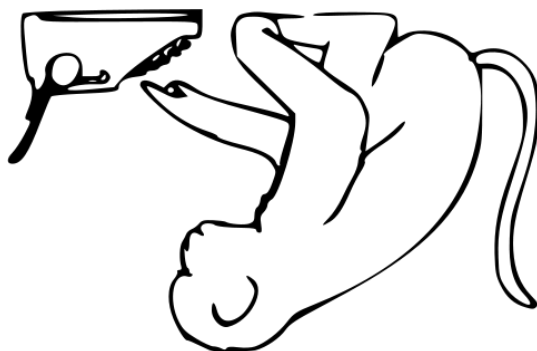


# IMITATOR User Manual



Version 3.3 (Cheese Caramel au beurre salé)



July 21, 2022

[www.imitator.fr](http://www.imitator.fr)



# Contents

Table of contents . . . . .	4
<b>1 Introduction</b>	<b>5</b>
<b>2 A brief introduction to the syntax</b>	<b>6</b>
2.1 Generalities . . . . .	6
2.2 Model syntax . . . . .	6
2.3 Property syntax . . . . .	12
<b>3 IMITATOR Parametric Timed Automata</b>	<b>14</b>
3.1 Formal definition . . . . .	14
3.1.1 Clocks, parameters . . . . .	14
3.1.2 Discrete rational variables . . . . .	14
3.1.3 Linear constraints . . . . .	15
3.1.4 Arithmetic expressions . . . . .	15
3.1.5 IMITATOR Parametric Timed Automata . . . . .	15
3.1.6 Networks of IMITATOR Parametric Timed Automata . . . . .	18
3.2 Initial state and initialization of variables . . . . .	19
3.3 Synchronization model . . . . .	19
3.4 Global constants . . . . .	20
3.5 Discrete variables . . . . .	20
3.5.1 Types . . . . .	20
3.5.2 Default initial value . . . . .	30
3.5.3 Updates . . . . .	30
3.5.4 Runtime errors . . . . .	34
<b>4 Parameter synthesis using IMITATOR</b>	<b>36</b>
4.1 Synthesis and emptiness . . . . .	36
4.2 Reachability . . . . .	36
4.3 Safety . . . . .	37
4.4 EF-minimization . . . . .	38
4.5 EF-maximization . . . . .	38
4.6 EF with minimal time reachability . . . . .	38
4.7 Parameter synthesis using patterns . . . . .	38
4.8 Parametric deadlock-freeness checking . . . . .	39
4.9 Parametric cycle synthesis . . . . .	40
4.9.1 Accepting cycle synthesis . . . . .	40

4.9.2	Accepting cycle with generalized acceptance condition (BFS)	41
4.9.3	Any cycle synthesis	42
4.10	Parametric non-Zeno cycle synthesis	42
4.11	Inverse method: Trace preservation and robustness	43
4.12	Behavioral cartography	44
4.13	Parametric reachability preservation	45
4.14	Summary	46
4.15	Symbolic state space computation	46
<b>5</b>	<b>Understanding the IMITATOR result</b>	<b>48</b>
5.1	Header	48
5.2	The resulting constraint	48
5.3	The cartography result	49
5.4	General statistics	49
5.5	Projection onto some parameters	49
<b>6</b>	<b>Graphical output and translation</b>	<b>50</b>
6.1	State space	50
6.2	Visualizing the synthesized constraint in 2D	51
6.3	Translation to UPPAAL	51
6.4	Translation to HYTECH	54
6.5	Translation to JANI	55
6.6	Export to graphics	56
6.7	Export to $\LaTeX$	57
<b>7</b>	<b>Inside the box</b>	<b>58</b>
7.1	Language and libraries	58
7.2	Symbolic states	58
7.3	Type system	59
7.3.1	Type checking	59
7.3.2	Expression type solving	59
7.3.3	Literal number type inference	60
7.3.4	Type conversion	60
7.4	Installation	60
<b>8</b>	<b>List of options</b>	<b>61</b>
<b>9</b>	<b>Grammar</b>	<b>70</b>
9.1	Variable names	70
9.2	Grammar of the model	70
9.2.1	Automata descriptions	70
9.2.2	Initial state	78
9.3	Grammar of the property file	79
9.4	Reserved words	83

<b>10 Missing features</b>	<b>85</b>
10.1 ASAP transitions . . . . .	85
10.2 Parameterized models . . . . .	85
10.3 Other synchronization models . . . . .	85
10.4 Initial intervals for discrete variables . . . . .	86
10.5 Complex updates for discrete variables . . . . .	86
10.6 Synthesis for L/U-PTA . . . . .	86
<b>11 Acknowledgments</b>	<b>87</b>
<b>12 Licensing and credits</b>	<b>88</b>
<b>References</b>	<b>91</b>

# Chapter 1

## Introduction

IMITATOR is a tool to perform automated parameter synthesis for concurrent timed systems [And21]. IMITATOR takes as input a network of IMITATOR parametric timed automata (NIPTA): NIPTA are an extension of parametric timed automata [AHV93], a formalism to specify and verify models of systems where timing constants can be replaced with parameters, *i.e.*, unknown constants.

IMITATOR addresses several variants of the following problem: “given a concurrent timed system, what are the values of the timing constants that guarantee that the model of the system satisfies some property?” Among other algorithms, IMITATOR implements parametric safety and reachability analysis [AHV93; JLR15], parametric liveness synthesis (with Büchi conditions) [NPV18; And+21], parametric deadlock-freeness synthesis [And16], robustness analysis (also known as the inverse method) [And+09; ALM20], the behavioral cartography [AF10], and parametric reachability preservation [And+15]. Some algorithms can also run distributed on a cluster. Numerous analysis options are available.

IMITATOR is a command-line only tool, but that can output results in graphical form.

IMITATOR was able to verify numerous case studies from the literature and from the industry, such as communication protocols, hardware asynchronous circuits, schedulability problems and various other systems such as models of coffee machines (probably the most critical systems from a researcher point of view). Numerous benchmarks are available at IMITATOR Web page [IMI-Web]. An official benchmarks library is available at [AMP21].

IMITATOR is free and open source software.

In this document, we present the input syntax, we formally define the input model of IMITATOR, and we explain how to perform various analyses using the numerous options.

**Keywords:** formal verification, model checking, software verification, parametric timed automata, parameter synthesis

## Chapter 2

# A brief introduction to the syntax

We first briefly introduce the syntax using a simple example for readers familiar with parametric timed automata, and not interested in subtle details (such as the synchronization model). A formal (and nearly exhaustive) definition of IMITATOR parametric timed automata (NIPTA) can be found in [Chapter 3](#). The complete syntax is given in [Chapter 9](#).

### 2.1 Generalities

IMITATOR performs parametric verification of models specified using networks of IMITATOR parametric timed automata (hereafter NIPTA). An IMITATOR parametric timed automaton (hereafter IPTA) is a variant of parametric automata (as introduced in [\[AHV93\]](#)). IPTA and NIPTA are formalized in [Section 3.1](#).

The input syntax of IMITATOR is originally based on the syntax of HYTECH [\[HHW95\]](#), with several improvements. Still nowadays, the syntax of HYTECH (and PHAVerLite, reusing itself in part the HYTECH) models is remarkably close to IMITATOR.

Comments are OCaml-like comments starting with `(*` and ending with `*)`. As in OCaml, comments can be nested.

**The Fischer mutual exclusion protocol** We use as a motivating example one timed version of the Fischer mutual exclusion protocol, coming from the PAT model checker [\[Sun+09\]](#). This version of the protocol is neither the most complete, nor the most simple; we just use it here to introduce various aspects of the IMITATOR input syntax.

Fischer mutual exclusion protocol is a protocol that guarantees the mutual exclusion of several processes (here two) that want to access a shared resource (called the critical section).

### 2.2 Model syntax

We give below this model using the IMITATOR syntax. This model is given in graphical form in [Fig. 2.1](#).<sup>1</sup>

---

<sup>1</sup>This  $\text{\LaTeX}$  representation, that makes use of the  $\text{\LaTeX}$  TikZ library, was automatically output by IMITATOR, using option `-imi2TikZ`, followed by some manual positioning optimization.

```

(*****
*
*                               IMITATOR MODEL
*
* Fischer's mutual exclusion protocol
*
* Description      : Fischer's mutual exclusion protocol with 2 processes
* Correctness      : Not 2 processes together in the critical section (location
*                   obs_violation unreachable)
* Source           : PAT library of benchmarks
* Author           : ?
* Input by        : Étienne André
*
* Created          : 2012/10/08
* Last modified   : 2021/10/14
*
* IMITATOR version: 3.2
*****)

var
  x1, (* proc1's clock *)
  x2, (* proc2's clock *)
    : clock;

  turn,
  counter
    : int;

  delta,
  gamma
    : parameter;

  IDLE = -1
    : int;

(*****)
automaton proc1
(*****)
synclabs: access_1, enter_1, exit_1, no_access_1, try_1, update_1;

loc idle1: invariant True
  when turn = IDLE sync try_1 do {x1 := 0} goto active1;

loc active1: invariant x1 <= delta
  when True sync update_1 do {turn := 1, x1 := 0} goto check1;

loc check1: invariant True
  when x1 >= gamma & turn = 1 sync access_1 do {x1 := 0} goto access1;
  when x1 >= gamma & turn <> 1 sync no_access_1 do {x1 := 0} goto idle1;

loc access1: invariant True
  when True sync enter_1 do {counter := counter + 1} goto CS1;

loc CS1: invariant True
  when True sync exit_1 do {counter := counter - 1, turn := IDLE, x1 := 0} goto idle1;

```

```

end (* proc1 *)

(*****)
automaton proc2
(*****)
synclabs: access_2, enter_2, exit_2, no_access_2, try_2, update_2;

loc idle2: invariant True
  when turn = IDLE sync try_2 do {x2 := 0} goto active2;

loc active2: invariant x2 <= delta
  when True sync update_2 do {turn := 2, x2 := 0} goto check2;

loc check2: invariant True
  when x2 >= gamma & turn = 2 sync access_2 do {x2 := 0} goto access2;
  when x2 >= gamma & turn <> 2 sync no_access_2 do {x2 := 0} goto idle2;

loc access2: invariant True
  when True sync enter_2 do {counter := counter + 1} goto CS2;

loc CS2: invariant True
  when True sync exit_2 do {counter := counter - 1, turn := IDLE, x2 := 0} goto idle2;

end (* proc2 *)

(*****)
automaton observer
(*****)
synclabs : enter_1, enter_2, exit_1, exit_2;

loc obs_waiting: invariant True
  when True sync enter_1 goto obs_1;
  when True sync enter_2 goto obs_2;

loc obs_1: invariant True
  when True sync exit_1 goto obs_waiting;
  when True sync enter_2 goto obs_violation;

loc obs_2: invariant True
  when True sync exit_2 goto obs_waiting;
  when True sync enter_1 goto obs_violation;

(* NOTE: no outgoing action to reduce state space *)
loc obs_violation: invariant True

end (* observer *)

(*****)
(* Initial state *)
(*****)

init := {

```

55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110



```

111 discrete =
112 (*-----*)
113 (* Initial location *)
114 (*-----*)
115 loc[proc1] := idle1 ,
116 loc[proc2] := idle2 ,
117 loc[observer] := obs_waiting ,
118 (*-----*)
119 (* Initial discrete assignments *)
120 (*-----*)
121 turn := IDLE,
122 counter := 0
123 ;
124
125 continuous =
126 (*-----*)
127 (* Initial clock constraints *)
128 (*-----*)
129 & x1 >= 0
130 & x2 >= 0
131
132 (*-----*)
133 (* Parameter constraints *)
134 (*-----*)
135 & delta >= 0
136 & gamma >= 0
137 ;
138
139 }
140
141
142 (*-----*)
143 (* The end *)
144 (*-----*)
145
146 end

```

**Header** Let us comment this case model by starting with the header. First, text in comments gives generalities about the model (author, date, description, etc.). The form is not normalized, but it could be in the future, so it is strongly advised to follow this form.<sup>2</sup>

**Variable declarations** The variable declarations starts with keyword `var`.

This model contains two clocks: `x1` is process 1's clock, and `x2` is process 2's clock.

This model contains two parameters: `delta` is the parametric duration specifying how long a process is idle at most, whereas `gamma` is the parametric duration specifying the minimum duration between the time a process checks for the availability of the critical section and the time the same process indeed enters the critical section (if it is still available).

<sup>2</sup>An empty model template with all these comments ready to be filled out (containing also a sample IPTA and its initial definitions) is available at:

<https://github.com/imitator-model-checker/imitator/blob/master/benchmarks/template.imi>.

Two integer-valued variables (*i.e.*, global, discrete variables, see [Section 3.5](#)) are used: `turn` checks which process is attempting to enter the critical section; `counter` records how many processes are in the critical section (this variable will not be used for the verification, but was used in the original PAT model, and we choose to keep it).

Finally, a global constant `IDLE` is set to `-1` (just as in the original PAT model), and encodes that no process is attempting to enter the critical section.

**Automata** This model contains three IPTA: the first and second ones (`proc1` and `proc2`) model the first and second process, respectively. The third one (`observer`) is an observer, *i.e.*, an IPTA that checks the system behavior without modifying it.

**The first process** Let us first describe the IPTA `proc1` (a graphical representation is given in [Fig. 2.1a](#)). This IPTA uses six actions, given in the `synclabs` declaration.

`proc1` is initially in location `idle1`, with no invariant (depicted by `invariant True`). At any time, when the discrete variable `turn` is equal to `IDLE`, then this IPTA may synchronize on action `try_1`, reset its clock `x1`, and enter location `active1`.

The invariant of this location is `x1 <= delta`, *i.e.*, `proc1` can only remain in `active1` as long as the value of `x1` does not exceed `delta`. At any time, this IPTA may synchronize on action `update_1`, reset its clock `x1` and set the global variable `turn` to `1`, and enter location `check1`.

In location `check1`, the process wait at least `gamma` time units (modeled by the inequality `x1 >= gamma`, in all outgoing transitions). If `turn` is still equal to `1` (that is, no other process attempted in the meanwhile to enter the critical section), then process 1 is indeed ready to enter the critical section, by synchronizing `access_1` and resetting `x1`. If `turn` is different from `1` (that is, another process attempted in the meanwhile to enter the critical section, and it is not safe for process 1 to enter), then process 1 returns to its idle location, by synchronizing `no_access_1` and resetting `x1`.

In location `access1`, process 1 can remain any time, and eventually enters the critical section by synchronizing `enter_1` and incrementing the global variable `counter` by 1.

In location `CS1`, process 1 can remain any time, and eventually leaves it, by decrementing the global variable `counter` by 1, and setting the global variable `turn` to its initial value `IDLE`.

**The second process** Process 2 is identical to process 1, except that `x1` is replaced with `x2`, and that the value of `turn` becomes 2.

**The observer** The observer is in charge to check that no more than one process is in critical section at the same time.<sup>3</sup> This observer will detect that this situation happens if an action `enter_1` is followed by an action `enter_2` without an action `exit_1` in between (or symmetrically if an action `enter_2` is followed by an action `enter_1` without an action `exit_2` in

<sup>3</sup>This observer is not really necessary to check the correctness of this protocol; instead of adding this observer and checking `unreachable loc[observer] = obs_violation`, one could just check either `counter > 1` or `loc[proc1] = CS1 & loc[proc2] = CS2`. However, this example comes from an earlier version of IMITATOR (that did not support checking global variables or more than one location in the `unreachable` property — which has been fixed since IMITATOR 2.7); furthermore, introducing an observer is also useful, as it is often used for the verification of more complex properties (see, *e.g.*, [ABL98; Ace+98]).

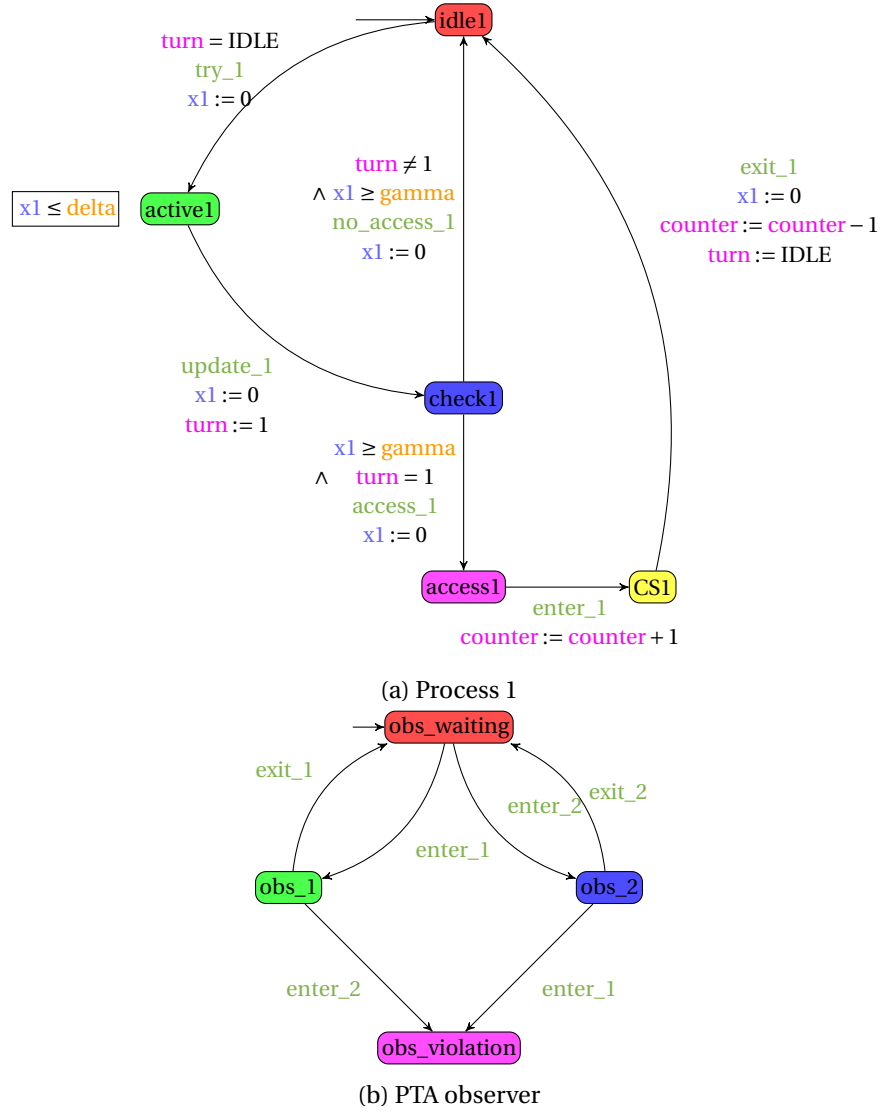


Figure 2.1: Fischer mutual exclusion protocol (graphical NIPTA)

between). Note that the observer simply observes the system state, and synchronizes on the actions used by `proc1` and `proc2`; it does not use any clock nor variable.

A graphical representation of the IPTA `observer` is given in Fig. 2.1b.

**Initial state definition** The initial state is defined by the part of the file following `init :=`.

Since IMITATOR 3.1, the initial definition section is split between the *discrete* initialization and the *continuous* initialization (the latter being given in the form of a set of constraints).

The discrete initialization (introduced by the `discrete` keyword) assigns an initial value to each discrete (global) variable, and sets the initial location for each automaton. Note that the assignment operator `:=` is used (in contrast with unification operator `=` used for constraints). For example, `loc[proc1] := idle1` states that `proc1` is initially in location `idle1`. The initial definition should assign a constant value to each discrete variable: here `turn` is initially assigned to `IDLE`, and `counter` is initially assigned to 0.

The continuous initialization (introduced by the `continuous` keyword) defines the initial constraints over clocks and parameters (possibly also using discrete variables). The initial definition may (only may, see Section 3.2) give an initial value to the clocks, for example requiring them to be equal to some constant (typically 0). In our example, clocks are only bound to be greater or equal to 0. Finally, in this example, parameters are bound to be non-negative as well. Note that the initial definition can introduce more complex constraints over clocks, parameters and discrete variables; see Section 3.2 for details.

## 2.3 Property syntax

**Property specification** For this model, the correctness property is that two processes cannot be in the critical section at the same time; as explained above, this is equivalent to the fact that the `obs_violation` location of the `observer` IPTA is unreachable. This is input in the property as follows:

```
property := #synth AGnot(loc[observer] = obs_violation);
```

Here, `AGnot` stands for safety. More elaborate properties are detailed in Section 4.7.

We give below the whole property file:

```
(*****
 *                               IMITATOR MODEL
 *
 * Fischer's mutual exclusion protocol
 *
 * Description      : Fischer's mutual exclusion protocol with 2 processes
 * Correctness      : Not 2 processes together in the critical section (location
 *                   obs_violation unreachable)
 * Source           : PAT library of benchmarks
 * Author           : ?
 * Input by         : Étienne André
 *
 * Created          : 2012/10/08
 * Last modified    : 2020/04/03
 *
 * IMITATOR version: 3
 *****)
```



```
property := #synth AGnot(loc[observer] = obs_violation);
```

**Parameter synthesis** Finally, let us run IMITATOR on this case study. Quite naturally, what we would be interested in is knowing for which parameter valuations this protocol is correct, *i.e.*, no more than one process can be present in the critical section at one time. Assuming this model is input in file `fischer.imi` and the property is in file `fischer.imiprop`, the command calling IMITATOR is as follows:

```
imitator fischer.imi fischer.imiprop
```

The result of the call to IMITATOR is

```
Final positive constraint guaranteeing safety:
delta >= 0
& gamma > delta
This positive constraint is exact (sound and complete)
```

That is, the system is safe if  $0 \leq \text{delta} < \text{gamma}$ , which is the well-known constraint ensuring mutual exclusion for this protocol.

## Chapter 3

# IMITATOR Parametric Timed Automata

This chapter formally introduces the input model of IMITATOR.

### 3.1 Formal definition

IMITATOR performs parametric verification of models specified using networks of IMITATOR parametric timed automata (hereafter NIPTA).

An IMITATOR parametric timed automaton (hereafter IPTA) is a variant of parametric automata (as introduced in [AHV93]). IPTA augment the expressiveness of PTA from [AHV93] with several features such as invariants, discrete variables which can be simple (*e.g.*, integer, rational, Boolean variables) or more complex (lists, arrays...), complex guards and invariants (*i.e.*, not only comparing a single clock to a single parameter), stopwatches (*i.e.*, the ability to stop some clocks in some locations), multi-rate clocks (*i.e.*, the speed of a clock is not necessarily 1, and can be defined to any rational) and arbitrary clock updates (*i.e.*, not necessarily to 0, but also to rationals, other clock valuations or even parameters).

#### 3.1.1 Clocks, parameters

*Clocks* are real-valued variables. A set of clocks is  $X = \{x_1, \dots, x_H\}$ ; a clock valuation is  $w : X \rightarrow \mathbb{R}_{\geq 0}$ . By default, all clocks are evolving at the same rate (1), but this rate can be defined to any rational value in IMITATOR. Clocks can also be *stopped* in IMITATOR, *i.e.*, their rate can be zero.

*Parameters* are rational-valued variables, that act as unknown constants. A set of parameters is  $P = \{p_1, \dots, p_M\}$ ; a parameter valuation is a function  $v : P \rightarrow \mathbb{Q}$ . We will often identify a valuation  $v$  with the *point*  $(v(p_1), \dots, v(p_M))$ .

#### 3.1.2 Discrete rational variables

IMITATOR features *discrete* variables. Discrete variables<sup>1</sup> are global variables. Their value is global, in the sense that they are shared by all IPTA of the model. They can be seen as

---

<sup>1</sup>The name “discrete variable” comes from HyTECH.

syntactic sugar to represent a possibly unbounded number of locations.

We only describe here rational-valued variables. Rational-valued variables are the only ones to be compared *directly* to clocks or parameters in IMITATOR. Further types of discrete variables (*e.g.*, integers, Booleans, arrays...) will be described in [Section 3.5](#). These latter types cannot be directly compared to clocks or parameters.

In IMITATOR, rationals are exact and unbounded, just as in maths (*i.e.*, they are *not* represented using a limited number of bits, such as 32 or 64 bits). Rational variables are encoded in IMITATOR as *exact rational arithmetics*, thanks to the GNU Multiple Precision Arithmetic (GMP) library. Therefore, neither floating-point approximation, nor overflow, can occur; and the representation of the constraints is always exact. Note that floating-point numbers (“float”) are totally absent from the IMITATOR implementation (except for the generation of graphical outputs).

In the following, we define the set of rational variables as  $R = \{r_1, \dots, r_J\}$ ; a rational variable valuation is a function  $\rho : R \rightarrow \mathbb{Q}$ .

### 3.1.3 Linear constraints

Let us formalize the set of linear constraints allowed in IMITATOR. Given a set of variables  $Var = \{z_1, \dots, z_N\}$  (in the following, this set will be instantiated with  $X$  and/or  $P$  and/or  $R$ ), a *linear term* over  $Var$  is an expression of the form

$$\sum_{1 \leq i \leq n} \alpha_i z_i + d$$

for some  $n \in \mathbb{N}$ , where  $z_i \in Var$ ,  $\alpha_i \in \mathbb{Q}$ , for  $1 \leq i \leq n$ , and  $d \in \mathbb{Q}$ .

An *atomic constraint* over  $Var$  is an expression of the form  $lt \triangleright \triangleleft 0$  where  $lt$  is a linear term over  $Var$ , and  $\triangleright \triangleleft \in \{<, \leq, =, \geq, >\}$ .

A *constraint* over  $Var$  is a conjunction of atomic constraints. We denote by  $\mathcal{LT}(Var)$  the set of linear terms over  $Var$ , and by  $\mathcal{LC}(Var)$  the set of constraints over  $Var$ . In IMITATOR, we will consider constraints belonging to sets such as  $\mathcal{LC}(X \cup P)$  (*i.e.*, the set of constraints over clocks and parameters), or  $\mathcal{LC}(X \cup P \cup R)$  (*i.e.*, the set of constraints over clocks, parameters and discrete rational-valued variables).

### 3.1.4 Arithmetic expressions

Let  $\mathcal{AE}(R)$  denote the set of arithmetic expressions over the numerical discrete rational variables, *i.e.*, made of addition, subtraction, multiplication, and division over rational (or integer) constants and discrete numerical variables.

### 3.1.5 IMITATOR Parametric Timed Automata

We can now give a formal definition of IPTA.

*Remark 3.1.* In the subsequent formal definition, we use for discrete variables the sole set of rational-valued variables instead of other types (int-valued variables, Boolean variables and binary words, lists, arrays, etc.), for sake of simplicity. These more complex types can be used in a straightforward manner. That is, guards (and invariants)

can also include operations on such variables. See the section dedicated to discrete variables (Section 3.5) and the grammar in Chapter 9 for the exact behavior allowed by IMITATOR.

Let  $\epsilon$  denote the unobservable action.

**Definition 3.1** (IPTA). An IMITATOR parametric timed automaton (IPTA) is a tuple  $\mathcal{A} = \langle \Sigma, L, \ell_{\text{init}}, R, X, P, I, \text{flow}, \rightarrow \rangle$ , where:

- $\Sigma$  is a finite set of actions;
- $L$  is a finite set of locations;
- $\ell_{\text{init}} \in L$  is the initial location;
- $R$  is a set of rational-valued variables;
- $X$  is a set of clocks;
- $P$  is a set of parameters;
- $I : L \rightarrow \mathcal{LC}(X \cup P \cup R)$  assigns to every location  $\ell$  a constraint over all variables, called the *invariant* of  $\ell$ ;
- $\text{flow} : L \times X \rightarrow \mathbb{Q}$  assigns to a every location and clock a given flow (or speed), *i.e.*, a rational number;
- $\rightarrow$  is a set of edges  $(\ell, g, a, X_{\text{up}}, D_{\text{up}}, \ell')$ , where  $\ell, \ell' \in L$  are the source and destination locations,  $g \in \mathcal{LC}(X \cup P \cup R)$  is a constraint over all variables (called *guard* of the transition),  $a \in \Sigma \cup \{\epsilon\}$  is the action associated with the transition,  $X_{\text{up}} : X \rightarrow \mathcal{LT}(X \cup P \cup R)$  is the (possibly partial) update function for clocks, and  $D_{\text{up}} : R \rightarrow \mathcal{AE}(R)$  is the (possibly partial) update function for discrete rational variables.

In the following, we explain this definition.

**Guards and invariants** Guards and invariants in IMITATOR are linear constraints over all variables. For example, the following expression can be used in a guard or an invariant:

```
r1 + .5 x1 + 3 x2 >= 2 p1 - r2 & p2 < 1/3
```

where **r1**, **r2** are discrete rational variables, **x1**, **x2** are clocks and **p1**, **p2** are parameters. This syntax includes in particular diagonal constraints (*e.g.*, **x1** - **x2** <= 2), not always supported in other model-checking tools.

**Actions** Transitions can be synchronized on an action in  $\Sigma$ , or have no synchronized action (“ $\epsilon$ ”), which is often referred to in the literature as a silent transition, or an  $\epsilon$ -transition. For the semantics of the synchronization model between various IPTA, refer to Section 3.3. A non-silent transition is said to be *observable*; and it can be synchronized with other automata.



**Clock updates** Observe that clocks can be updated to any value, *i.e.*, a clock can be assigned not only to 0, but to any linear term over the other clocks, the parameters and the discrete rational variables. This considerably extends the traditional syntax of PTAs defined in [AHV93]. In fact, the IMITATOR includes (more than just) the updatable timed automata of [Bou+04], as well as the reset-to-parameter (parametric) timed automata of [ALR18b]. If clocks are always reset to constants (*i.e.*, not assigned to more complex linear terms), IMITATOR will apply some optimizations that (may) increase the analysis speed.

**Discrete updates** Discrete variables can be assigned to expressions compatible with their type. Notably, rational variables can be assigned to rational arithmetic expressions over  $R$ . On the one hand, this is more restrictive than clock updates, because discrete rational variables cannot be assigned to a clock or to a parameter (in contrast to clocks, that *can* be assigned to linear expressions over clocks, parameters and discrete rational variables). On the other hand, arithmetic expressions are richer than linear constraints, as they notably allow multiplication or division of rational variables with each other.

Since IMITATOR 2.12, `if-then-else` conditions are allowed in *discrete* updates. In addition, while by default updates are *not* sequential in IMITATOR, we also allow sequential updates since IMITATOR 3.3. See [Section 3.5.3](#) for details on updates.

Finally note that, by definition, parameters (that are unknown timing *constants*) cannot be updated.

**Flows and stopwatches** By default, all clocks evolve at the same rate in all locations, and this rate is 1. That is, by default,  $\forall \ell \in L, \forall x \in X : \text{flow}(\ell, x) = 1$ .

However, one can explicitly define an alternative (constant, rational-valued) flow using two methods:

**Using stopwatches** A stopwatch [CL00] is a clock whose elapsing is stopped in some locations, *i.e.*,  $\text{flow}(\ell, x) = 0$ . There is no distinction between clocks and stopwatches. That is, any clock can potentially be stopped in some location. IMITATOR will detect whether a model has or not stopwatches; if there is no stopwatch in the model, IMITATOR will apply some optimizations that (may) increase the analysis speed.

Clocks can be stopped in locations, thanks to the optional `stop { ... }` keyword (see grammar in [Section 9.2](#)). A clock is stopped in a location if it belongs to the list of clocks in `stop { ... }`. It is resumed when leaving the location—unless the target location again stops this clock.

**Using explicit flows** IMITATOR also support constant flows, *i.e.*, multi-rate (parametric) timed automata [Alu+95]. That is, it is possible to specify the *speed* of a clock in a location. This value is an arbitrary rational number, including 0 (in which case it is equivalent to a stopwatch [CL00]) or a negative number. This can be specified using a syntax of the following form: `flow {x' = 2}`. A clock for which no flow is explicitly defined has of course flow 1. A stopwatch has flow 0.

**Remark 3.2.** It is possible to defined ill-formed models where contradictory flows are defined for a given clock. For example, in

```
stop { x } flow {x' = 2, y' = 0}
```

$x$  is assigned both flow 0 (because it is defined as a stopped clock) and 2 (in the explicit flow definition).

Alternatively, in an NIPTA, it is possible to be in a global location such that, in one location of one of the automata a given clock is assigned a given flow, and in another location of another automata in parallel, the same clock is assigned another flow.

In the case of such contradictory flows, IMITATOR will trigger a warning on-the-fly, and the result of the analysis is unspecified.

**Urgent locations** A location can be defined as *urgent*, using the keyword `urgent` (see grammar in [Section 9.2](#)). In an urgent location, time cannot elapse, *i.e.*, it must be left in 0-time. Note however that “in 0-time” does not necessarily mean immediately: that is, several actions may be performed (in 0-time) before the location is left.

### 3.1.6 Networks of IMITATOR Parametric Timed Automata

**Definition 3.2** (NIPTA). Given a set of IPTA  $\mathcal{A}_i = \langle \Sigma_i, L_i, (\ell_{\text{init}})_i, R_i, X_i, P_i, I_i, \text{flow}_i, \rightarrow_i \rangle$ ,  $1 \leq i \leq N$  for some  $N \in \mathbb{N}$ , a network of IPTA (NIPTA) is a tuple  $\langle \Sigma, R, X, P, \{\mathcal{A}_i \mid 1 \leq i \leq N\}, C_{\text{init}} \rangle$ , where:

- $\Sigma = \bigcup_{1 \leq i \leq N} \Sigma_i$  is the set of all actions;
- $R = \bigcup_{1 \leq i \leq N} R_i$  is the set of all discrete rational variables;
- $X = \bigcup_{1 \leq i \leq N} X_i$  is the set of all clocks;
- $P = \bigcup_{1 \leq i \leq N} P_i$  is the set of all parameters;
- $C_{\text{init}} \in \mathcal{LC}(X \cup P \cup R)$  is the initial constraint over  $R$ ,  $X$  and  $P$ .

Observe that each set of actions, discrete variables, clocks and parameters is not disjoint between all IPTA. That is, actions, discrete variables, clocks and parameters may be shared between different IPTA. If a variable is required to be local to an IPTA, then it should just not be used in any other IPTA of the model.

Different from many tools for (parametric) timed automata, clocks are not necessarily initially equal to 0 (this is similar to HyTech [HHW95] but different from UPPAAL [LPY97]). The initial value of the clocks is defined by  $C_{\text{init}}$  (see [Section 3.2](#)). If nothing is defined in  $C_{\text{init}}$ , then their value is supposed to be arbitrary (any real value).

In an NIPTA, a clock  $x$  is stopped in a location  $(\ell_1, \dots, \ell_N)$  if it stopped in at least one of the locations  $\ell_i$ , *i.e.*,  $x \in \text{flow}_i(\ell_i) = 0$  for some  $i \in \{1, \dots, N\}$ .

Note that parameters are not assumed to be positive; however, the behavior of IMITATOR has not been tested for negative parameters, and it is strongly advised to constrain them to be non-negative in  $C_{\text{init}}$  (if it is not the case, a warning is issued by IMITATOR).

Finally, note that the number of IPTA, locations, variables and actions that can be defined in a model is bounded in IMITATOR by some very large number (most probably  $2^{32}$ ); but, well, you don't seriously plan to build such a large model, do you?

**Determinism** IMITATOR may use the determinism in (yet to come) algorithms; and the deterministic nature of an NIPTA is detected by IMITATOR. In our setting, the strong determinism denotes at most one transition going out from a location and labeled with a given observable (*i.e.*, non-silent) transition. Let us precisely define what it means:

**Definition 3.3** (strong determinism). Let  $\mathcal{A}$  be an NIPTA made of  $N$  IPTA  $\mathcal{A}_i = \langle \Sigma_i, L_i, (\ell_{\text{init}})_i, R_i, X_i, P_i, I_i, \text{flow}_i, \rightarrow_i \rangle$ ,  $1 \leq i \leq N$  for some  $N \in \mathbb{N}$ .

$\mathcal{A}$  is *strongly-deterministic* if for all automaton  $\mathcal{A}_i$ , for all location  $\ell_i \in L_i$ , for all *observable* action  $a \in \Sigma_i$ , there exists at most one transition from  $\ell_i$  labeled with  $a$ .

Note that an existing weaker notion of determinism allows multiple transitions from the same  $\ell_i$  labeled with the same action  $a$ , as long as guards are mutually exclusive. This is not the approach we choose in IMITATOR, where we impose at most one transition labeled with the same action.

Once more, the notion of strong determinism in IMITATOR is *not* used in any algorithm, but its nature is “simply” detected (and part of the result file).

## 3.2 Initial state and initialization of variables

For each IPTA, a unique initial location must be defined.

For variables, the definition of the initial value is very permissive in IMITATOR. Clocks are not necessarily equal to 0, and parameters are not even necessarily positive.

Parameters and clocks can be initially bound by any linear constraint over parameters, clocks, and discrete variables. That is, we can define initial constraints such as:

```
x1 + x2 <= 2 p1 + 0.5 p2 - i
```

However, discrete variables must be initialized to a *constant* compatible with their type. See [Section 3.5.2](#) for details. Given a discrete variable `i`, if the definition of the initial state does not contain an equality of the form `i := ...` followed by a constant (or a linear term over  $\mathcal{LT}(R)$ , for rational variables), then IMITATOR will assume that `i` is initially set to a default value, and will issue a warning.

## 3.3 Synchronization model

By default, all IPTA of an IMITATOR model declare their set of actions.<sup>2</sup>

The IMITATOR synchronization model is such that *all* IPTA declaring an action must synchronize *together* on this action. This can be seen as a *strong broadcast*. That is, for a

<sup>2</sup>An alternative is an automatic recognition of the actions used, see option `-sync-auto-detect` in [Chapter 8](#).

transition labeled with action `act` to be executed, all IPTA declaring `act` must be ready to execute `act` locally. Otherwise, this transition cannot be taken (yet).

If an IPTA declares an action `act` that is never used in this IPTA, then action `act` will never be executed in the entire model.<sup>3</sup>

Also note that, when synchronizing transitions, the order in which the associated updates are executed is described in [Section 3.5.3](#).

## 3.4 Global constants

IMITATOR supports global constants, *i.e.*, a variable the value of which is known once for all. The syntax is the following one:

```
c = 1: constant;
```

Then, any occurrence of `c` in the model is replaced with `1`.

By default, constants are (unbounded, exact) rationals. It is possible to define constants of another type by making their type explicit:

```
c = 1: int;
```

In this case, `c` is a constant of type `int`.

Limited linear expressions over rationals can be used in the definition of a (rational) constant; however no other constant can be used in a definition, *i.e.*, one cannot (yet) write `c1 = c2 + 1: constant;`.

*Hint 3.1.* In fact, a variable (*e.g.*, a parameter) can be turned to a constant as follows in the definition of the parameters:

```
p = 2: parameter;
```

This is equivalent to replacing `p` with `2` everywhere in the model; this is particularly useful when some parameters should be valuated. In contrast, if the parameter is valuated in the initial state definition, IMITATOR still counts it as a parameter, which makes all constraints suffer from an additional dimension.

## 3.5 Discrete variables

### 3.5.1 Types

Discrete, global variables are syntactic sugar for a potentially unbounded number of locations. Discrete variables can be tested in guards, and updated along transitions.

IMITATOR currently supports four primitive types of discrete variables:

1. rational-valued variables
2. “int”-valued variables (over 32 bits)

<sup>3</sup>In this case, IMITATOR will detect this situation and will entirely delete this action from the model, while issuing a warning.

3. Boolean-valued variables
4. binary word variables, allowing to use bitwise binary operations.

In addition, more complex types such as *arrays*, *lists*, *stacks* and *queues* (of primitive types, or of composite types such as arrays of lists) can be used.

We review in the following these various types, together with their associated built-in functions.

### 3.5.1.1 Rational-valued variables

*Rational variables* are (exact, unbounded) rational-valued variables, and were described in [Section 3.1.2](#).

### 3.5.1.2 “Int”-valued variables

*Int variables* are bounded signed integer-valued variables encoded using 32 bits (which is architecture independent).

Int variables are encoded in IMITATOR using the `int` type of OCaml.

**Warning 1** (int encoding). All arithmetic operations over the “int” variables are taken modulo  $2^{32}$ . Overflow can therefore occur.

For critical models, it is highly advised to use *rational* variables instead (the computation using rational variables may be significantly slower, but is always exact and without overflow).

Formally, the set of integer variables is  $I = \{i_1, \dots, i_K\}$ ; an integer variable valuation is a function  $v: I \rightarrow \mathbb{Z} \cap [-2, 147, 483, 648, +2, 147, 483, 647]$ .

### Arithmetic functions

- `pow(x:'a number, e:int):'a` with `'a number:[rational|int]` - compute the power of a (rational or integer) number

**Example 3.1.** To illustrate this operation, note that the following guard in the model fragment below will evaluate to true:

```
when pow(2, 3) = 8 goto l1;
```

1  
2

### Conversion functions

- `rational_of_int(x:int):rational` - convert an int number to a rational one

**Example 3.2.** To illustrate this operation, note that the following guard in the model fragment below will evaluate to true:

```
(* i : int *)
(* r : rational *)
when i = 2 & r = 2 & rational_of_int(i) = r goto l1;
```

1  
2  
3  
4

*Remark 3.3.* Comparing natively (using a “cast”) an integer and a rational is not allowed. That is, the following code is incorrect:

```
when i = 2 & r = 2 & i = r goto l1; (* incorrect! *)
```

1  
2

### 3.5.1.3 Boolean variables

IMITATOR also features Boolean-valued variables. The set of Boolean variables is  $B = \{b_1, \dots, b_L\}$ ; a Boolean variable valuation is a function  $\beta : B \rightarrow \mathbb{B}$ , where  $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ .

**Example 3.3.** To illustrate Boolean variables, note that the following guards in the dummy model fragment below will all evaluate to true:

```
(* Assume true_bool = True, another_true_bool = True *)
(* Assume false_bool = False *)
(* Assume i = 1 *)

when True goto l1;
when true_bool goto l1;
when not(false_bool) goto l1;
when (true_bool | i = 0) goto l1;
when true_bool = another_true_bool & not(i = 5) goto l1;
when true_bool <> false_bool goto l1;
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13

### 3.5.1.4 Binary word variables

Binary word variables are sequences of bits of fixed length—the maximum size of a binary word is  $2^{16}$ . Bitwise operations can be defined on these variables (on same length).

For example, defining two binary words `bw1` and `bw2` both of size 4 (type : `binary(4)`), such that initially `bw1 := 0b1010` and `bw2 := 0b1011`, we can use a guard `logor(bw1, bw2) <> bw1` that checks whether the result of the bitwise (logical) “or” operation on both variables differs from `bw1`. Then, if this check is true, one may want to shift the bits of `b1` by 2 positions, and set `bw2` to the result of the bitwise “and” operation on `bw1` and `bw2`. A

transition featuring such a guard and such updates would be written as follows in IMITATOR syntax:

```
(* before: bw1 = 0b1010 and bw2 = 0b1011 *)
when logor(bw1, bw2) <> bw1 sync a do {bw1 := shift_left(bw1, 2), bw2 :=
  logand(bw1, bw2)} goto l1;
(* after: bw1 = 0b1000 and bw2 = 0b1010 *)
```

Note that binary words are of fixed length, and operations can only be applied to words of the *same* length.

We denote by  $W$  the set of binary word variables.

### Binary word functions

- `shift_left(b:binary_word(l), i:int):binary_word(l)` - shift a binary word of length  $l$  to left
- `shift_right(b:binary_word(l), i:int):binary_word(l)` - shift a binary word of length  $l$  to right
- `fill_left(b:binary_word(l), i:const int):binary_word(l + i)` - shift a binary word of length  $l$  to left (non destructive)
- `fill_right(b:binary_word(l), i:const int):binary_word(l + i)` - shift a binary word of length  $l$  to right (non destructive)
- `logand(b1:binary_word(l), b2:binary_word(l)):binary_word(l)` - apply AND bitwise operation to two binary words of length  $l$
- `logor(b1:binary_word(l), b2:binary_word(l)):binary_word(l)` - apply OR bitwise operation to two binary words of length  $l$
- `logxor(b1:binary_word(l), b2:binary_word(l)):binary_word(l)` - apply XOR bitwise operation to two binary words of length  $l$
- `lognot(b:binary_word(l)):binary_word(l)` - apply NOT bitwise operation to a binary word of length  $l$

**Example 3.4.** To illustrate the binary words operations, note that all the following guards in the model fragment below will evaluate to true:

```
when shift_left(0b010110 , 3)      = 0b110000      goto l1;
when shift_right(0b010110 , 3)     = 0b000010      goto l1;
when fill_left(0b010110 , 3)       = 0b010110000   goto l1;
when fill_right(0b010110 , 3)      = 0b000010110   goto l1;
when logand(0b010110 , 0b101010 ) = 0b000010      goto l1;
when logor(0b010110 , 0b101010 ) = 0b111110      goto l1;
when logxor(0b010110 , 0b101010 ) = 0b111100      goto l1;
when lognot(0b010110 )             = 0b101001      goto l1;
```

### 3.5.1.5 Arrays

Array variables are fixed-length collections of indexed elements. Array elements can be accessed or updated using an integer-valued arithmetic expression as index. An array variable always has a fixed length and an underlying type, which can be any of the available types of IMITATOR (including arrays, lists, stacks and queues). For example, it is possible to declare an array of integers, or an array of Booleans, or an array of arrays of arrays of Booleans—which can be seen as a multidimensional array.

Examples of array definitions are given below:

```
my_const_rational_array = [1, 2, 3] : rational array (3)
my_rational_array       : rational array (5);
my_int_array            : int array (2);
my_bool_array          : bool array (3);
my_binary_word_array    : binary (4) array (2);
my_2d_int_array         : int array (2) array (2);
```

### Array functions

- `a[i:int]:'a` with `'a:any` - access to element of array `a` at index `i`
- `array_append(a1:'a array(l1), a2:'a array(l2)):'a array(l1+l2)` with `'a:any` - concatenates an array of length `l1` with an array of length `l2`, resulting in an array of length `l1 + l2`
- `array_length(a1:'a array(l1)):int=l1` with `'a:any` - length of the array `a1`
- `array_mem(a1:'a, l:'a array(l1)):bool` with `'a:any` - true if `e` is in the array `a1`

**Example 3.5.** To illustrate the arrays operations, note that the following guard in the model fragment below will evaluate to true:

```
when
  & my_const_rational_array[0] = 1
  & array_append([1, 2, 3], [4, 5, 6]) = [1, 2, 3, 4, 5, 6]
  & array_append([[1, 2]], [[3, 4]]) = [[1, 2], [3, 4]]
  & array_length([20, 4, 6]) = 3
  & array_mem(2, [8, 6, 2])
goto l1;
```

### 3.5.1.6 Lists

Lists are collections of variable length. Contrarily to arrays, the type of a list does not include its length; it is therefore possible to compare two lists of different lengths. List elements can only be accessed—not updated—using an integer-valued arithmetic expression as index. A list variable always has an underlying type, which can be any of the available



types of IMITATOR (including arrays, lists, stacks and queues). For example, it is possible to declare a list of integers, or a list of Booleans, or a list of lists of arrays of Booleans.

Examples of list definitions are given below:

```
my_const_rational_list = list([1, 2, 3]) : rational list;
my_rational_list       : rational list;
my_int_list            : int list;
my_bool_list           : bool list;
my_binary_word_list    : binary(4) list;
my_2d_int_list         : int list list;
```

1  
2  
3  
4  
5  
6

### List functions

- `l[i:int]:'a` with `'a:any` - access to element of list `l` at index `i`
- `list_cons(e:'a, l:'a list):'a list` with `'a:any` - construct a list by inserting a new element `e` at the beginning of `l`
- `list_hd(l:'a list):'a` with `'a:any` - head element of a list `l`
- `list_is_empty(l:'a list):bool` with `'a:any` - Check if the list `l` is empty
- `list_length(l:'a list):int` with `'a:any` - length of the list `l`
- `list_mem(e:'a, l:'a list):bool` with `'a:any` - true if `e` belongs to the list `l`
- `list_rev(l:'a list):'a list` with `'a:any` - reverse the list `l`
- `list_tl(l:'a list):'a list` with `'a:any` - tail of the list `l`

**Example 3.6.** To illustrate the lists operations, note that the following guard in the model fragment below will evaluate to true:

```
when
  & my_int_list[0] = 1
  & list_cons(1, list([2, 3])) = list([1, 2, 3])
  & list_hd(list([3,4])) = 3
  & list_is_empty(list([]))
  & list_length(list([1, 1, 1])) = 3
  & list_mem(2, list([8, 6, 2]))
  & list_rev(list([1, 2, 3])) = list([3, 2, 1])
  & list_tl(list([1, 2, 3])) = list([2, 3])
goto l1;
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11

### 3.5.1.7 Stacks

Stacks are LIFO collections (Last In First Out) of variable length. It is possible to compare two stacks of different lengths. Stack elements can only be accessed—not updated—using pop or top functions. A stack variable always has an underlying type, which can be any of the available types of IMITATOR (including arrays, lists, stacks and queues). For example, it is possible to declare a stack of integers, or a stack of Booleans, or a stack of arrays of lists of binary words of length 5.

*Remark 3.4.* A literal stack value can only be an empty stack.

Examples of stack definitions are given below:

```

my_const_rational_stack = stack() : rational stack; (* useless but
possible *)
my_rational_stack      : rational stack;
my_int_stack           : int stack;
my_bool_stack          : bool stack;
my_binary_word_stack   : binary(4) stack;
my_int_list_stack      : int list stack;

```

### Stack functions

- `stack_push(e:'a, s:'a stack)` with `'a:any` - Add an element `e` at the top of the stack `s` and return the stack—the stack is modified, this function is subject to side-effects
- `stack_pop(s:'a stack)':'a` with `'a:any` - Get and remove the element at the top of the stack `s`—the stack is modified, this function is subject to side-effects
- `stack_top(s:'a stack)':'a` with `'a:any` - Get the element at the top of stack `s`—the stack is left unchanged
- `stack_clear(s:'a stack)` with `'a:any` - Remove all elements of the stack `s` and return the stack—the stack is modified, this function is subject to side-effects
- `stack_is_empty(s:'a stack):bool` with `'a:any` - Check if the stack `s` is empty
- `stack_length(s:'a stack):int` with `'a:any` - Get the number of elements in the stack `s`

**Example 3.7.** The following code fragment illustrates some possibilities offered by the stacks operations:

```

loc 10: invariant True
when
  & True
do {
  seq
    stack_push(1, s);

```

```

        stack_push(2, s);
        i := stack_top(s);
        j := stack_pop(s);
    }
    goto l1;

loc l1: invariant
    i = j
    & i = 2
    & stack_length(s) = 1
    & not(stack_is_empty(s))
when
    & True
do {
    seq
        stack_clear(s);
}
goto l2;

loc l2: invariant stack_is_empty(s)

```

The `seq` keyword denotes a sequential update (see [Section 3.5.3.3](#)). Side-effect functions can only be used in such sequential updates.

### 3.5.1.8 Queues

Queues are FIFO collections (First In First Out) of variable length. It is possible to compare two queues of different lengths. Queue elements can only be accessed—not updated—using `pop` or `top` functions. A queue variable always has an underlying type, which can be any of the available types of IMITATOR (including arrays, lists, stacks and queues). For example, it is possible to declare a queue of integers, a queue of Booleans, or a queue of stacks of binary words of length 3.

*Remark 3.5.* A literal queue value can only be an empty queue.

Examples of queue definitions are given below:

```

my_const_rational_queue = queue() : rational queue; (* useless but
    possible *)
my_rational_queue       : rational queue;
my_int_queue            : int queue;
my_bool_queue           : bool queue;
my_binary_word_queue    : binary(4) queue;
my_int_list_queue       : int list queue;

```

### Queue functions

- `queue_push(e:'a, q:'a queue)` with `'a: any` - Add an element `e` at the beginning of the queue `q` and return the queue—the queue is modified, this function is subject to side-effects

- `queue_pop(q: 'a queue): 'a` with `'a: any` - Get and remove the last element of the queue `q`—the queue is modified, this function is subject to side-effects
- `queue_top(q: 'a queue): 'a` with `'a: any` - Get the last element of the queue `q`—the queue is left unmodified
- `queue_clear(q: 'a queue)` with `'a: any` - Remove all elements of the queue `q` and return the queue—the queue is modified, this function is subject to side-effects
- `queue_is_empty(q: 'a queue): bool` with `'a: any` - Check if the queue `q` is empty
- `queue_length(q: 'a queue): int` with `'a: any` - Get the number of elements in the queue `q`

**Example 3.8.** The following code fragment illustrates some possibilities offered by the queues operations:

```

loc 10: invariant True
when
  & True
do {
  seq
    queue_push(1, q);
    queue_push(2, q);
    i := queue_top(q);
    j := queue_pop(q);
}
goto 11;

loc 11: invariant
  i = j
  & i = 1
  & queue_length(q) = 1
  & not(queue_is_empty(q))
when
  & True
do {
  seq
    queue_clear(q);
}
goto 12;

loc 12: invariant queue_is_empty(q)

```

### 3.5.1.9 Built-in functions over discrete variables

We summarize in [Table 3.1](#) the built-in functions for discrete variables.

Table 3.1: Built-in functions summary

Name	Description
<b>Arithmetic</b>	
<code>pow</code>	Pow a number
<code>rational_of_int</code>	Convert an int expression to a rational one
<b>Binary words</b>	
<code>shift_left</code>	Shift a binary to the left (truncate to the binary word length)
<code>shift_right</code>	Shift a binary to the right (truncate to the binary word length)
<code>fill_left</code>	Shift a binary to the left (no truncate)
<code>fill_right</code>	Shift a binary to the right (no truncate)
<code>logand</code>	And bitwise on binary numbers
<code>logor</code>	Or bitwise on binary numbers
<code>logxor</code>	Xor bitwise on binary numbers
<code>lognot</code>	Not bitwise on a binary number
<b>Arrays</b>	
<code>array_append</code>	Concatenate two arrays
<code>array_length</code>	Get length of an array
<code>array_mem</code>	Search existence of an element in an array
<b>Lists</b>	
<code>list_cons</code>	Construct a list by adding an element to the beginning
<code>list_hd</code>	Get head of a list
<code>list_is_empty</code>	Check if a list has elements
<code>list_length</code>	Get length of a list
<code>list_rev</code>	Reverse a list
<code>list_mem</code>	Search existence of an element in a list
<code>list_tl</code>	Get tail of a list
<b>Stack</b>	
<code>stack_push</code>	Push an element on the top of a stack - the stack is modified
<code>stack_pop</code>	Get and remove the element on the top of a stack - the stack is modified
<code>stack_top</code>	Get the element on the top of a stack without modifying it
<code>stack_clear</code>	Remove all elements of a stack - the stack is modified
<code>stack_is_empty</code>	Check if the stack has elements
<code>stack_length</code>	Get the number of elements of a stack
<b>Queues</b>	
<code>queue_push</code>	Add an element at the beginning of a queue - the queue is modified
<code>queue_pop</code>	Get and remove the last element of a queue - the queue is modified
<code>queue_top</code>	Get the last element of a queue without modifying it
<code>queue_clear</code>	Remove all elements of a queue - the queue is modified
<code>queue_is_empty</code>	Check if the queue has elements
<code>queue_length</code>	Get the number of elements of a queue

*Remark 3.6.* The parameters of a function are evaluated from left to right.

*Remark 3.7.* Some functions, called “Side-effects functions” can modify the value of some input variables and therefore are subject to side-effects. Side-effects functions

can only be used in updates, more precisely in the body of the `seq` block of an update (see [Section 3.5.3](#)).

### 3.5.2 Default initial value

Discrete variables must be initialized to a single constant value in the `init` definition; if they are not, a warning is issued, and they are arbitrarily set to a standard value.

This initial default value of primitive types is as follows, depending on the variable type:

Rational	0
Integer	0
Boolean	<b>false</b>
Binary words	0b0

The initial default value of composite types is as follows:

Array	<code>[]</code>
List	<code>list([])</code>
Stack	<code>stack([])</code>
Queue	<code>queue([])</code>

### 3.5.3 Updates

The presence of clocks, parameters, discrete variables, and structures (such as arrays) requires a particular handling of the updates, which we describe in the following.

In any case, the paradigm is that discrete variables are first tested in guards, then updated in updates.

#### 3.5.3.1 Conflicts in updates

Conflicts can arise in standard updates due to the modular nature of NIPTA. If two IPTA in parallel update the same variable on the same synchronized action `a` (e.g., an IPTA performs `i := 2` on a transition labeled with `a`, while another one performs `i := 3` on a transition labeled with `a`), then a warning is issued, and the behavior of the NIPTA becomes unspecified (i.e., IMITATOR will choose one or the other assignment in an unspecified manner).

#### 3.5.3.2 Standard updates

By default, updates are not sequential in IMITATOR, but rather define a partial function on variables (including clocks). That is, in the following block:

```
when x = 0 & i = 1 do {i := i+1, i := 5, x := i}
```

after the update, `x` is equal to 0 (because `x := i` implies that the new value of `x` becomes the value of `i` *before* the update), while the value of `i` is unspecified due to a non-deterministic assignment, i.e., `i` is updated to two different values in the update (this is an ill-formed behavior, and a warning will be triggered).

Since IMITATOR 3.3, we also allow for some sequential updates on the discrete variables, as explained in the following.

### 3.5.3.3 seq-then updates

**seq-then** updates are more complex updates, allowing first a set of sequential updates (on the discrete variables only, including side-effects functions), called the “**seq** block”, followed by “standard updates” (non-sequential), called the **then** block.

**The seq block** A **seq** block can be used to define sequential updates on any discrete variable of any type. Contrary to the **then** block (or to the standard updates), it is possible to make modifications of data structures (such as stacks and queues) using side-effects functions. The **seq** block can be seen as an imperative update part that is processed before the **then** block. It is not possible to update clocks in a **seq** block.

In a model that contains multiple synchronized IPTA, given a transition, all **seq** blocks (all sequential updates) in all IPTA are processed as follows: the updates are performed sequentially starting from the top-most IPTA in the input model file (*i.e.*, following the definition order of IPTA). That is, all the sequential updates of the first of the IPTA are processed, followed by all the sequential updates of the second IPTA, and so on.

Examples of a **seq** update expression:

```
do {
    (* only do expression *)
    seq
    i := 1; (* after this update: i = 1 *)
    i := i + 1; (* after this update: i = 2 *)
end
}
```

```
do {
    (* only do expression *)
    seq
    (* modification of stack s (with side-effet) *)
    stack_push(1, s);
    (* modification of stack s (with side-effet) *)
    stack_push(2, s);
end
}
```

**The then block** The **then** block can be used to update clocks or discrete variables. These updates are not sequential so, as described before, in case of multiple updates of the same variable, a warning is issued, and the behavior of the NIPTA becomes unspecified. The **then** block is essentially the same as a standard update (see [Section 3.5.3.2](#)). This kind of update was the only one in IMITATOR until IMITATOR 3.3.

Below is an example of a **then** update expression:

```
(* Assume initially i = 1, j = 2, x = 2046 *)

do {
```

Table 3.2: Blocks features summary

Block	Sequential behavior	Clock update	Discrete update	Side effects	Variable auto remove
<code>seq</code>	✓	✗	✓	✓	✗
<code>then</code>	✗	✓	✓	✗	✓

```

(* only "then" expression *)
i := j + 3,
j := i * 3,
x := i - j
}
(* Now: i=5, j=3, x=-1 *)

```

4  
5  
6  
7  
8  
9

Below is an example of a `then` update expression, that is ill-formed and will trigger a warning:

```

(* Assume initially i = 1 *)

do {
  (* only "then" expression *)
  i := 3,
  i := 4,
  x := i
}
(* Now, x = 1, while i = 3 OR i = 4 (unspecified!) *)

```

1  
2  
3  
4  
5  
6  
7  
8  
9

**The `seq-then` block** In one given transition, the `seq` block is executed *before* the `then` block.

Recall that, in a model that contains multiple synchronized IPTA, given a transition, all `seq` blocks are performed sequentially starting from the top-most IPTA in the input model file. Only once all `seq` blocks are computed (sequentially), all `then` blocks are then executed (at once). Keep in mind that, contrarily to the `seq` blocks, updates contained in `then` blocks are *not* computed sequentially.

The value of the variables used in the right-hand part of an update in the `then` block is the value computed *after* executing the `seq` block updates.

*Remark 3.8.* Note the use of semicolon in the sequential block (`seq` block), in contrast to the comma in the declarative block (`then` block).

Table 3.2 summarizes what is allowed in the `seq` and `then` blocks. “Side effects” refers to side effect functions, while “variable auto remove” denotes the fact that any variable used in the `seq` block is considered useful and will not be automatically removed (see option `-no-var-autoremove` in Chapter 8 for more explanations on IMITATOR automatic variable deletion).

Examples of a `seq-then` update expression:

```

do {
  seq

```

1  
2  
3



```

    i := 0; (* i = 0 *)
    i := i + 1; (* now, i = 1 *)
  then
    i := 3, j := i (* now, i = 3 and j = 1 *)
  end
}

```

```

do {
  seq
    (* modification of stack s using side-effects *)
    stack_push(1, s);
    (* modification of stack s using side-effects *)
    stack_push(2, s);
    (* set i to 0 *)
    i := 0;
    (* i = 1, below *)
    i := i + 1;
    (* below, updating a clock is forbidden in a do block *)
    (* x := 0 *)
  then
    x := i,
    j := stack_top(s),
    (* below, r is update several time *)
    (* as these constraints updates are not sequential *)
    (* the value of r will be unspecified *)
    r := 0,
    r := 1
    (* using side-effects functions is forbidden in a then block *)
    (* r := stack_pop(s) *)
  end
}

```

Examples of a **seq-then** update expression with multiple synchronized transitions:

```

(*****)
automaton ptal
(*****)
synclabs : a;

loc l1: invariant x <= 0
when True
do {
  seq
    stack_push(0, s);
    i := 1;
  then
    (* r1 = 1 and x = 2, because all seq updates in all IPTA are made
    before *)
    r1 := stack_top(s),
    x := i
  end
}
sync a goto lend;

```

```

accepting loc lend: invariant True
end (* pta *)
(*****)

(*****)
automaton pta2
(*****)
synclabs : a;
loc l1: invariant True
when
& True
do {
  seq
    i := 2;
    stack_push(1, s);
  then
    (* r2 = 1, because all seq updates in all IPTA are made before *)
    r2 := stack_top(s)
  end
}
sync a goto lend;

accepting loc lend: invariant True
end (* pta *)
(*****)

(*****)
automaton pta3
(*****)
synclabs : a;
loc l1: invariant True
when
& True
do {
  (* r3 = 1, because all seq updates in all IPTA are made before *)
  r3 := stack_top(s)
}
sync a goto lend;

accepting loc lend: invariant True
end (* pta *)

```

### 3.5.4 Runtime errors

Runtime errors can be encountered during updates. For a example, during an update, a division by zero can be encountered: in an update  $i := 2 / i$ , when the current value of  $i$  is 0, the result becomes undefined.

When a runtime error is encountered, an exception is raised and IMITATOR terminates with an error.

The possible runtime errors include:

- division by 0;



- non-integer division on integer expression;
- index out of range (*e.g.*, in an array);
- operation on an empty collection (*e.g.*, accessing the top element of an empty stack).

## Chapter 4

# Parameter synthesis using IMITATOR

We give here the commands corresponding to the main analysis features of IMITATOR. We only give the most useful options. For more detailed commands, and a complete list of options, see [Chapter 8](#).

### 4.1 Synthesis and emptiness

**Main command** The standard IMITATOR command for all algorithms is:

```
imitator model.imi property.imiprop
```

**Synthesis and emptiness** IMITATOR features two modes for properties:

- synthesis (`#synth`): synthesizing all valuations such that a property holds;
- witness (`#exhibit` or `#witness`, both equivalent): find (at least) one valuation such that a property holds.

While the “witness” mode is close to (the contrary of) emptiness-checking, the “witness” mode is not strictly speaking an emptiness check: rather, the algorithm stops as soon as it finds some valuations, but there may be more than one, so the result can still be symbolic. For example, in the case of reachability-checking, IMITATOR would output the parameter valuations associated with the first target symbolic state found.

While all properties implement synthesis, not all of them implement emptiness; a warning would then be raised.

In the following, we now describe the algorithms implemented in IMITATOR.

### 4.2 Reachability

A main problem in parametric timed automata is to compute the set of parameter valuations for which some location (for instance, an error location) is reachable.

The property syntax is as follows:<sup>1</sup>

```
property := #synth EF(state_predicate);
```

where `state_predicate` is a state predicate, for example of the form `loc[AUTOMATON] = LOCATION`, where `AUTOMATON` is an automaton name, and `LOCATION` is a location name. State predicates allow conjunction, disjunction and the use of global discrete variables (see [Section 9.3](#) for details).

The algorithm EFsynth implemented in IMITATOR is a basic breadth-first procedure, close to the one described in [JLR15]. Of course, the EF-emptiness problem being undecidable [AHV93], the analysis is not guaranteed to terminate.

One obtains a result in an external text file (`model.res`) formatted using a standardized manner (see [Chapter 5](#)), and therefore easier to parse using an external tool than the terminal output.

The options `-merge` [AFS13] and `-comparison inclusion` are implicitly activated for this algorithm, as they usually greatly increase the analysis efficiency and the termination. You may want to use `-merge none` and `-comparison none` to disable them. The options `-dynamic-elimination` (not activated by default) and `-comparison doubleinclusion` can also be used to (try to) reduce the state space.

IMITATOR can also output the state space in a graphical form (option `-draw-statespace`), output the constraint synthesized in a graphical form in two dimensions (option `-draw-cart`), or output the text description of all states (option `-states-description`).

*Remark 4.1.* For all reachability/safety algorithms (including cycle synthesis), one can also use the `accepting` keyword in the model (in front of a location, see grammar in [Section 9.2](#)). In that case, one can use the `accepting` predicate inside the property state predicate. That is, one can define properties of the form:

```
property := #witness EF(accepting);
```

or even:

```
property := #synth EF(loc[pta] = 11 | (accepting & loc[pta] <> 12));
```

The semantics of the acceptance of a state is as follows: “at least one of the locations is accepting (as specified by the `accepting` keyword in the model).”

## 4.3 Safety

Often, we are rather interested in *safety* synthesis, *i.e.*, the set of valuations for which the target location is unreachable.

The property syntax is as follows:

```
property := #synth A Gnot(state_predicate);
```

<sup>1</sup>IMITATOR uses the TCTL syntax, where EF notably denotes reachability.

Internally, this algorithm works exactly the same as the reachability-synthesis, and at the end the obtained constraint is *negated* (with parameters being constrained to be included in the initial state valuations).

## 4.4 EF-minimization

This algorithm synthesizes the minimum valuation for a given parameter for which a given location is reachable. This algorithm is briefly mentioned in [And+19].

The property syntax is as follows:

```
property := #synth EFpmin(state_predicate, p);
```

where  $p$  is the parameter the value of which must be minimized.

## 4.5 EF-maximization

This algorithm is the dual of EF-minimization (see Section 4.4).

The property syntax is as follows:

```
property := #synth EFpmax(state_predicate, p);
```

## 4.6 EF with minimal time reachability

This algorithm synthesizes the parameter valuations for which a given location is reachable in minimal time [And+19].

Our algorithm uses a priority queue, with priority to the earliest successor for the selection of the next state.

The property syntax is as follows:

```
property := #synth EFtmin(state_predicate);
```

*The maximum time reachability is currently not implemented.*

## 4.7 Parameter synthesis using patterns

IMITATOR basically only supports bad state reachability synthesis on the one hand, and algorithms such as the inverse method and the cartography on the other hand. However, many correctness properties can reduce to reachability using *observers* (see [ABL98; Ace+98; Ace+03; And13b]).

Encoding observers can be done manually (using *ad-hoc* IPTA), or using predefined correctness property patterns commonly met in the literature.

**Warning 2.** These properties assume *time-progress*, i.e., if the model is stuck in a *time-lock*, these properties may give incorrect results.

If using a predefined property pattern, the property pattern must be specified as follows:

```
property := #synth pattern(<PROP>);
```

where **<PROP>** must follow the syntax of the following patterns, where **a**, **a1**, **a2** are actions, and the deadline **d** is a (possibly parametric) linear expression:

- ```
if a2 then a1 has happened before
```
- ```
everytime a2 then a1 has happened before
```
- ```
everytime a2 then a1 has happened once before
```
- ```
a within d
```
- ```
if a2 then a1 has happened within d before
```
- ```
everytime a2 then a1 has happened within d before
```
- ```
everytime a2 then a1 has happened once within d before
```
- ```
if a1 then eventually a2 within d
```
- ```
everytime a1 then eventually a2 within~d
```
- ```
if a1 then eventually a2 within d once before next
```
- ```
sequence a1, ..., an
```
- ```
always sequence a1, ..., an
```

The semantics of these properties is detailed in [\[And13b\]](#).

IMITATOR then translates the selected pattern into an additional observer automaton, and performs some safety or reachability synthesis.

## 4.8 Parametric deadlock-freeness checking

Given an NIPTA, PDFC synthesizes a parameter constraint such that, for any parameter valuation in that constraint, the system is deadlock-free [\[And16\]](#).

The property syntax is as follows:

```
property := #synth DeadlockFree;
```

As usual, IMITATOR can also output the state space in a graphical form (option `-draw-statespace`) or output the constraint synthesized in a graphical form in two dimensions (option `-draw-cart`).

## 4.9 Parametric cycle synthesis

### 4.9.1 Accepting cycle synthesis

Given an NIPTA, IMITATOR synthesizes a parameter constraint such that, for any parameter valuation in that constraint, the system contains at least one *accepting* cycle, *i.e.*, an infinite run passing infinitely often by location matching an accepting condition. Such accepting conditions are given in the form of a state predicate, *i.e.*, a conjunction or disjunction of locations and values for discrete variables. This can be seen as a liveness property, or more precisely Büchi acceptance condition.

IMITATOR implements two algorithms to this goal:

1. a basic BFS algorithm using a variant of Tarjan's strongly connected components algorithm [And+21] (Section 4.9.1.1); and
2. an NDFS algorithm with several options [NPV18; And+21] (Section 4.9.1.2).

In both cases, the property syntax is as follows:

```
property := #synth CycleThrough(state_predicate);
```

*Syntax freedom* 1. IMITATOR accepts `LoopThrough` as an equivalent keyword for `CycleThrough`.

The second algorithm (Section 4.9.1.2) is the default in IMITATOR.

As usual, IMITATOR can also output the state space in a graphical form (option `-draw-statespace`) or output the constraint synthesized in a graphical form in two dimensions (option `-draw-cart`).

#### 4.9.1.1 Accepting cycle (BFS + Tarjan)

This (non-default) algorithm can be called using the following option syntax:

```
-cycle-algo BFS
```

#### 4.9.1.2 Accepting cycle (NDFS)

This (default) algorithm can be called using the following option syntax:

```
-cycle-algo NDFS
```

It is also called when no `-cycle-algo` is specified.

Two additional options can be specified (a third one, `-pending-order`, is explained later):

1. `-layer` (or `-no-layer`) to enable (or disable) layered NDFS [NPV18];
2. `-subsumption` (or `-no-subsumption`) to enable (or disable) subsumption [NPV18].



By default, `-layer` is disabled, and `-subsumption` is enabled, as experiments showed this is the most efficient setting [NPV18].

All combinations of options yield an exact result.

At each cycle found, IMITATOR outputs its number (when using the synthesis) and the node at which the cycle was detected. When it terminates the complete parameter constraint is displayed.

When using the `-depth-limit` option, the depth-first search stops exploring the current branch and backtracks when this depth is reached.

**Exploration with layers** The exploration with layers considers different ordering policies for its pending list via the `-pending-order` option:

- `-pending-order none` : the states are added to the pending list without specific policy (default option);
- `-pending-order accepting` : the accepting states are added at the head of the pending list, the others at the tail;
- `-pending-order param` : the states are added to the list on the basis of a largest projection of zone on parameters first;
- `-pending-order zone` : the states are added to the list on the basis of a largest zone first.

**Accepting cycle (NDFS) with iterative deepening** The NDFS algorithm described above with all its options can also be applied in an iterative manner, based on depth limits. Each iteration explores a deeper state space, taking advantage of the previous computation. The iterations are applied until either the exact constraint is found or a limit is reached. This iterative analysis uses the following options :

- `-depth-init` allows for starting this iterative computing by indicating the initial depth limit to be used;
- `-depth-limit` is the maximal depth at which the last iteration will be run.
- `-depth-step` indicates the step between the depths of two iterations;

The results obtained at each iteration are displayed, thus giving valuable information to the user.

As usual, IMITATOR can also output the state space in a graphical form (option `-draw-statespace`) or output the constraint synthesized in a graphical form in two dimensions (option `-output-cart`).

#### 4.9.2 Accepting cycle with generalized acceptance condition (BFS)

Generalized Büchi conditions are supported by IMITATOR. The property syntax is as follows:

```
property := #synth CycleThrough(state_predicate_1, ..., state_predicate_n);
```

The semantics is that each of the  $n$  conditions must hold on at least one state of the same cycle, in order for this cycle to be accepting.

*Remark 4.2.* Only BFS (see [Section 4.9.1.1](#)) is available for these generalized conditions.

**Example 4.1.** Assume the following condition:

```
property := #synth CycleThrough(loc[pta1] = 11, loc[pta1] = 12 and loc[pta2]
    = 12, accepting, i = 2 or i = 3);
```

A cycle is accepting if it contains:

- a state in which `pta1` is in location `11`; and
- another state in which `pta1` is in location `12` and at the same time `pta2` is in location `12`; and
- another state syntactically labeled as `accepting` in the model; and
- another state in which either `i` is equal to 2, or `i` is equal to 3.

### 4.9.3 Any cycle synthesis

Given an NIPTA, IMITATOR synthesizes a parameter constraint such that, for any parameter valuation in that constraint, the system contains at least one cycle, *i.e.*, an infinite run (this is notably discussed in [\[AL17\]](#)).

The property syntax is as follows:

```
property := #synth Cycle;
```

This is syntactic sugar for the following property:

```
property := #synth CycleThrough(True);
```

*Syntax freedom 2.* IMITATOR accepts the `Loop` keyword as an equivalent for `Cycle`.

## 4.10 Parametric non-Zeno cycle synthesis

Given an NIPTA, IMITATOR synthesizes a parameter constraint such that, for any parameter valuation in that constraint, the system contains at least one cycle, under the non-Zeno assumption [\[And+17\]](#). That is, only parameter valuations yielding at least one non-Zeno cycle are synthesized. Parameter valuations yielding only Zeno-cycles or no cycles are ignored.

The method implemented in IMITATOR is based on CUB-IPTA, a syntactic subclass of PTA based on the CUB-TA proposed in [\[Wan+15\]](#). The precise algorithm we use, including the transformation of an arbitrary PTA into a CUB-PTA, is described in [\[And+17\]](#).

The property syntax is:

```
property := #synth NZCycle;
```

Two options are possible in IMITATOR:

1. a partial method (but slightly faster), that first detects whether the input NIPTA is already a CUB-PTA; if so, it applies non-Zeno checking on this CUB-PTA; otherwise, the returned constraint will be false.

Use `-nz-method check`.

2. a complete method (though of course without guarantee of termination), that transforms the NIPTA into a network of CUB-PTAs, and applies non-Zeno checking on the transformed CUB-PTA.

Use `-nz-method transform`.

Use `-nz-method already`.

*Warning 3* (restrictions). In contrast to most other algorithms of IMITATOR, these two algorithms work on a slightly restricted syntax:

- in a guard or invariant, each clock must be used at most once;
- invariants should not contain discrete inequalities (*i.e.*, inequalities over discrete variables only);
- the use of discrete variables was not tested (and may not be always working);
- the conditional updates are not supported;
- the use of coefficients on parameters (different from 0 or 1) was not tested;
- the use of stopwatches or multi-rate was not tested.

Overall, this part of IMITATOR has been less tested and is probably less stable than the rest of the tool.

As usual, IMITATOR can also output the state space in a graphical form (option `-draw-statespace`) or output the constraint synthesized in a graphical form in two dimensions (option `-draw-cart`).

*Remark 4.3.* The *accepting* cycles are not (yet?) supported for non-Zeno synthesis.

## 4.11 Inverse method: Trace preservation and robustness

Given an NIPTA and a reference parameter valuation, the inverse method IM synthesizes a parameter constraint such that, for any parameter valuation in that constraint, the set of traces is the same as for the reference valuation [And+09]. This problem is known as the trace-preservation synthesis, and formalized in [ALM20]. The trace-preservation emptiness problem being undecidable [ALM20], the analysis is not guaranteed to terminate (although it often does in practice).

The property syntax is of the following form:

```
property := #synth IM(parameter_valuation);
```

where `parameter_valuation` is a reference valuation of the form `p1 = 1 & p2 = 2 & ...` (see [Section 9.3](#) for details).

*Syntax freedom 3.* IMITATOR accepts the following equivalent keywords for `IM`:

- `InverseMethod`
- `TracePreservation`

IMITATOR can also output the state space in a graphical form (option `-draw-statespace`) or output the constraint synthesized in a graphical form in two dimensions (option `-draw-cart`).

Recall that IMITATOR generates a result in an external text file (`model.res`) and formatted using a standardized manner (see [Chapter 5](#)).

IMITATOR also offers two similar algorithms: `IMK` and `IMunion` return similar but slightly different results (see [\[AS11\]](#)).

## 4.12 Behavioral cartography

Given an NIPTA and a bounded parameter domain, the behavioral cartography BC synthesizes tiles, *i.e.*, parameter domains such that for any parameter valuation in that domain, the set of traces is the same [\[AF10\]](#). The corresponding problem being undecidable [\[ALM20\]](#), the analysis is not guaranteed to terminate; when it terminates, it may also leave “holes”, *i.e.*, parameter domains not covered by any tile.

The property syntax is of the following form:

```
property := #synth BCcover(hyper_rectangle);
```

where `hyper_rectangle` is a reference hyper-rectangle of the form `p1 = 1..5 & p2 = 2..4 & ...` (see [Section 9.3](#) for details).

IMITATOR can also output all state spaces in a graphical form (option `-draw-statespace`), or output the constraints synthesized in a graphical form in two dimensions (option `-draw-cart`).

When using the following syntax with an optional second argument:

```
property := #synth BCcover(hyper_rectangle, step=2/3);
```

one can specify the interval between any two points of which the coverage is checked (see [\[AF10\]](#)), here  $\frac{2}{3}$ . By default, it is 1; setting  $\frac{1}{3}$  often leads to full coverage when 1 was not enough. Any strictly positive rational (or integer) is allowed.

*Warning 4.* This optional argument is syntactically accepted for other cartography-like algorithms, but may not always be considered by the algorithm, nor was fully tested.

In addition, state spaces and separate graphical cartographies for each tile can also be generated by adding option `-tiles-files`.

## Behavioral cartography with random coverage

An alternative to the behavioral cartography is a random coverage [AF10]; it can be seen as a kind of sampling.

The property syntax is of the following form:

```
property := #synth BCRandom(hyper_rectangle , nb);
```

where `nb` is the number of times an integer point is randomly selected within the domain defined. If this point is already covered by one of the tiles, the inverse method is not called, an another point is selected. Note that `nb` represents the number of integer points randomly selected; the number of *calls* to the inverse method can be significantly smaller.

A second algorithm is `BCRandomseq` [ACE14], which first selects random point selections, and then enumerates all integer points to make sure they are actually covered. This algorithm is mostly interesting when distributed (see [ACE14; ACN15]).

## Behavioral cartography with shuffle enumeration

A second alternative to the behavioral cartography is an enumeration of all integer points in a random fashion. That is, all integer points in the reference parameter domain are generated in a data structure (an array), and then are shuffled. Then the points are enumerated [ACN15]. It differs from the random cartography in the sense that the random cartography randomly samples points without guarantee of full coverage, whereas the shuffle enumeration guarantees the coverage of all integer points.

The property syntax is:

```
property := #synth BCshuffle(hyper_rectangle);
```

## 4.13 Parametric reachability preservation

IMITATOR implements an algorithm solving the following problem: “given a reference parameter valuation  $v$  and some location  $\ell$ , synthesize other valuations that preserve the reachability of  $\ell$ ”. By preserving the reachability, we mean that  $\ell$  is reachable for the other valuations iff  $\ell$  is reachable for  $v$ .

This algorithm PRP, that somehow combines EFsynth and IM (see [And+15] for details), is called using the following property syntax:

```
property := #synth PRP(state_predicate , parameter_valuation);
```

### Parametric reachability preservation cartography

An extension of PRP to the cartography (named PRPC) is also available: PRPC synthesizes parameter constraints in which the (non-)reachability of  $\ell$  is uniform. PRPC was showed in [And+15] to be a suitable alternative to EFsynth, especially when distributed (see option `-distributed`).

The property syntax for this algorithm PRPC is as follows:

```
property := #synth PRPC(state_predicate , hyper_rectangle);
```

## 4.14 Summary

We summarize algorithms in [Table 4.1](#), recalling the basic syntax, and specifying whether they support the `#witness` mode that stops as soon as (at least) one parameter valuation is witnessed.

Table 4.1: Summary of the algorithms syntax

Algorithm	Syntax	Synth	Witness
Reachability	<code>EF(state_predicate)</code>	✓	✓
Safety	<code>AGnot(state_predicate)</code>	✓	✓
Parameter minimization	<code>EFpmin(state_predicate, p)</code>	✓	✓
Parameter maximization	<code>EFpmax(state_predicate, p)</code>	✓	✓
Minimal-time	<code>EFTmin(state_predicate)</code>	✓	✓
Cycle	<code>Cycle</code>	✓	✓
Accepting cycle	<code>CycleThrough(state_predicate)</code>	✓	✓
Accepting cycle	<code>CycleThrough(state_predicates)</code>	✓	✓
Non-Zeno cycles	<code>NZCycle</code>	✓	✗
Deadlock-freeness	<code>DeadlockFree</code>	✓	✗
Inverse method	<code>IM(parameter_valuation)</code>	✓	✗
Inverse method	<code>IMK(parameter_valuation)</code>	✓	✗
Inverse method	<code>IMunion(parameter_valuation)</code>	✓	✗
Cartography	<code>BCcover(hyper_rectangle)</code>	✓	✗
Cartography	<code>BCrandom(hyper_rectangle, nb)</code>	✓	✗
Cartography	<code>BCrandomseq(hyper_rectangle, nb)</code>	✓	✗
Cartography	<code>BCshuffle(hyper_rectangle)</code>	✓	✗
PRP	<code>PRP(state_pred, parameter_val)</code>	✓	✗
PRPC	<code>PRPC(state_pred, hyper_rect)</code>	✓	✗
Patterns	<code>pattern(&lt;pattern&gt;)</code>	✓	✓

We give in [Table 4.2](#) the default values for merging and comparison. Whenever merging is enabled (cell ✓), the following options are used by default:

```
-merge onthefly -merge-candidates queue -merge-update merge
-merge-restart off
```

## 4.15 Symbolic state space computation

Finally, instead of synthesizing parameter valuations or checking for emptiness, IMITATOR can also compute the entire symbolic state space (“parametric zone graph”). Of course, the state space may be infinite, and this analysis is not guaranteed to terminate.

The standard command is:

```
imitator model.imi -mode statespace -states-description
```

The option `-states-description` generates a file with a textual description of all states (without this option, IMITATOR will not output anything). Each state is represented with

Table 4.2: Summary of the algorithms default options

Algorithm	Syntax	Merging	-comparison
State space computation	N/A	×	equality
Reachability	EF(state_predicate)	✓	inclusion
Safety	AGnot(state_predicate)	✓	inclusion
Parameter minimization	EFpmin(state_predicate, p)	✓	inclusion
Parameter maximization	EFpmax(state_predicate, p)	✓	inclusion
Minimal-time	Eftmin(state_predicate)	✓	inclusion
Cycle	Cycle	×	equality
Accepting cycle	CycleThrough(state_predicate)	×	equality
Accepting cycle	CycleThrough(state_predicates)	×	equality
Non-Zeno cycles	NZCycle	×	equality
Deadlock-freeness	DeadlockFree	✓	inclusion
Inverse method	IM(parameter_valuation)	×	equality
Inverse method	IMK(parameter_valuation)	×	equality
Inverse method	IMunion(parameter_valuation)	×	equality
Cartography	BCcover(hyper_rectangle)	×	equality
Cartography	BCrandom(hyper_rectangle, nb)	×	equality
Cartography	BCrandomseq(hyper_rectangle, nb)	×	equality
Cartography	BCshuffle(hyper_rectangle)	×	equality
PRP	PRP(state_pred, parameter_val)	✓	inclusion
PRPC	PRPC(state_pred, hyper_rect)	✓	inclusion
Patterns	pattern(<pattern>)	✓	inclusion

its discrete part (location and values of the discrete variables), the clock and parameter constraints, and below their projection onto the parameters.

IMITATOR can also output the state space in a graphical form using option `-draw-statespace`.

## Chapter 5

# Understanding the IMITATOR result

IMITATOR generates a file `model.res`. This file has a standardized format, and can therefore be parsed using an external tool.

To disable the creation of this file, please use `-no-output-result` (see [Chapter 8](#)).

We do not give a formal grammar for this file (yet), but it can be easily inferred from example outputs.

### 5.1 Header

The file header recalls the exact version of IMITATOR used to run the analysis, including the build number, the git branch and the git SHA hash. It also recalls the model name, the exact command used, and the time when the file was generated (which may slightly differ from the time the analysis was run, if the analysis was significantly long).

Then, the header recalls global information on the model (number of IPTA, of clocks, of parameters, whether the model contains clocks with a rate  $\neq 1$ , etc.).

### 5.2 The resulting constraint

The main result of a single synthesis (*i.e.*, EFSynth, PDFC, IM and its variants, PRP...) is a constraint. This result is clearly delimited by delimiters `BEGIN CONSTRAINT` and `END CONSTRAINT`.

The result can be a convex or a non-convex constraint. In some cases, the result is made of *two* (convex or non-convex) constraints: a good constraint (characterizing good parameter valuations), and a bad constraint (characterizing bad parameter valuations). Both parts of the results are then separated using the keyword `<good|bad>` (of course the good constraint comes left of this separator, and the bad constraint comes right).

The resulting constraint comes with two other information:

- its nature, *i.e.*, whether it attempts to characterize a good set of valuations, a bad set of valuations, or both a good set and a bad of valuations. “Good” and “bad” must be understood to the property that is being checked (non-reachability of some states, deadlock-freeness, etc.). The constraint nature is only an *attempt*, as the constraint may not always be sound (see below).



- its soundness, *i.e.*, whether the constraint is exact (IMITATOR returned exactly the set of parameter valuations solution to the analysis requested), a possible under-approximation of that result, a possible over-approximation of that result (in some rare cases), or a possibly invalid constraint (in which case the result output by IMITATOR shall not be used).

Note that in almost all analyses, IMITATOR returns an exact or an under-approximated constraint. A possibly invalid constraint can be synthesized when some options are used: for example, computing the result of IM with the merging enabled (`-merge yes`) yields a possibly invalid constraint, as it was shown that the merging optimization does not preserve the validity of the result of IM [AFS13].

Finally, the result also comes with an evaluation of the termination: the termination can be regular (the analysis went to its end without interruption) or an early termination, with some states unexplored (*e.g.*, if a maximum analysis time (`-time-limit`), a maximum exploration depth (`-depth-limit`), etc., was set).

### 5.3 The cartography result

The behavioral cartography does not strictly speaking generate a result, but a list of tiles: each of them is made of the reference valuation that yielded that tile, the associated constraint again with its nature, its soundness, and the analysis termination. In addition, each tile comes with its associated number of states and transitions, and its computation time.

### 5.4 General statistics

The result file finally contains general statistics such as the global computation time (excluding the generation of graphics, or the result file), an estimation of the memory used, the number of states and transitions computed, etc. Finally, some statistics on specific operations (model parsing, graphics drawing, etc.) are given. More statistics are obtained with higher levels of verbosity, or with the `-statistics` option.

### 5.5 Projection onto some parameters

The result can be projected onto selected parameters, by using the following syntax at the end of the property file:

```
projectresult(param1, ..., paramn);
```

In that case, all parameters not in that set are eliminated using variable elimination, and the result in the result file only contains the selected parameters.

This projection is also added to the states description (see option `-states-description`) and to the graphical state space output (see option `-draw-statespace full`).

## Chapter 6

# Graphical output and translation

Again, we only give the most useful options. For more detailed commands, and a complete list of options, see [Chapter 8](#).

### 6.1 State space

To generate the (discretized) state space of a given computation in a graphical form, use:

```
imitator model.imi [property.imiprop] [options] -draw-statespace
normal
```

IMITATOR will generate a file `model-statespace.pdf`. States are made of two part: the left-hand number (e.g., `s_0`) is the number of the symbolic state, which comes from IMITATOR internal representation (`s_0` is the first one to be generated, and so on); the right-hand part displays all the discrete part of the symbolic state vertically, that is the current location in each of the NIPTA in parallel, and the value of possible discrete variables (if any). Consider state `s_1` in [Fig. 6.1b](#) (and originating from the example in [Chapter 2](#)): the current location of IPTA `proc_1` is `active1`, the current location of IPTA `proc_2` is `idle2`, while the current location of IPTA `observer` is `obs_waiting`. In addition, the current value of variable `turn` is `-1` while the current value of `counter` is `0`. The color is chosen according to IMITATOR internal choice: however, two symbolic states with the same color have the same discrete part (location and values of the discrete variables). For example, this is the case of `s_4` and `s_5` in [Fig. 6.1b](#).

Note that, beyond about 1,000 states or 1,000 transitions, the dot utility (responsible to generate the state space) may crash.

Using `-draw-statespace undetailed` makes a more compact representation, but is also less informative as only the state number and the transition labels are shown.

Conversely, `-draw-statespace full` makes a more detailed representation, by also adding to the state space all constraints: the clock and parameter constraints, and below their projection onto the parameters. This option is mostly suitable for small state spaces.

**Example 6.1.** An example of state space with no details (output using `-draw-statespace undetailed`), with normal details (using `-draw-statespace`

normal), and in verbose mode (using `-draw-statespace full`) are given in Figs. 6.1a to 6.1c.

These graphics were generated using the example in Chapter 2 with the following command:

```
imitator fischer.imi fischer.imiprop -depth-limit 4
      -draw-statespace normal
```

(where `normal` should be replaced with `undetailed` or `full`, respectively)

## 6.2 Visualizing the synthesized constraint in 2D

To visualize the constraint generated by IMITATOR using a 2-dimensional plot (thanks to the external `plot` utility), use:

```
imitator model.imi property.imiprop [options] -draw-cart
```

This will generate file `model_cart.png`.

The two dimensions chosen for the plot are the first two (non-constant) parameter dimension in the model.

Additional useful options are `-draw-cart-x-min`, `-draw-cart-x-max`, `-draw-cart-y-min`, `-draw-cart-y-max` to tune the values of the axes, and `-graphics-source` to keep the plot source.

## 6.3 Translation to UPPAAL

An automatic translation of the input model to the UPPAAL syntax [LPY97] is available. To generate an equivalent UPPAAL model without performing any analysis, use:

```
imitator model.imi -imi2Uppaal
```

IMITATOR will generate a file `model-uppaal.xml`. Note that the translation of properties is not supported.

Due to the different semantics and features between IMITATOR and UPPAAL, the translation is done in a “best effort” manner, but the semantics may differ. We review some points in the following.

**Parameters** A fundamental difference IMITATOR and UPPAAL is that UPPAAL does not support timing parameters. Parameters are therefore translated to constants, the value of which can be manually changed by the user.

**Form of the constraints** IMITATOR allows a much wider form of constraints than UPPAAL. In particular, IMITATOR allows any conjunction of linear constraints (see Section 3.1.3) while UPPAAL mainly allows clocks to be compared to integers. In addition, UPPAAL only allows invariants of the form  $x \leq c$  or  $x < c$ , which is not a restriction in IMITATOR. No checks are made when translating, and therefore UPPAAL may not accept models translated from IMITATOR input models using these features.

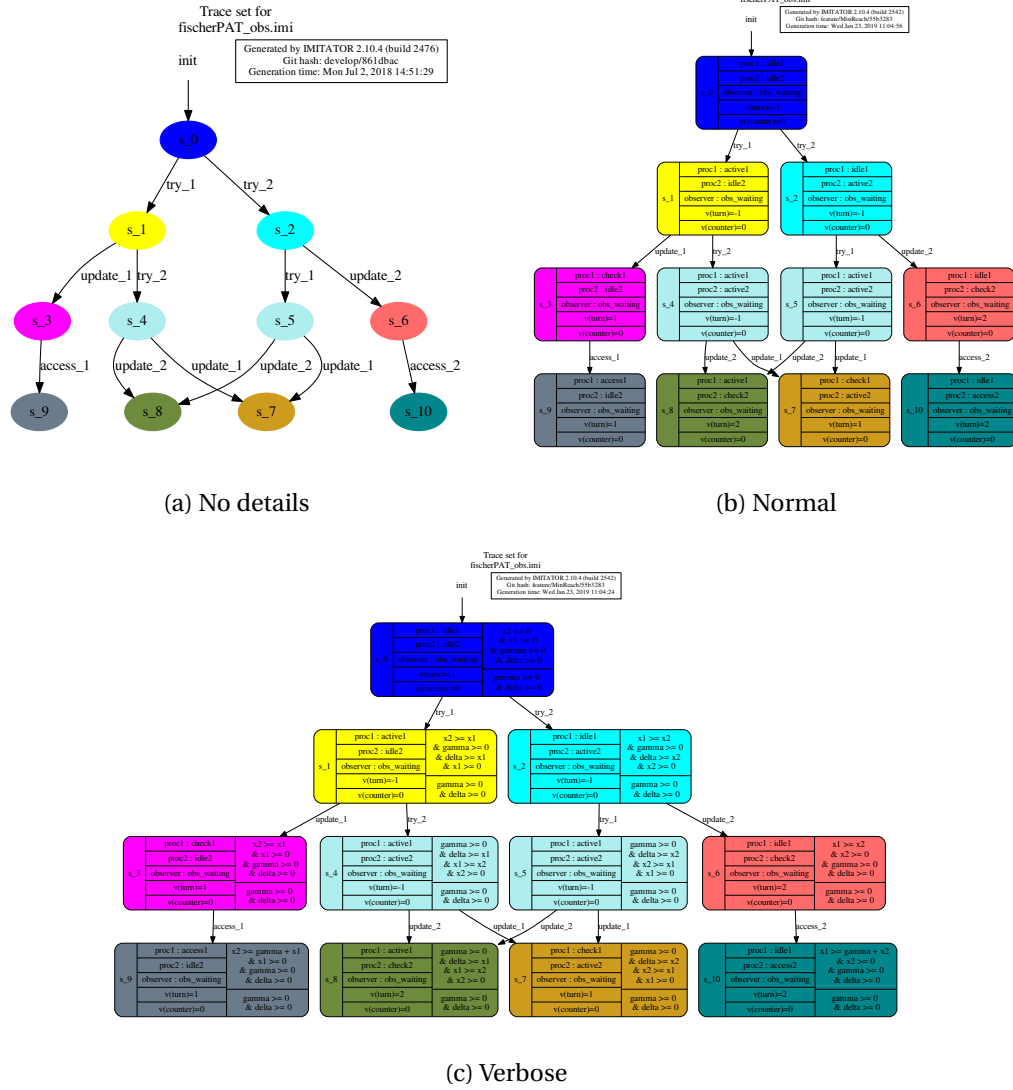


Figure 6.1: Examples of graphical state space

**Rational variables** Rational variables in IMITATOR are rational-valued with an exact value. In UPPAAL, they are interpreted as (bounded) integers: overflows may occur and, if their value becomes non-integer, it seems UPPAAL will just round them to the nearest integer, thus creating a difference in semantics.

### Binary variables

*Warning 5.* Binary words translation into UPPAAL formalism is experimental. Binary words are translated and encoded into 32-bit integer variables in UPPAAL (with dedicated functions). For this reason, translating binary words of length greater than 31 bits or shifting a binary word of an arbitrary length can lead to unexpected behaviors and results.

**Arrays** Arrays are only partially supported by our translation to UPPAAL. Array variables, array literals, arrays element access (read / write) and arrays comparisons are supported by our translation into UPPAAL.

*Warning 6.* Although it is possible in IMITATOR to compare an array variable to a literal array or literal array to another one, this type of comparison is not supported by UPPAAL.

*Warning 7.* Built-in functions operating on arrays (such as `array_concat`) are not supported by our translation.

**Lists** Lists are not supported by our translation to UPPAAL.

**Stacks** Stacks are not supported by our translation to UPPAAL.

**Queues** Queues are not supported by our translation to UPPAAL.

**Built-in functions** The only IMITATOR built-in function fully supported by our translation is the `pow` function.

Partially supported IMITATOR built-in functions are the following: `shift_left`, `shift_right`, `fill_left`, `fill_right`. Because of binary words are encoded into integer variables, these functions could have a unspecified behaviors.

*Warning 8.* Unsupported IMITATOR built-in functions will not be declared (and implemented) in UPPAAL translation, resulting to a bad program when trying to call one. The declaration and implementation of these functions in UPPAAL, if the UPPAAL input syntax allows it, is the responsibility of the user.

**Strong broadcast synchronization** IMITATOR uses a single communication model: strong broadcast (see [Section 3.3](#)), while UPPAAL uses both channel binary synchronization, and broadcast synchronization. None of them match IMITATOR's semantics. Our translation tries to preserve IMITATOR semantics, using the following scheme:

- Unnamed actions (necessarily used in a single IPTA) remain unnamed in UPPAAL.
- An action `a` used in a single IPTA is implemented in UPPAAL using a broadcast action `a!`.
- An action `a` used in exactly two IPTA is implemented in UPPAAL using a binary action `a`, where the first (*i.e.*, declared first in the IMITATOR file) uses `a!` while the second uses `a?`.
- An action `a` used in  $n$  IPTA with  $n \geq 3$  is implemented in UPPAAL using a broadcast action `a` and an additional variable `nb__a` initially set to  $n$ , and that will be responsible to check that all automata can indeed synchronize, as follows:
  - all invariants contain `nb__a = n`
  - the first automaton (*i.e.*, declared first in the IMITATOR file) involved in the synchronization synchronizes on `a!` and performs `nb__a := 1;`
  - all other automata involved in the synchronization perform `nb__a := nb__a + 1.`

From the UPPAAL official semantics, the discrete variables are updated by first executing the update label given on the `a!` transition, and then the update labels given on the `a?` for increasing, which guarantees the correctness of this construction. (This construction was suggested by Jiří Srba.)

However, a major difference (which may prevent UPPAAL to interpret the translated models) is that UPPAAL forbids guard on receiving actions (of the form `a?`). Therefore, translating an IMITATOR model where an action `a` is used in two automata (or more) with guards on more than one transition will lead UPPAAL to not accept the model. A manual editing will be needed.

**Initial constraint** The initial constraint is *not* translated, except for the initial value of the discrete variables. Notably, the clocks are initially 0 in UPPAAL, which is not necessarily the case in IMITATOR.

**Properties** Properties are so far not translated to UPPAAL.

## 6.4 Translation to HYTECH

IMITATOR supports a translation of the model to the HYTECH syntax [\[HHW95\]](#) (that is quite close to that of IMITATOR anyway). To generate an equivalent HYTECH model without performing any analysis, use:

```
imitator model.imi -imi2HyTech
```

IMITATOR will generate a file `model.hy`.

The syntax going beyond that of HYTECH (typically, Booleans, binary words, arrays, lists...), which notably includes IMITATOR built-in functions, is obviously not translated correctly. In addition, the translation of properties is not supported.

## 6.5 Translation to JANI

Since IMITATOR 3.1, IMITATOR supports a translation of the model to the [JANI specification](#) [Bud+17]. To generate an equivalent JANI model without performing any analysis, use:

```
imitator model.imi -imi2Jani
```

IMITATOR will generate a file `model.jani`. Note that the translation of properties is not supported.

Due to the different semantics and features between IMITATOR and JANI, the translation is done in a “best effort” manner, but the semantics may differ. We review some points in the following.

**Updates** IMITATOR and JANI do not have exactly the same semantics in the case of `if-then-else` updates. In order to provide a similar behavior, the translation is performed as follow.

```
if (x=1) then (x:=2, y:=3) else (x:=4)
```

is transformed (and, subsequently, written in JANI format) into

```
x := (if (x=1) then 2 else x);
y := (if (x=1) then 3 else y);
x := (if (x=1) then x else 4);
```

**Stopwatches and multi-rate clocks** If no clock of the input model has a rate different from 1, then all of them are declared in the output JANI model with the `clock` type.

Otherwise, if at least one rate uses a non-1 rate in the IMITATOR model, then *all* of the clocks (including non multi-rate clocks) are declared with the `continuous` type and the corresponding derivative (1, if non specified in the IMITATOR input model) is explicitly specified in each location of the output model.

**Synchronization** JANI synchronization semantics is more permissive than the IMITATOR one, therefore the exact meaning can be kept. It is done by listing synchronization vectors in the `syncs` entry.

**Binary words** Binary words are somehow supported by our translation to JANI, but not the built-in functions.

*Warning 9.* Built-in functions operating on binary words are not supported by our translation.

**Arrays** Arrays are only partially supported by our translation to JANI. Array variables, array literals, arrays element access (read / write) and arrays comparisons are supported by our translation into JANI. But not the built-in functions.

**Lists** Lists are not supported by our translation to JANI.

**Stacks** Stacks are not supported by our translation to JANI.

**Queues** Queues are not supported by our translation to JANI.

**Built-in functions** The only IMITATOR built-in function fully supported by our translation is the `pow` function.

*Warning 10.* Unsupported IMITATOR built-in functions will not be declared in JANI translation, resulting to a bad program when trying to call one of them. The declaration and implementation of these functions in JANI, if the JANI formalism allow it, is the responsibility of the user.

**Properties** Properties are so far not translated to JANI. The corresponding input in the JANI file exists but remains empty.

## 6.6 Export to graphics

To generate a graphic representation of the NIPTA model without performing any analysis, use:

```
imitator model.imi -imi2JPG
```

```
imitator model.imi -imi2PDF
```

```
imitator model.imi -imi2PNG
```

IMITATOR will generate a file `model-pta.jpg`, `model-pta.pdf` and `model-pta.png` respectively (using the `dot` utility).

Colored transitions are synchronized transitions (*i.e.*, shared by at least two IPTA), black transitions are local transitions, while dotted transitions are unnamed local transitions (often referred to as  $\epsilon$ -transitions in the literature).



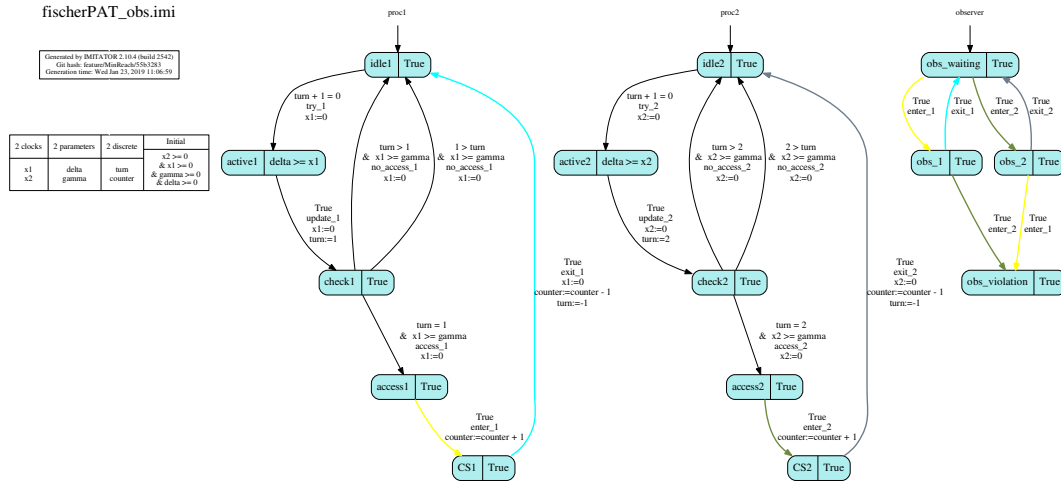


Figure 6.2: Example of PDF export

**Example 6.2.** Consider again the case study given in [Chapter 2](#) (and of which a TikZ export was given in [Fig. 2.1](#)). Then the result of the `-imi2PDF` command is given in [Fig. 6.2](#).

## 6.7 Export to $\LaTeX$

To generate a  $\LaTeX$  representation of the NIPTA model (using the `tikz` package) without performing any analysis, use:

```
imitator model.imi -imi2TikZ
```

IMITATOR will generate a file `model.tex`. This file is a standalone  $\LaTeX$  file containing a single figure, which contains the different IPTA in “subfigure” environments. The node positioning is not yet supported (locations are depicted vertically), so you may need to manually position all nodes, and bend some transitions if needed.

**Example 6.3.** Consider again the case study given in [Chapter 2](#): a TikZ export was given in [Fig. 2.1](#), with some manual positioning.

## Chapter 7

# Inside the box

### 7.1 Language and libraries

In short, IMITATOR is written in OCaml, and contains about 26,000 lines of code.

IMITATOR makes use of the following external libraries:

- The OCaml ExtLib library (Extended Standard Library for Objective Caml);
- The GNU Multiple Precision Arithmetic Library (GMP);
- The Parma Polyhedra Library (PPL) [BHZ08], used to compute operations on polyhedra.

### 7.2 Symbolic states

Verification of timed systems (and specially parametric timed systems) is necessarily done in a symbolic manner, in the sense that the timing information is abstracted by clock constraints. However, IMITATOR does not perform what is referred to as *symbolic model checking*; in other words, the representation of locations in IMITATOR is explicit (and not symbolic using, *e.g.*, binary decision diagrams).

In short, a symbolic state in IMITATOR is made of the following elements:

- the current location (index) of each IPTA;
- the current value of the discrete variables;
- a constraint on  $X \cup P$  representing the continuous information.

In IMITATOR, all rationals (*i.e.*, the value of the rational-valued discrete variables and the coefficients used in the constraints) are unbounded rationals (implemented using GMP). Integers over 32 bits are used to encode `int` variables. No floating-point approximations (`float`) are used, except for the generation of graphics.

## 7.3 Type system

Since IMITATOR 3.1, several types of discrete variables are supported. Common primitive types as integer, rational and Booleans are available. Recall from [Section 3.5](#) that they can be declared with the following keywords:

- Rational: `rational` (or `discrete` for backward-compatibility)
- Integer (32 bits): `int`
- Boolean: `bool`
- Binary word: `binary(l)` with `l` the length of the word

These types are divided into three categories: numerical (for `rational` and `int`), Booleans (for `bool`) and Binary (for `binary(l)`).

IMITATOR models are statically typed, in the sense of their variables and expressions types are resolved prior to the algorithms execution.

### 7.3.1 Type checking

In order to verify the model type consistency, a step called *type checking* is applied—during the semantic analysis prior to the start of the verification or synthesis algorithm—on all expressions of the model: declarations, assignments, updates, conditions (`if-then-else`), guards and invariants. Type checking does the following :

- It checks type consistency within the expressions;
- It resolves the types of the expressions;
- It applies a numerical type to literal numbers according to the numerical type held by their outer expression;
- It checks consistency between an expression type and variable type, in case of an assignment;
- It checks the consistency between an expression type and its grammar rule.

### 7.3.2 Expression type solving

The expression tree is traversed (in a depth-first search manner) in order to deduce its type. As soon as a variable, constant or literal is found, the expression takes its type. If the expression only contains numbers (e.g., `2 + 3 * 4`), the inferred type will be a *rational* arithmetic expression (and not the `int` type).

IMITATOR does not support implicit conversion. In other words, `r + i + 3`, where `r` is a rational and `i` is an `int`, is not allowed. Neither is allowed `r := i`.

If an expression mixes different types, a type error is risen before the start of the algorithm.

### 7.3.3 Literal number type inference

The types of literal numbers are automatically resolved during the semantic analysis and more precisely during the type checking step. Once an expression type is resolved, all literal numbers within this expression are automatically typed as the numerical type held by the expression.

For example, in the following update expression `b := True & i < 3 + 1` with `i : int`, literal numbers `3` and `1` will be automatically typed as an `int`.

### 7.3.4 Type conversion

As said previously, there is no implicit conversion of variables, constants or literals from one type to another type. All expressions hold exactly one type and cannot mix different types. However, it is possible to convert an expression to another type using built-in conversion functions:

- `rational_of_int`: converts an `int` expression to a rational one.

## 7.4 Installation

This document does not aim at explaining how to install IMITATOR. See the installation information available on the website for the most up-to-date information.

Binaries and source code packages are available on IMITATOR's Web page [\[IMI-Web\]](#). Several standalone binaries are provided for Linux systems, that require no installation.

## Chapter 8

# List of options

The options available for IMITATOR are explained in the following.

Note that some more options are available in the current implementation of IMITATOR. If these options are not listed here, they are experimental (or deprecated). If needed, more information can be obtained by contacting the IMITATOR team.

**-acyclic (default: disabled)** Does not test if a new state was already encountered. Without this option, 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 consequence, 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-tiles-limit <limit> (default: none)** In cartography algorithm, set up a maximum of tiles to be generated by the algorithm.

**-cart-time-limit <limit> (default: none)** In cartography algorithm, set up a global time limit to the algorithm. In contrast, **-time-limit** is applied to each call to the inverse method.

**-comparison (default: depending on the algorithm)** This option specifies how to compare the constraint of a newly computed symbolic state with the constraints of the formerly computed symbolic states. Possible values are:

**none** : Does not test if a new state was already encountered. To be set to this value *only* if the state space is a tree with all states being different (otherwise analysis may enter an infinite loop and never terminate).

**equality** Test the *equality* of the constraints.

**inclusion** Consider an *inclusion* of parametric zone instead of the equality when computing the successors of a set of states. In other terms, when encountering a new state, IMITATOR checks whether an old state shares the same discrete part (location and value of the discrete variables) and is such that the new state constraint is included into (*i.e.*, is smaller than or equal to) the old state constraint; if so, IMITATOR discards this “new” state.

**including** When encountering a new state, IMITATOR checks whether an old state shares the same discrete part (location and value of the discrete variables) and is such that the new state constraint includes (*i.e.*, is larger than or equal to) the old state constraint; if so, IMITATOR replaces the old state with the new state, and explores it further.

**doubleinclusion** Consider a bidirectional inclusion of parametric zones when computing the successors of a set of states. When this value is enabled, it suffices that a previous state with the same location and a constraint greater than or equal to (*resp.* *smaller or equal* to) the constraint of the new state has been encountered to discard the new state (*resp.* the old state, which is replaced by the new one).

It seems that, although less states are computed, this value is *less* efficient than **-comparison inclusion** (in part due to the extra inclusion checks required by **-comparison doubleinclusion**). This explains why this option, while preserving the correctness, is not default for reachability algorithms.

In reachability algorithms (Sections 4.2 to 4.7 and 4.13), **-comparison inclusion** is usually enabled by default; however, for loop synthesis algorithms (Sections 4.9 and 4.10), **-comparison equality** is used, as inclusion may introduce spurious cycles. See Table 4.2 for the default behavior of algorithms w.r.t. this option.

**-contributors** Print the list of contributors and exits.

**-cycle-algo (default: NDFS)** For cycle synthesis algorithms (Section 4.9), call either **BFS** (breadth-first search) or **NDFS** (nested depth-first search), with some additional options in the latter case. See **-layer**, **-subsumption**, **-pending-order**, **-depth-init**, **-depth-limit**, **-depth-step** for possible additional options.

**-depth-init <initial\_depth> (default: none)** For the cycle detection using NDFS (Section 4.9.1.2), setting **-depth-init** allows for running iteratively NDFS, starting with depth limit **<initial\_depth>** and increasing this limit at each iteration.

**-depth-limit <limit> (default: none)** Limits the depth of the exploration of the state space. In the cartography mode, this option gives a limit to *each* call to the inverse method. Setting **-depth-limit** guarantees the termination of any execution of IMITATOR, but not necessarily the correctness of the algorithms.



`-depth-step` (**default: 1**) For the cycle detection using NDFS (Section 4.9.1.2), `-depth-step` is used for iterative NDFS: it indicates the step between two iterations.

`-distributed <mode>` (**default: not distributed**) Distributed version of the behavioral cartography. Various distribution modes are possible:

`no` Non-distributed mode (default)

`static` Static domain decomposition [ACN15];

`sequential` Master-worker scheme with sequential point distribution [ACE14]

`randomXX` Master-worker scheme with random point distribution (e.g., `random5` or `random10`); after `XX` successive unsuccessful attempts (where the generated point is already covered), the algorithm will switch to an exhaustive sequential iteration [ACE14]

`shuffle` Master-worker scheme with shuffle point distribution [ACN15]

`dynamic` Master-worker dynamic subdomain decomposition [ACN15]

`-draw-cart` (**default: off**) After execution of the behavioral cartography or EFsynth, plots the generated zones as a .png file. This will generate file `model_cart.png`. If the model contains more than two parameters, then `-draw-cart` will plot the projection of the generated zones on the first two parameters of the model (or on the two *varying* parameters in the case of BC).

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 current directory, unless option `-output-prefix` is used.

Additional useful options are `-draw-cart-x-min`, `-draw-cart-x-max`, `-draw-cart-y-min`, `-draw-cart-y-max` to tune the values of the axes, and `-graphics-source` to keep the plot source.

`-draw-cart-x-min` (**default: off**) Set minimum value for the  $x$  axis when plotting the cartography (not entirely functional in all situations yet).

`-draw-cart-x-max` (**default: off**) Set maximum value for the  $x$  axis when plotting the cartography (not entirely functional in all situations yet).

`-draw-cart-y-min` (**default: off**) Set minimum value for the  $y$  axis when plotting the cartography (not entirely functional in all situations yet).

`-draw-cart-y-max` (**default: off**) Set maximum value for the  $y$  axis when plotting the cartography (not entirely functional in all situations yet).

**-draw-statespace (default: none)** Graphical output using dot. In this case, IMITATOR outputs a file `<input_file>.pdf`, which is a graphical output in the PDF format, generated using dot, corresponding to the state space.

Note that the path and the name of those two files can be changed using the **-output-prefix** option.

Three values for this option are allowed:

- Use value **undetailed** for the structure only (no location names): does *not* provide detailed information on the local locations of the composed IPTA, and instead only outputs the state id. Enabling the option with this value instead of **-draw-statespace normal** may yield a smaller graph, which is useful when generating relatively large state spaces.
- Use value **normal** for location names.
- Use value **full** for location names and constraints. This provides very detailed information, by adding to the right of the local locations of the composed IPTA the associated constraint as well. In addition, the parametric constraint is printed too. Enabling the option with this value instead of **-draw-statespace normal** will yield a very large graph, and it is useful (and readable) mostly for very small state spaces.

**-dynamic-elimination (default: disabled)** Dynamic elimination of clocks that are known to not be used in the future of the current state [And13a]. That is, IMITATOR deletes from the symbolic states all clocks that will not be used in a guard or an invariant until their next reset. This technique uses some basic static analysis of the IPTA composing the NIPTA.

**-expl-order** Exploration orders are considered for standard reachability analysis (and counterexample analysis, using **#witness**), and are documented in [ANP17].

- **-expl-order layerBFS**: layer-based breadth-first search (default);
- **-expl-order queueBFS**: queue-based breadth-first search;
- **-expl-order queueBFSRS**: queue-based breadth-first search with ranking system
- **-expl-order queueBFSPRIOR**: priority-based BFS with ranking system.

This framework remains relatively experimental.

**-extrapolation (default: disabled)** Clock extrapolation was defined in a number of papers for timed automata (e.g., [Beh+03; Beh+06; Li09; Tri09]), and then extended to parametric timed automata in [ALR15; AA22].

Values for this option are:

<b>none</b>	no extrapolation
<b>M</b>	simple <i>M</i> -extrapolation (maximum constant computed for each clock)
<b>Mglobal</b>	simple <i>M</i> -extrapolation with single maximum constant for all clocks
<b>LU</b>	LU-extrapolation (constants computed for each clock)
<b>LUglobal</b>	LU-extrapolation with single maximum constants for all clocks



*Warning 11* (syntax restriction). Extrapolation can be used only when all guards and invariants are “simple clock guards”, *i.e.*, of the form  $x \triangleright \triangleleft lt$ , where  $lt$  is a linear term over  $P$ . In other words, no discrete variables nor linear constraints over the clocks (which notably excludes “diagonal” constraints) are allowed.

**-graphics-source (default: disabled)** Keep file(s) used for generating graphical output (*e.g.*, state space, cartography); these files are otherwise deleted after the generation of the graphics.

**-imi2HyTech (default: disabled)** Translates the input model to a HyTECH model, and exits. See [Section 6.4](#) for details.

**-imi2IMI (default: disabled)** Regenerates the model into an IMITATOR model, and exits.

**-imi2JPG (default: disabled)** Translates the input model to a graphical, human-readable form (in .jpg format), and exits.

**-imi2PDF (default: disabled)** Translates the input model to a graphical, human-readable form (in .pdf format), and exits.

**-imi2PNG (default: disabled)** Translates the input model to a graphical, human-readable form (in .png format), and exits.

**-imi2TikZ (default: disabled)** Translates the input model to a  $\text{\LaTeX}$  representation of the model (using the tikz package) without performing any analysis, and exits. Note that node positioning is not (much) supported, so may want to edit manually some positions.

**-imi2Uppaal (default: disabled)** Translates the input model to a UPPAAL model, and exits. See [Section 6.3](#) for details.

**-layer (default: disabled)** For the cycle detection using NDFS ([Section 4.9.1.2](#)), consider layered NDFS [\[NPV18\]](#).

**-no-layer (default: disabled)** For the cycle detection using NDFS ([Section 4.9.1.2](#)), do not consider layered NDFS [\[NPV18\]](#).

**-merge (default: depending on the algorithm)** Use the merging techniques of [\[AFS13; And+22\]](#). This option is safe (and used by default) for the EFSynth algorithm, and similar algorithms. However, not all the properties of the inverse method are preserved when using merging (see [\[AFS13\]](#) for details).

The possible values for this option are:

<code>none</code>	No merging even if the algorithm requires it by default
<code>onthe-fly</code>	Update the statespace by deleting the merged state and updating its transitions <i>in situ</i> [And+22]
<code>reconstruct</code>	Update the statespace with a copy of the reachable part [And+22]
<code>2.12</code>	Former version of merging in IMITATOR 2.12 [AFS13] reimplemented into IMITATOR 3.3

See Table 4.2 for the default value of this option depending on the algorithm.

**-merge-candidates (default: depending on the algorithm)** Use the merging technique of [And+22] and specify the candidates to merge with a state. The value of option `-merge` must be different from `none` for this option to be enabled.

The possible values for this option are:

<code>ordered</code>	First within the queue, and then the visited states
<code>queue</code>	Only the states in the queue
<code>visited</code>	All the visited states

See Table 4.2 for the default value of this option depending on the algorithm.

**-merge-update (default: depending on the algorithm)** Use the merging technique of [And+22] and specify when to update the statespace. The value of option `-merge` must be different from `none` for this option to be enabled.

The possible values for this option are:

<code>candidates</code>	After having processed the whole candidate list of a state
<code>merge</code>	After each successful merge with each sibling

See Table 4.2 for the default value of this option depending on the algorithm.

**-merge-restart (default: depending on the algorithm)** Use the merging technique of [And+22] and set if a state can be merged to one of its siblings, if the search through all candidate siblings is restarted. The value of option `-merge` must be different from `none` for this option to be enabled.

The possible values for this option are:

<code>on</code>
<code>off</code>

See Table 4.2 for the default value of this option depending on the algorithm.

**-mode (default: none)** When executing IMITATOR without a property, *i.e.*, in a special mode.

The possible values for this option are:

`checksyntax` Simple syntax check and no analysis.

`statespace` Generation of the entire parametric state space  
(see [Section 4.15](#))

`-no-acceptfirst` (**default: disabled**) For the cycle detection using NDFS ([Section 4.9.1.2](#)), do not put accepting states at the head of the successors list.

`-no-cumulative-pruning` (**default: disabled**) In EFsynth and BFS-based (accepting) loop synthesis, no inclusion test of the new states parameter constraints in the already synthesized constraint is performed when this option is enabled. Otherwise, by default, the algorithm checks whether a new state parameter constraint is included into the already computed parameter constraint and, if so, cuts the branch. This can save time by cutting branches, but can also slow down the analysis for very complex constraints with dozens or hundreds of disjunctions. This option was designed when using IMITATOR for *monitoring* or *parametric timed pattern matching*, when constraints can contain up to thousands of disjunctions [[AHW18](#)].

`-no-lookahead` (**default: disabled**) For the cycle detection using NDFS ([Section 4.9.1.2](#)), do not perform a lookahead for finding successors closing an accepting cycle.

`-no-random` (**default: disabled**) In the inverse method, no random selection of the  $\pi_0$ -incompatible inequality (select the first found). By default, select an inequality in a random manner.

`-no-var-autoremove` (**default: disabled**) Usually, IMITATOR automatically removes from the analysis the variables declared in the header, but used nowhere in the IPTAs nor in the correctness property. Note that a variable updated (to 0 or to any other value) is *not* considered used, as long as it is not used elsewhere (*i.e.*, in a guard, an invariant, a value to be updated to, a property...). Using `-no-var-autoremove` prevents IMITATOR from automatically remove these variables.

`-nz-method` (**default: transform**) For non-Zeno cycle synthesis ([Section 4.10](#)), choose to either:

- synthesize valuations only for the subset of valuations for which the NIPTA already satisfies the CUB assumption (`-nz-method check`), or
- transform the NIPTA into an equivalent CUB-NIPTA (`-nz-method transform`).

`-output-float` (**default: disabled**) Convert (exact-valued) discrete rational variables into (possibly approximated) floats in all outputs.

`-output-prefix` (**default: <input\_file>**) Set the path prefix for all generated files. The path can be either relative (to the path to the `./imitator` binary) or absolute, and must be followed by the file name.

Examples:



- `-output-prefix log`
- `-output-prefix ./log`
- `-output-prefix /home/imitator/outputs`

By default, the path prefix is the *current directory*, *i.e.*, where the user is (therefore not necessarily the model directory).

`-no-output-result` (**default: enabled**) Does not write the result of the analysis to a file. By default, a file named `<input_file>.res` is created using a normalized syntax, that can be easily parsed, *e.g.*, using an external script.

`-pending-order` (**default: none**) For the cycle detection using NDFS ([Section 4.9.1.2](#)), defines the ordering strategy used for the pending list in NDFS exploration with layers. The possible values are:

<code>none</code>	the states are added to the pending list without specific policy (default option).
<code>accepting</code>	the accepting states are added at the head of the pending list, the others at the tail.
<code>param</code>	the states are added to the list on the basis of a largest projection of zone on parameters first.
<code>zone</code>	the states are added to the list on the basis of a largest zone first.

`-recompute-green` (**default: disabled**) For the cycle detection using NDFS ([Section 4.9.1.2](#)) with iterative deepening, force re-exploration of green states when encountered on a shorter path.

`-states-description` (**default: disabled**) Generates a file `<input_file>.states` describing the reachable states in plain text (value of the location, of the discrete variables, associated constraint, and its projection onto the parameters).

`-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 (*i.e.*, the exploration of the state space at the current depth) before stopping.

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

`-subsumption` (**default: enabled for NDFS**) For the cycle detection using NDFS ([Section 4.9.1.2](#)), use subsumption [[NPV18](#)].

`-no-subsumption` (**default: enabled for NDFS**) For the cycle detection using NDFS ([Section 4.9.1.2](#)), do not use subsumption [[NPV18](#)].



**-sync-auto-detect (default: disabled)** IMITATOR considers that all the IPTA declaring a given action must be able to synchronize all together, so that the synchronization can happen. By default, IMITATOR considers that the actions declared in an IPTA are those declared in the `synclabs` section. Therefore, if an action is declared but never used in (at least) one IPTA, this label will never be synchronized in the execution.<sup>1</sup>

The option `-sync-auto-detect` allows to detect automatically the actions in each IPTA: the actions declared in the `synclabs` section are ignored, and IMITATOR considers as declared actions only the actions really used in this IPTA.

**-tiles-files (default: disabled)** In cartography, generates the required files for each tile (*i.e.*, the `.res` files, as well as the graphical cartography files whenever `-draw-cart` is enabled).

**-time-elapsing-after (default: disabled)** When computing a new symbolic state, compute the time elapsing before taking the transition instead of after.

**-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 the exploration of the state space at the current depth). In the cartography mode, this option represents a *global* time limit, not a limit for each call to the inverse method.

**-timed (default: disabled)** Add a timing information to each shell output of the program.

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

<code>mute</code>	No output (the result is still output to an external file)
<code>warnings</code>	Prints only warnings
<code>standard</code>	Give little information (number of steps, computation time)
<code>experiments</code>	Give some additional information, typically enough for experiments
<code>low</code>	Give some additional information on what happens internally
<code>medium</code>	Give quite a lot of information
<code>high</code>	Give much information
<code>total</code>	Give really too much information

**-version** Prints IMITATOR header including the version number and exits.

<sup>1</sup>In such a case, action label is actually completely removed before the execution, in order to optimize the execution, and the user is warned of this removal.

# Chapter 9

## Grammar

We give in this chapter the complete grammar of input models and properties for IMITATOR.

### 9.1 Variable names

A variable name (represented by `<name>` in the grammar below) is a string starting with a letter (small or capital), and followed by a set of letters, digits and underscores (“\_”). By letter we mean the 26 letters of the Latin alphabet, without any diacritic mark.

The set of clock names, parameter names and discrete variable names must (quite naturally) be disjoint. However, the sets of IPTA names, location names, action names, and variable names are not required to be disjoint. That is, the same name can be given to a clock, an automaton, an action and a location.

Furthermore, the names of the sets of locations of various IPTA are not-necessarily disjoint either: that is, a same name can be given to two different locations in two different IPTA (and they still represent two different things).

### 9.2 Grammar of the model

The IMITATOR input model is described by the following grammar. Non-terminals appear *<within chevrons>*. 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  $\epsilon$  denotes the empty string.

```
<imitator_input> ::  
  <automata_descriptions> <init>
```

We define each of those two components below.

#### 9.2.1 Automata descriptions

```
<include_file_list> ::  
  <include_file> <include_file_list>  
  |  $\epsilon$ 
```

```

<include_file> ::
    #include "<path>" ;

<automata_descriptions> ::
    <include_file_list> <declarations> <automata>

<declarations> ::
    var <var_lists>
    | ε

<var_lists> ::
    <var_list> : <var_type> ; <var_lists>
    | ε

<var_list> ::
    <name>
    | <name> = <global_expression>
    | <name> , <var_list>
    | <name> = <global_expression> , <var_list>

<var_type> ::
    <var_type_discrete>
    | clock
    | parameter

<var_type_discrete> ::
    <var_type_scalar>
    | <var_type> array( <arithmetic_expression> )
    | <var_type> list
    | <var_type> stack
    | <var_type> queue

<var_type_scalar> ::
    | binary ( <pos_integer> )
    | bool
    | constant
    | discrete
    | int
    | rational

<automata> ::
    <automaton> <automata>
    | <include_file> <automata>
    | ε

<automaton> ::
    automaton <name> <prolog> <locations> end

```

```

<prolog> ::
| <sync_labels>
|  $\epsilon$ 

<sync_labels> ::
  synclabs : <name_list> ;

<name_list> ::
  <name_nonempty_list>
|  $\epsilon$ 

<name_nonempty_list> ::
  <name> , <name_nonempty_list>
| <name>

<locations> ::
  <location> <locations>
|  $\epsilon$ 

<location_attribute> ::
|  $\epsilon$ 
| urgent
| accepting
| accepting urgent
| urgent accepting

<location> ::
  <location_attribute> loc <name> : invariant <nonlinear_convex_predicate>
  <stopw_and_flow_opt> <transitions>

<stopw_and_flow_opt> ::
  <stopwatches> <flow>
| <flow> <stopwatches>
| <flow>
| <stopwatches>
|  $\epsilon$ 

<flow> ::
  flow { <flow_list> }

<flow_list> ::
  <flow_nonempty_list>
|  $\epsilon$ 

<flow_nonempty_list> ::
  <single_flow> , <flow_nonempty_list>
| <single_flow> <comma_opt>

```



```

<single_flow> ::
    <name>' = <constant_arithmetic_expr>

<stopwatches> ::
    stop{ <name_list> }

<transitions> ::
    <transition> <transitions>
    | ε

<transition> ::
    when <nonlinear_convex_predicate> <update_synchronization> goto <name> ;

<update_synchronization> ::
    <updates>
    | <syn_label>
    | <updates> <syn_label>
    | <syn_label> <updates>
    | ε

<updates> ::
    do { <seq_then_updates> }

<seq_then_updates> ::
    seq <update_seq_nonempty_list> <then_updates>
    | <update_list>

<then_updates> ::
    then <update_list> [end]
    | ε

<update_list> ::
    <update_nonempty_list>
    | ε

<update_nonempty_list> ::
    <update> , <update_list>
    | <condition_update> , <update_list>
    | <update>
    | <condition_update>

<update_seq_nonempty_list> ::
    <update> ; <update_list>
    | <condition_update> ; <update_list>
    | <update>
    | <condition_update>

<update> ::
    <name> := <global_expression>

```

```

<normal_update_list> ::
  <update> , <normal_update_list>
| <update>
| ε

<conditon_update> ::
  if ( <boolean_expression> ) then <normal_update_list> end
| if ( <boolean_expression> ) then <normal_update_list> else <normal_update_list> end

<syn_label> ::
  sync <name>

<global_expression> ::
  <arithmetic_expression>
| <binary_word_expression>
| <boolean_expression>
| <array_expression>
| <list_expression>
| <stack_expression>
| <queue_expression>

<array_expression> ::
  ( <array_expression> )
| array_append ( <array_expression> , <array_expression> )
| array_length ( <array_expression> )
| array_mem ( <global_expression> , <array_expression> )
| <name>
| <literal_array>

<list_expression> ::
  ( <list_expression> )
| list_cons ( <global_expression> , <list_expression> )
| list_hd ( <list_expression> )
| list_is_empty ( <list_expression> )
| list_length ( <list_expression> )
| list_mem ( <global_expression> , <list_expression> )
| list_rev ( <list_expression> )
| list_tl ( <list_expression> )
| <name>
| <literal_list>

<stack_expression> ::

```

```
    ( <stack_expression> )
|  stack_push ( <global_expression> , <stack_expression> )
|  stack_pop ( <stack_expression> )
|  stack_top ( <stack_expression> )
|  stack_clear ( <stack_expression> )
|  stack_is_empty ( <stack_expression> )
|  stack_length ( <stack_expression> )
|  <name>
|  stack()

<queue_expression> ::
    ( <queue_expression> )
|  queue_push ( <global_expression> , <queue_expression> )
|  queue_pop ( <queue_expression> )
|  queue_top ( <queue_expression> )
|  queue_clear ( <queue_expression> )
|  queue_is_empty ( <queue_expression> )
|  queue_length ( <queue_expression> )
|  <name>
|  queue()

<literal_array> ::
    [ ]
|  [ <literal_expression_fol> ]

<literal_expression_fol> ::
    <global_expression> , <literal_expression_fol>
|  <global_expression>

<array_access> ::
    <array_expression> [ <arithmetic_expression> ]
|  <name> [ <arithmetic_expression> ]

<literal_list> ::
    list([ ])
|  list([ <literal_expression_fol> ])

<binary_word_expression> ::
```

```

    ( <binary_word_expression> )
|  shift_left ( <binary_word_expression> , <arithmetic_expression> )
|  shift_right ( <binary_word_expression> , <arithmetic_expression> )
|  fill_left ( <binary_word_expression> , <arithmetic_expression> )
|  fill_right ( <binary_word_expression> , <arithmetic_expression> )
|  log_and ( <binary_word_expression> , <binary_word_expression> )
|  log_or ( <binary_word_expression> , <binary_word_expression> )
|  log_xor ( <binary_word_expression> , <binary_word_expression> )
|  log_not ( <binary_word_expression> )
|  <array_access>
|  <name>
|  <binary_word>

```

<boolean\_expression> ::

```

    <boolean_expression> & <boolean_expression>
|  <boolean_expression> | <boolean_expression>
|  <arithmetic_expression> <relop> <arithmetic_expression>
|  <discrete_boolean_expression>

```

<discrete\_boolean\_expression> ::

```

    <arithmetic_expression> <relop> <arithmetic_expression>
|  <arithmetic_expression> in [ <arithmetic_expression> , <arithmetic_expression> ]
|  ( <boolean_expression> )
|  not ( <boolean_expression> )
|  <array_access>
|  <name>
|  True
|  False

```

<arithmetic\_expression> ::

```

    <arithmetic_term>
|  <arithmetic_expression> + <arithmetic_term>
|  <arithmetic_expression> - <arithmetic_term>

```

<arithmetic\_term> ::

```

    <arithmetic_factor>
|  <rational> <name>
|  <arithmetic_term> * <arithmetic_factor>
|  <arithmetic_term> / <arithmetic_factor>
|  - <arithmetic_term>

```

<arithmetic\_factor> ::

```

    ( <arithmetic_expression> )
|  pow( <arithmetic_expression> , <arithmetic_expression> )
|  rational_of_int( <arithmetic_expression> )
|  <array_access>
|  <name>
|  <rational>

```

```

<convex_predicate> ::
    & <convex_predicate_fol>
    | <convex_predicate_fol>

<convex_predicate_fol> ::
    <linear_constraint> & <convex_predicate>
    | <linear_constraint>

<nonlinear_convex_predicate> ::
    & <nonlinear_convex_predicate_fol>
    | <nonlinear_convex_predicate_fol>

<nonlinear_convex_predicate_fol> ::
    <nonlinear_constraint> & <nonlinear_convex_predicate>
    | <nonlinear_constraint>

<linear_constraint> ::
    <linear_expression> <relop> <linear_expression>
    | True
    | False

<nonlinear_constraint> ::
    <discrete_boolean_expression>
    | True
    | False

<relop> ::
    <
    | <=
    | =
    | <>
    | >=
    | >

<linear_expression> ::
    <linear_term>
    | <linear_expression> + <linear_term>
    | <linear_expression> - <linear_term>

<linear_term> ::
    <rational>
    | <rational> <name>
    | <rational> * <name>
    | - <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>

```

### 9.2.2 Initial state

```

<init> ::
  init := { <init_discrete_continuous_definition> }

```

```

<init_discrete_continuous_definition> ::
  <init_discrete_definition> <init_continuous_definition>
| <init_continuous_definition> <init_discrete_definition>

```

```

<init_discrete_definition> ::
  discrete = <init_discrete_expression> ;
| ε

```

```

<init_discrete_expression> ::
  , <init_discrete_expression_nonempty_list>
| <init_discrete_expression_nonempty_list>
| ε

```

```

<init_discrete_expression_nonempty_list> ::
  <init_discrete_state_predicate> , <init_discrete_expression_nonempty_list>
| <init_discrete_state_predicate>
| <init_discrete_state_predicate> ,

```

```

<init_discrete_state_predicate> ::
  <name> := <global_expression>
| <init_discrete_loc_predicate>
| ( <init_discrete_state_predicate> )

```

```

<init_discrete_loc_predicate> ::
    loc[ <name> ] := <name>
|   <name> is in <name>

<init_continuous_definition> ::
    continuous = <region_expression> ;
|   ε

<region_expression> ::
    & <region_expression_fol>
|   <region_expression_fol>

<region_expression_fol> ::
    <linear_constraint>
|   ( <region_expression_fol> )
|   <region_expression_fol> & <region_expression_fol>

```

### 9.3 Grammar of the property file

```

<property_definition> ::
    <property_kw_opt> <quantified_property>

<property_kw_opt> ::
    property :=
|   ε

<quantified_property> ::
    #synth <property> <semicolon_opt> <projection_definition_opt>
|   #exhibit <property> <semicolon_opt> <projection_definition_opt>
|   #witness <property> <semicolon_opt> <projection_definition_opt>

<property> ::

```

```

    EF <state_predicate>
  | AGnot <state_predicate>
  | EFpmin ( <state_predicate> , <name> )
  | EFpmax ( <state_predicate> , <name> )
  | EFtmin <state_predicate>
  | Cycle
  | Loop /* alias for Cycle */
  | CycleThrough <state_predicate_list>
  | LoopThrough <state_predicate_list> /* alias for CycleThrough */
  | NZCycle
  | DeadlockFree
  | IM ( <parameter_valuation> )
  | InverseMethod ( <parameter_valuation> ) /* alias for IM */
  | TracePreservation ( <parameter_valuation> ) /* alias for IM */
  | IMconvex ( <parameter_valuation> )
  | IMK ( <parameter_valuation> )
  | IMunion ( <parameter_valuation> )
  | PRP ( <parameter_valuation> )
  | BCcover ( <hyper_rectangle> <step_opt> )
  | BCshuffle ( <hyper_rectangle> <step_opt> )
  | BCRandom ( <hyper_rectangle> , <pos_integer> <step_opt> )
  | BCRandomseq ( <hyper_rectangle> , <pos_integer> <step_opt> )
  | PRPC ( <state_predicate> , <hyper_rectangle> <step_opt> )
  | pattern( <pattern> )

```

<pattern> ::

```

  | if <name> then <name> has happened before
  | everytime <name> then <name> has happened before
  | everytime <name> then <name> has happened once before
  | <name> within <linear_expression>
  | if <name> then <name> has happened within <linear_expression> before
  | everytime <name> then <name> has happened within <linear_expression> before
  | everytime <name> then <name> has happened once within <linear_expression> before
  | if <name> then eventually <name> within <linear_expression>
  | everytime <name> then eventually <name> within <linear_expression>
  | everytime <name> then eventually <name> within <linear_expression> once before next
  | sequence <var_list>
  | sequence ( <var_list> )
  | always sequence <var_list>
  | always sequence ( <var_list> )

```

<state\_predicate\_list> ::

```

  | <non_empty_state_predicate_list>
  | ε

```

<non\_empty\_state\_predicate\_list> ::



```

| <state_predicate> , <non_empty_state_predicate_list>
| <state_predicate> <comma_opt>

<state_predicate> ::
    <state_predicate> || <state_predicate>
| <state_predicate_term>
| True
| False
| accepting
| ε

<state_predicate_term> ::
    <state_predicate_term> & <state_predicate_term>
| <state_predicate_factor>

<state_predicate_factor> ::
    <simple_predicate>
| not <state_predicate_factor>
| ( <state_predicate> )

<simple_predicate> ::
    <discrete_boolean_predicate>
| <loc_predicate>

<loc_predicate> ::
    loc[ <name> ] = <name>
| <name> is in <name>
| loc[ <name> ] <> <name>
| <name> is not in <name>

<discrete_boolean_predicate> ::
    <discrete_expression> <op_bool> <discrete_expression>
| <discrete_expression> in [ <discrete_expression> , <discrete_expression> ]
| <discrete_expression> in [ <discrete_expression> .. <discrete_expression> ]

<discrete_expression> ::
    <discrete_expression> + <discrete_term>
| <discrete_expression> - <discrete_term>
| <discrete_term>

<discrete_term> ::
    <discrete_term> * <discrete_factor>
| <discrete_term> / <discrete_factor>
| <discrete_factor>

<discrete_factor> ::
    <name>
| <positive_rational>
| ( <discrete_expression> )
| - <discrete_factor>

```

```

<op_bool> ::
  <
  | <=
  | =
  | <>
  | >=
  | >

<positive_rational> ::
  <pos_integer>
  | <pos_float>

<projection_definition_opt> ::
  projectresult( <name_nonempty_list> ) <semicolon_opt>
  | ε

<parameter_valuation> ::
  <parameter_assignments> <semicolon_opt>

<parameter_assignments> ::
  <parameter_assignment> <parameter_assignments>
  | ε

<parameter_assignment> ::
  <and_opt> <name> = <constant_arithmetic_expr> <comma_opt>

<hyper_rectangle> ::
  <rectangle_parameter_assignments> <semicolon_opt>

<rectangle_parameter_assignments> ::
  <rectangle_parameter_assignment> <rectangle_parameter_assignments>
  | ε

<rectangle_parameter_assignment> ::
  <and_opt> <name> = <constant_arithmetic_expr> . . <constant_arithmetic_expr> <comma_opt>
  | <and_opt> <name> = <constant_arithmetic_expr> <comma_opt>

<constant_arithmetic_expr> ::
  <constant_arithmetic_expr> + <constant_expr_mult>
  | <constant_arithmetic_expr> - <constant_expr_mult>
  | <constant_expr_mult>

<constant_expr_mult> ::
  <constant_expr_mult> * <constant_neg_atom>
  | <constant_expr_mult> / <constant_neg_atom>
  | <constant_neg_atom>

```

```

<constant_neg_atom> ::
| <constant_atom>
| - <constant_atom>

<constant_atom> ::
| ( <constant_arithmetic_expr> )
| <rational>

<step_opt> ::
| , step= <pos_rational>
| ε

<and_opt> ::
| &
| ε

<comma_opt> ::
| ,
| ε

<semicolon_opt> ::
| ;
| ε

```

*Remark 9.1.* In the parameter valuation definition ( $\langle parameter\_valuation \rangle$ ), all parameters of the model must be given a valuation; but the definition of the parameter valuation may also use names that do not appear in the model (a warning will just be issued).

*Remark 9.2.* In the hyper-rectangle definition ( $\langle hyper\_rectangle \rangle$ ), all parameters of the model must be given an interval (possibly punctual); again, the definition of the hyper-rectangle may also use names that do not appear in the model (a warning will just be issued).

## 9.4 Reserved words

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

#exhibit	and
#include	array_append
#synth	array_length
#witness	array_mem
accepting	automatically_generated_observer
always	automatically_generated_x_obs

automaton	nosync_obs
before	not
binary	once
clock	or
constant	parameter
discrete	pow
do	projectresult
else	property
end	queue_clear
eventually	queue_is_empty
everytime	queue_length
False	queue_pop
fill_left	queue_push
fill_right	queue_top
flow	rational_of_int
goto	seq
happened	sequence
has	shift_left
if	shift_right
in	special_0_clock
init	stack_clear
initially	stack_is_empty
invariant	stack_length
is	stack_pop
let	stack_push
list_cons	stack_top
list_hd	step
list_is_empty	stop
list_length	sync
list_mem	synclabs
list_rev	then
list_tl	True
loc	urgent
logand	var
lognot	wait
logor	when
logxor	while
next	within

## Chapter 10

# Missing features

Although we try to make IMITATOR as complete as possible, it misses some features, not implemented due to lack of time (contributors are welcome!) or due to complexity, or to keep the tool consistent. We enumerate in the following what seems to us to be the “most missing” features and, when applicable, we give hints to overcome these limitations.

### 10.1 ASAP transitions

ASAP (as soon as possible) transitions are transitions that can be taken as soon as all IPTA synchronizing with this transition can execute their local transition. This is different from urgent transitions, that must be taken in 0 time. Here, time can elapse, but not after all IPTA are ready to execute their local transition.

This is not supported by IMITATOR.

### 10.2 Parameterized models

Parameterized models are understood here as models with an arbitrary number of components (*e.g.*, Fischer’s mutual exclusion protocol with  $n$  processes), that would be instantiated (*e.g.*,  $n = 15$ ) before performing the analysis. IMITATOR does not currently support such parameterized models, and one should use copy/paste utilities to instantiate  $n$  models. For complicated models with many processes, we usually write short scripts to generate the model (a script [CSMACDgenerator.py](#) to model the varying part of parameterized models for the CSMA/CD case study is available on the IMITATOR project on [GitHub](#)).

### 10.3 Other synchronization models

One-to-one synchronization could possibly be simulated by using as many transitions as pairs of IPTA in the model, although this may make the model rather complex.

Broadcast synchronization (“only the IPTA ready to execute a given transition execute it”) is not supported. Once more, it could possibly be simulated by using as many transitions as subsets of IPTA in the model, although this will make the model definitely complex.

Message passing is not supported. This can be easily simulated using dedicated discrete variables, that would be read / written in the transition.

## 10.4 Initial intervals for discrete variables

Discrete variables must be set to a constant rational in the `init` definition (e.g., `i = 0`). Setting a variable to an arbitrarily value (e.g., `i in [0 .. 10]`) is currently not supported. This can be simulated using an initialization IPTA that nondeterministically sets `i` to any of the values, in 0 time so as to not disturb the model.

## 10.5 Complex updates for discrete variables

So far, discrete variables can only be set to arithmetic expressions in  $\mathcal{AE}(R)$ ; hence, assigning a discrete variable to a clock, or to a parameter, or to any more complex expression, is not allowed. A reason for this restriction is that the value of the discrete variables would not anymore be constant (recall that discrete variables are syntactic sugar for *locations*).

However, this can be (partially) simulated with stopwatches: we can replace a discrete variable with a clock that is stopped in all locations (*i.e.*, it does not evolve with time), and that is updated to the desired value (recall from [Definition 3.1](#) that the clock updates allow assignments to linear expressions over clocks, discrete variables and parameters). However, in this latter case, the non-linear power of arithmetic expressions (multiplications and divisions between variables) cannot be used anymore.

## 10.6 Synthesis for L/U-PTA

IMITATOR does not implement specific algorithms for lower-bound / upper-bound PTA (L/U-PTA). This subclass of PTA, introduced in [\[Hun+02\]](#), constrains parameters to appear either always as upper-bounds in inequalities comparing them with clocks, or always as lower-bounds. L/U-PTA benefit from some decidability results (see e.g., [\[Hun+02; BL09; JLR15; AL17; ALM20\]](#)); however, exact synthesis seems to be intractable in practice [\[JLR15; ALR16\]](#). Still, some subclasses for which exact synthesis can be performed were proposed [\[BL09; ALR18a\]](#).

However, IMITATOR does detect whether an input model is an L/U-PTA, in which case a message is printed with the list of lower-bounds parameters and upper-bounds parameters.

*Remark 10.1.* Note that our definition of L/U-PTAs is consistent with that of, e.g., [\[BL09\]](#); however, we do *not* consider the constraint of the initial state while checking for the L/U nature of the model. That is, we refer to as L/U-PTAs even for the *constrained* L/U-PTAs of [\[BL09\]](#).

## Chapter 11

# Acknowledgments

Étienne André initiated the development of IMITATOR in 2008, and keeps developing it. Emmanuelle Encrenaz and Laurent Fribourg have been great supporters 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 linking with Apron [JM09] in an early 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 in part the merging technique [AFS13], and brought several case studies. Giuseppe Lipari and Sun Youcheng provided examples from the real-time systems community, and collaborated on several algorithms. Camille Coti, Sami Evangelista and Nguyễn Hoàng Gia worked on the distributed version of IMITATOR. Nguyễn Hoàng Gia worked on the non-Zeno synthesis algorithms. Laure Petrucci and Jaco van de Pol implemented an NDFS algorithm. Stéphan Rosse improved a script for comparing results and speeds of various versions of IMITATOR. Vincent Bloemen implemented the optimal-time reachability synthesis. Jaime Arias simplified a lot the source code management and implemented new features. Jiří Srba suggested the translation of IMITATOR's strong broadcast into UPPAAL. Benjamin Loillier significantly enhanced the syntax, notably allowing new primitive types (Boolean, int, binary words), and composite types (arrays, lists, stacks, queues). Dylan Marinho reformatted the library of benchmarks [AMP21] and implemented the translation to JANI. Johan Arcile implemented the extrapolation [AA22].

We acknowledge the help of the PPL [BHZ08] developers for their help with solving several issues over the years, and more generally for making this very useful library available.

## Chapter 12

# Licensing and credits

### IMITATOR license

IMITATOR is free software available under the GNU GPL license.



### Contributors

The following people contributed to the development of IMITATOR.

Étienne André	2008 –
Johan Arcile	2021 – 2022
Jaime Arias	2018 –
Vincent Bloemen	2018
Camille Coti	2014
Daphne Dussaud	2010
Sami Evangelista	2014
Ulrich Kühne	2010 – 2011
Benjamin Loillier	2021 –
Dylan Marinho	2021 –
Nguyễn Hoàng Gia	2014 – 2018
Laure Petrucci	2019 –
Jaco van de Pol	2019 –
Romain Soulat	2010 – 2013

The following people contributed to the compiling, testing and packaging facilities.















Corentin Guillevic	2015
Sarah Hadbi	2015
Fabrice Kordon	2015
Alban Linard	2014 – 2015
Stéphane Rosse	2016 – 2017















## User manual

This user manual is available under the Creative Commons CC-BY-SA license.



## Graphics credits

Version	Image	Source	Author	License
IMITATOR		<a href="#">Typingmonkey.svg</a>	KaterBegemot	
2.x versions		<a href="#">Stick_of_butter.jpg</a>	Renee Comet	
version 2.7		<a href="#">Andouille-Scheiben.jpg</a>	Pwagenblast	
version 2.8		<a href="#">Schinken-roh.jpg</a>	Rainer Zenz	
version 2.9		<a href="#">Physalisperuviana.jpg</a>	Hans B. common-swiki (assumed?)	
version 2.10		<a href="#">méduses tentacles</a>	(?)	

version 2.11		Véritable Kouign Amann de Douarnenez	Haltopub	
version 2.12		Photo of a live lobster	Junior Libby	
version 3.0		French goat's cheese Banon cut to show the inside of the cheese	Tangopaso	
3.x versions		Swiss cheese vacherin	swissboy	
version 3.1		artichoke	cactus cowboy	
version 3.2		Fresh And Sweet Blueberries	Kathy Zinn	
version 3.3		A pile of loose caramels	Evan-Amos	

# References

- [AA22] Johan Arcile and Étienne André. “Zone extrapolations in parametric timed automata”. In: *Proceedings of the 14th NASA Formal Methods Symposium (NFM 2022)* (May 24–27, 2022). Ed. by Klaus Havelund, Jyo Deshmukh, and Ivan Perez. Lecture Notes in Computer Science. Caltech, Pasadena, CA, USA: Springer, 2022 (cit. on pp. 64, 87).
- [ABL98] Luca Aceto, Augusto Burgueño, and Kim Gulstrand Larsen. “Model Checking via Reachability Testing for Timed Automata”. In: *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems TACAS 1998* (Mar. 28–Apr. 4, 1998). Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Lisbon, Portugal: Springer, 1998, pp. 263–280. ISBN: 3-540-64356-7. DOI: [10.1007/BFb0054177](https://doi.org/10.1007/BFb0054177) (cit. on pp. 10, 38).
- [Ace+03] Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim Guldstrand Larsen. “The power of reachability testing for timed automata”. In: *Theoretical Computer Science* 300.1-3 (2003), pp. 411–475. DOI: [10.1016/S0304-3975\(02\)00334-1](https://doi.org/10.1016/S0304-3975(02)00334-1) (cit. on p. 38).
- [Ace+98] Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim Guldstrand Larsen. “The Power of Reachability Testing for Timed Automata”. In: *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1998)* (Dec. 17–19, 1998). Ed. by Vikraman Arvind and Ramaswamy Ramanujam. Vol. 1530. Lecture Notes in Computer Science. Chennai, India: Springer, 1998, pp. 245–256. ISBN: 3-540-65384-8. DOI: [10.1007/b71635](https://doi.org/10.1007/b71635) (cit. on pp. 10, 38).
- [ACE14] Étienne André, Camille Coti, and Sami Evangelista. “Distributed Behavioral Cartography of Timed Automata”. In: *Proceedings of the 21st European MPI Users’ Group Meeting (EuroMPI/ASIA 2014)* (Sept. 9–12, 2014). Ed. by Jack Dongarra, Yutaka Ishikawa, and Hori Atsushi. Kyoto, Japan: ACM, Sept. 2014, pp. 109–114. DOI: [10.1145/2642769.2642784](https://doi.org/10.1145/2642769.2642784) (cit. on pp. 45, 63).
- [ACN15] Étienne André, Camille Coti, and Hoang Gia Nguyen. “Enhanced Distributed Behavioral Cartography of Parametric Timed Automata”. In: *Proceedings of the 17th International Conference on Formal Engineering Methods (ICFEM 2015)* (Nov. 3–6, 2015). Ed. by Michael Butler, Sylvain Conchon, and Fatiha Zaïdi. Vol. 9407. Lecture Notes in Computer Science. Paris, France: Springer, Nov. 2015, pp. 319–335. ISBN: 978-3-319-25422-7. DOI: [10.1007/978-3-319-25423-4\\_21](https://doi.org/10.1007/978-3-319-25423-4_21) (cit. on pp. 45, 63).
- [AF10] Étienne André and Laurent Fribourg. “Behavioral Cartography of Timed Automata”. In: *Proceedings of the 4th Workshop on Reachability Problems in Computational Models (RP 2010)* (Aug. 28–29, 2010). Ed. by Antonín Kučera and Igor Potapov. Vol. 6227. Lecture Notes in Computer Science. Brno, Czech Republic: Springer, Aug. 2010, pp. 76–90. DOI: [10.1007/978-3-642-15349-5\\_5](https://doi.org/10.1007/978-3-642-15349-5_5) (cit. on pp. 5, 44, 45).

- [AFS13] Étienne André, Laurent Fribourg, and Romain Soulat. “Merge and Conquer: State Merging in Parametric Timed Automata”. In: *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA 2013)* (Oct. 15–18, 2013). Ed. by Dang-Van Hung and Mizuhito Ogawa. Vol. 8172. Lecture Notes in Computer Science. Ha Noi, Viet Nam: Springer, Oct. 2013, pp. 381–396. DOI: [10.1007/978-3-319-02444-8\\_27](https://doi.org/10.1007/978-3-319-02444-8_27) (cit. on pp. 37, 49, 65, 66, 87).
- [AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. “Parametric real-time reasoning”. In: *Proceedings of the 25th annual ACM symposium on Theory of computing (STOC 1993)* (May 16–18, 1993). Ed. by S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal. San Diego, California, United States: ACM, 1993, pp. 592–601. ISBN: 0-89791-591-7. DOI: [10.1145/167088.167242](https://doi.org/10.1145/167088.167242) (cit. on pp. 5, 6, 14, 17, 37).
- [AHW18] Étienne André, Ichiro Hasuo, and Masaki Waga. “Offline timed pattern matching under uncertainty”. In: *Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems (ICECCS 2018)* (Dec. 12–14, 2018). Ed. by Anthony Widjaja Lin and Jun Sun. Melbourne, Australia: IEEE Computer Society, 2018, pp. 10–20. DOI: [10.1109/ICECCS2018.2018.00010](https://doi.org/10.1109/ICECCS2018.2018.00010) (cit. on p. 67).
- [AL17] Étienne André and Didier Lime. “Liveness in L/U-Parametric Timed Automata”. In: *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD 2017)* (June 25–30, 2017). Ed. by Alex Legay and Klaus Schneider. Zaragoza, Spain: IEEE, 2017, pp. 9–18. DOI: [10.1109/ACSD.2017.19](https://doi.org/10.1109/ACSD.2017.19) (cit. on pp. 42, 86).
- [ALM20] Étienne André, Didier Lime, and Nicolas Markey. “Language Preservation Problems in Parametric Timed Automata”. In: *Logical Methods in Computer Science* 16.1 (Jan. 2020). DOI: [10.23638/LMCS-16\(1:5\)2020](https://doi.org/10.23638/LMCS-16(1:5)2020) (cit. on pp. 5, 43, 44, 86).
- [ALR15] Étienne André, Didier Lime, and Olivier H. Roux. “Integer-Complete Synthesis for Bounded Parametric Timed Automata”. In: *Proceedings of the 9th International Workshop on Reachability Problems (RP 2015)* (Sept. 21–23, 2015). Ed. by Mikołaj Bojańczyk, Sławomir Lasota, and Igor Potapov. Vol. 9328. Lecture Notes in Computer Science. Warsaw, Poland: Springer, Sept. 2015, pp. 7–19. DOI: [10.1007/978-3-319-24537-9\\_2](https://doi.org/10.1007/978-3-319-24537-9_2) (cit. on p. 64).
- [ALR16] Étienne André, Didier Lime, and Olivier H. Roux. “Decision Problems for Parametric Timed Automata”. In: *Proceedings of the 18th International Conference on Formal Engineering Methods (ICFEM 2016)* (Nov. 16–18, 2016). Ed. by Kazuhiro Ogata, Mark Lawford, and Shaoying Liu. Vol. 10009. Lecture Notes in Computer Science. Tokyo, Japan: Springer, 2016, pp. 400–416. DOI: [10.1007/978-3-319-47846-3\\_25](https://doi.org/10.1007/978-3-319-47846-3_25) (cit. on p. 86).
- [ALR18a] Étienne André, Didier Lime, and Mathias Ramparison. “TCTL model checking lower/upper-bound parametric timed automata without invariants”. In: *Proceedings of the 16th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2018)* (Sept. 4–6, 2018). Ed. by David N. Jansen and Pavithra Prabhakar. Vol. 11022. Lecture Notes in Computer Science. Beijing, China: Springer, 2018, pp. 1–17. DOI: [10.1007/978-3-030-00151-3\\_3](https://doi.org/10.1007/978-3-030-00151-3_3) (cit. on p. 86).
- [ALR18b] Étienne André, Didier Lime, and Mathias Ramparison. “Timed automata with parametric updates”. In: *Proceedings of the 18th International Conference on Application of Concurrency to System Design (ACSD 2018)* (June 24–29, 2018). Ed. by Gabriel Juhás, Thomas Chatain, and Radu Grosu. Bratislava, Slovakia: IEEE, 2018, pp. 21–29. DOI: [10.1109/ACSD.2018.000-2](https://doi.org/10.1109/ACSD.2018.000-2) (cit. on p. 17).
- [Alu+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. “The Algorithmic Analysis of Hybrid Systems”. In: *Theoretical Computer Science* 138.1 (1995), pp. 3–34. DOI: [10.1016/0304-3975\(94\)00202-T](https://doi.org/10.1016/0304-3975(94)00202-T) (cit. on p. 17).

- [AMP21] Étienne André, Dylan Marinho, and Jaco van de Pol. “A Benchmarks Library for Extended Timed Automata”. In: *Proceedings of the 15th International Conference on Tests and Proofs (TAP 2021)* (June 21–25, 2021). Ed. by Frédéric Loulergue and Franz Wotawa. Vol. 12740. Lecture Notes in Computer Science. virtual: Springer, 2021, pp. 39–50. DOI: [10.1007/978-3-030-79379-1\\_3](https://doi.org/10.1007/978-3-030-79379-1_3) (cit. on pp. 5, 87).
- [And+09] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. “An Inverse Method for Parametric Timed Automata”. In: *International Journal of Foundations of Computer Science* 20.5 (2009), pp. 819–836. DOI: [10.1142/S0129054109006905](https://doi.org/10.1142/S0129054109006905) (cit. on pp. 5, 43).
- [And+15] Étienne André, Giuseppe Lipari, Hoang Gia Nguyen, and Youcheng Sun. “Reachability Preservation Based Parameter Synthesis for Timed Automata”. In: *Proceedings of the 7th NASA Formal Methods Symposium (NFM 2015)* (Apr. 27–29, 2015). Ed. by Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 9058. Lecture Notes in Computer Science. Pasadena, CA, USA: Springer, Apr. 2015, pp. 50–65. DOI: [10.1007/978-3-319-17524-9\\_5](https://doi.org/10.1007/978-3-319-17524-9_5) (cit. on pp. 5, 45).
- [And+17] Étienne André, Hoang Gia Nguyen, Laure Petrucci, and Jun Sun. “Parametric model checking timed automata under non-Zenoness assumption”. In: *Proceedings of the 9th NASA Formal Methods Symposium (NFM 2017)* (May 16–18, 2017). Ed. by Clark Barrett and Temesghen Kahsai. Vol. 10227. Lecture Notes in Computer Science. Moffett Field, CA, USA: Springer, 2017, pp. 35–51. DOI: [10.1007/978-3-319-57288-8\\_3](https://doi.org/10.1007/978-3-319-57288-8_3) (cit. on p. 42).
- [And+19] Étienne André, Vincent Bloemen, Laure Petrucci, and Jaco van de Pol. “Minimal-Time Synthesis for Parametric Timed Automata”. In: *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019), Part II* (Apr. 8–11, 2019). Ed. by Tomáš Vojnar and Lijun Zhang. Vol. 11428. Lecture Notes in Computer Science. Prague, Czech Republic: Springer, 2019, pp. 211–228. DOI: [10.1007/978-3-030-17465-1\\_12](https://doi.org/10.1007/978-3-030-17465-1_12) (cit. on p. 38).
- [And+21] Étienne André, Jaime Arias, Laure Petrucci, and Jaco Van de Pol. “Iterative Bounded Synthesis for Efficient Cycle Detection in Parametric Timed Automata”. In: *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021)* (Mar. 27–Apr. 1, 2021). Ed. by Jan Friso Groote and Kim G. Larsen. Vol. 12651. Virtual: Springer, 2021, pp. 311–329. DOI: [10.1007/978-3-030-72016-2\\_17](https://doi.org/10.1007/978-3-030-72016-2_17) (cit. on pp. 5, 40).
- [And+22] Étienne André, Dylan Marinho, Laure Petrucci, and Jaco van de Pol. *Efficient Convex Zone Merging in Parametric Timed Automata*. Tech. rep. Submitted. 2022 (cit. on pp. 65, 66).
- [And13a] Étienne André. “Dynamic Clock Elimination in Parametric Timed Automata”. In: *Proceedings of the 1st French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013)* (July 15–16, 2013). Ed. by Christine Choppy and Jun Sun. Vol. 31. OpenAccess Series in Informatics (OASICS). Singapore: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, July 2013, pp. 18–31. DOI: [10.4230/OASICS.FSFMA.2013.18](https://doi.org/10.4230/OASICS.FSFMA.2013.18) (cit. on p. 64).
- [And13b] Étienne André. “Observer Patterns for Real-Time Systems”. In: *Proceedings of the 18th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2013)* (July 17–19, 2013). Ed. by Yang Liu and Andrew Martin. Singapore: IEEE Computer Society, July 2013, pp. 125–134. DOI: [10.1109/ICECCS.2013.26](https://doi.org/10.1109/ICECCS.2013.26) (cit. on pp. 38, 39).

- [And16] Étienne André. “Parametric Deadlock-Freeness Checking Timed Automata”. In: *Proceedings of the 13th International Colloquium on Theoretical Aspects of Computing (ICTAC 2016)* (Oct. 24–28, 2016). Ed. by Augusto Cesar Alves Sampaio and Farn Wang. Vol. 9965. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, 2016, pp. 469–478. DOI: [10.1007/978-3-319-46750-4\\_27](https://doi.org/10.1007/978-3-319-46750-4_27) (cit. on pp. 5, 39).
- [And21] Étienne André. “IMITATOR 3: Synthesis of timing parameters beyond decidability”. In: *Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV 2021)* (July 18–23, 2021). Ed. by Rustan Leino and Alexandra Silva. Vol. 12759. Lecture Notes in Computer Science. virtual: Springer, 2021, pp. 1–14. DOI: [10.1007/978-3-030-81685-8\\_26](https://doi.org/10.1007/978-3-030-81685-8_26) (cit. on p. 5).
- [ANP17] Étienne André, Hoang Gia Nguyen, and Laure Petrucci. “Efficient parameter synthesis using optimized state exploration strategies”. In: *Proceedings of the 22nd International Conference on Engineering of Complex Computer Systems (ICECCS 2017)* (Nov. 6–8, 2017). Ed. by Zhenjiang Hu and Guangdong Bai. Fukuoka, Japan: IEEE, 2017, pp. 1–10. DOI: [10.1109/ICECCS.2017.28](https://doi.org/10.1109/ICECCS.2017.28) (cit. on p. 64).
- [AS11] Étienne André and Romain Soulat. “Synthesis of Timing Parameters Satisfying Safety Properties”. In: *Proceedings of the 5th Workshop on Reachability Problems in Computational Models (RP 2011)* (Sept. 28–30, 2011). Ed. by Giorgio Delzanno and Igor Potapov. Vol. 6945. Lecture Notes in Computer Science. Genova, Italy: Springer, Sept. 2011, pp. 31–44. DOI: [10.1007/978-3-642-24288-5\\_5](https://doi.org/10.1007/978-3-642-24288-5_5) (cit. on p. 44).
- [Beh+03] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim Guldstrand Larsen. “Static Guard Analysis in Timed Automata Verification”. In: *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2003)* (Apr. 7–11, 2003). Ed. by Hubert Garavel and John Hatcliff. Vol. 2619. Lecture Notes in Computer Science. Warsaw, Poland: Springer, 2003, pp. 254–277. DOI: [10.1007/3-540-36577-X\\_18](https://doi.org/10.1007/3-540-36577-X_18) (cit. on p. 64).
- [Beh+06] Gerd Behrmann, Patricia Bouyer, Kim Guldstrand Larsen, and Radek Pelánek. “Lower and upper bounds in zone-based abstractions of timed automata”. In: *International Journal on Software Tools for Technology Transfer* 8.3 (2006), pp. 204–215. DOI: [10.1007/s10009-005-0190-0](https://doi.org/10.1007/s10009-005-0190-0) (cit. on p. 64).
- [BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. “The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems”. In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21. DOI: [10.1016/j.scico.2007.08.001](https://doi.org/10.1016/j.scico.2007.08.001) (cit. on pp. 58, 87).
- [BL09] Laura Bozzelli and Salvatore La Torre. “Decision problems for lower/upper bound parametric timed automata”. In: *Formal Methods in System Design* 35.2 (2009), pp. 121–151. DOI: [10.1007/s10703-009-0074-0](https://doi.org/10.1007/s10703-009-0074-0) (cit. on p. 86).
- [Bou+04] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. “Updatable Timed Automata”. In: *Theoretical Computer Science* 321.2–3 (Aug. 2004), pp. 291–345. DOI: [10.1016/j.tcs.2004.04.003](https://doi.org/10.1016/j.tcs.2004.04.003) (cit. on p. 17).
- [Bud+17] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. “JANI: Quantitative Model and Tool Interaction”. In: *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017), Part II, Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2017)* (Apr. 22–29, 2017). Ed. by Axel Legay and Tiziana Margaria. Vol. 10206. Lecture Notes in Computer Science. Uppsala, Sweden, 2017, pp. 151–168. DOI: [10.1007/978-3-662-54580-5\\_9](https://doi.org/10.1007/978-3-662-54580-5_9) (cit. on p. 55).





- [CL00] Franck Cassez and Kim Guldstrand Larsen. “The Impressive Power of Stopwatches”. In: *CONCUR* (Aug. 22–25, 2000). Ed. by Catuscia Palamidessi. Vol. 1877. Lecture Notes in Computer Science. University Park, PA, USA: Springer, 2000, pp. 138–152. DOI: [10.1007/3-540-44618-4\\_12](https://doi.org/10.1007/3-540-44618-4_12) (cit. on p. 17).
- [HHW95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. “A User Guide to HyTech”. In: *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1995)* (May 19–20, 1995). Ed. by Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen. Vol. 1019. Lecture Notes in Computer Science. Aarhus, Denmark: Springer, 1995, pp. 41–71. ISBN: 3-540-60630-0. DOI: [10.1007/3-540-60630-0\\_3](https://doi.org/10.1007/3-540-60630-0_3) (cit. on pp. 6, 18, 54).
- [Hun+02] Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. “Linear parametric model checking of timed automata”. In: *Journal of Logic and Algebraic Programming* 52-53 (2002), pp. 183–220. DOI: [10.1016/S1567-8326\(02\)00037-1](https://doi.org/10.1016/S1567-8326(02)00037-1) (cit. on p. 86).
- [IMI-Web] IMITATOR. *IMITATOR Web page*. <https://www.imitator.fr>. 2022 (cit. on pp. 5, 60).
- [JLR15] Aleksandra Jovanović, Didier Lime, and Olivier H. Roux. “Integer Parameter Synthesis for Real-Time Systems”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 445–461. DOI: [10.1109/TSE.2014.2357445](https://doi.org/10.1109/TSE.2014.2357445) (cit. on pp. 5, 37, 86).
- [JM09] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)* (June 26–July 2, 2009). Vol. 5643. Lecture Notes in Computer Science. Grenoble, France: Springer, 2009, pp. 661–667. DOI: [10.1007/978-3-642-02658-4\\_52](https://doi.org/10.1007/978-3-642-02658-4_52) (cit. on p. 87).
- [Li09] Guangyuan Li. “Checking Timed Büchi Automata Emptiness Using LU-Abstractions”. In: *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2009)* (Sept. 14–16, 2009). Ed. by Joël Ouaknine and Frits W. Vaandrager. Vol. 5813. Lecture Notes in Computer Science. Budapest, Hungary: Springer, 2009, pp. 228–242. DOI: [10.1007/978-3-642-04368-0\\_18](https://doi.org/10.1007/978-3-642-04368-0_18) (cit. on p. 64).
- [LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a Nutshell”. In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (1997), pp. 134–152. DOI: [10.1007/s100090050010](https://doi.org/10.1007/s100090050010) (cit. on pp. 18, 51).
- [NPV18] Hoang Gia Nguyen, Laure Petrucci, and Jaco Van de Pol. “Layered and Collecting NDFS with Subsumption for Parametric Timed Automata”. In: *Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems (ICECCS 2018)* (Dec. 12–14, 2018). Ed. by Anthony Widjaja Lin and Jun Sun. Melbourne, Australia: IEEE Computer Society, Dec. 2018, pp. 1–9. DOI: [10.1109/ICECCS2018.2018.00009](https://doi.org/10.1109/ICECCS2018.2018.00009) (cit. on pp. 5, 40, 41, 65, 68).
- [Sun+09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. “PAT: Towards Flexible Verification under Fairness”. In: *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)* (June 26–July 2, 2009). Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Grenoble, France: Springer, 2009, pp. 709–714. ISBN: 978-3-642-02657-7. DOI: [10.1007/978-3-642-02658-4\\_59](https://doi.org/10.1007/978-3-642-02658-4_59) (cit. on p. 6).
- [Tri09] Stavros Tripakis. “Checking timed Büchi automata emptiness on simulation graphs”. In: *ACM Transactions on Computational Logic* 10.3 (2009), 15:1–15:19. DOI: [10.1145/1507244.1507245](https://doi.org/10.1145/1507244.1507245) (cit. on p. 64).
- [Wan+15] Ting Wang, Jun Sun, Xinyu Wang, Yang Liu, Yuanjie Si, Jin Song Dong, Xiaohu Yang, and Xiaohong Li. “A Systematic Study on Explicit-State Non-Zenoness Checking for Timed Automata”. In: *IEEE Transactions on Software Engineering* 41.1 (2015), pp. 3–18. DOI: [10.1109/TSE.2014.2359893](https://doi.org/10.1109/TSE.2014.2359893) (cit. on p. 42).