 3
		& dNor1_u		= 3 .. 20
		& dNor2_l		= 5
		& dNor2_u		= 5 .. 20
		& 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 typewriter 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

⟨sync_labels⟩ ::
    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⟩ ⟨stop_opt⟩ ⟨wait_opt⟩ ⟨transitions⟩

⟨wait_opt⟩ ::
     wait()
  |  wait
  |  ϵ

⟨stop_opt⟩ ::
     stop{ ⟨var_list⟩ }
  |  ϵ

⟨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⟩ ::
    <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⟩
      ⟨float⟩
  |   ⟨integer⟩ / ⟨pos_integer⟩

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

⟨pos_integer⟩ ::
     <int>

⟨float⟩ ::
      ⟨pos_float⟩
  |   - ⟨pos_float⟩

⟨pos_float⟩ ::
     <float>
```

B.1.2 Initial State

```
⟨init⟩ ::
      ⟨init_declaration⟩ ⟨init_definition⟩ ⟨bad_definition⟩ ⟨reach_command⟩
  |   ⟨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*⟩

⟨*bad_definition*⟩ ::
   `bad :=` ⟨*loc_expression*⟩ `;`

⟨*loc_expression*⟩ ::
    ⟨*loc_predicate*⟩
  | ⟨*loc_predicate*⟩ `&` ⟨*loc_expression*⟩
  | ⟨*loc_predicate*⟩ ⟨*loc_expression*⟩

⟨*loc_predicate*⟩ ::
   `loc[ <name> ] = <name>`

B.2 Reserved Words

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

`and`, `automaton`, `bad`, `clock`, `discrete`, `do`, `end`, `False`, `goto`, `if`, `init`, `initially`, `loc`, `locations`, `not`, `or`, `parameter`, `region`, `sync`, `stop`, `synclabs`, `True`, `var`, `wait`, `when`, `while`